Power ISA™ Version 2.07

May 3, 2013

Softcopy Distribution: www.power.org.documentation

IBM®

Printed in the United States of America January 2015

By downloading the POWER® Instruction set Architecture ("ISA") Specification, you agree to be bound by the terms and conditions of this agreement.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

Note: This document contains information on products in the design, sampling and/or initial production phases of development. This information is subject to change without notice. Verify with your IBM field applications engineer that you have the latest version of this document before finalizing a design.

You may use this documentation solely for developing technology products compatible with Power Architecture® in support of growing the POWER ecosystem. You may not modify this documentation. You may distribute the documentation to suppliers and other contractors hired by you solely to produce your technology products compatible with Power Architecture® technology and to your customers (either directly or indirectly through your resellers) in conjunction with their use and instruction of your technology products compatible with Power Architecture® technology. This agreement does not include rights to create a CPU design to run the POWER ISA unless such rights have been granted by IBM under a separate agreement. The POWER ISA specification is protected by copyright and the practice or implementation of the information herein may be protected by one or more patents or pending patent applications. No other license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. IBM makes no representations or warranties, either express or implied, including but not limited to, warranties of merchantability, fitness for a particular purpose, or non-infringement, or that any practice or implementation of the IBM documentation will not infringe any third party patents, copyrights, trade secrets, or other rights. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com®**.

The following paragraph does not apply to the United Kingdom or any country or state where such provisions are inconsistent with local law.

The specifications in this manual are subject to change without notice. This manual is provided “AS IS”. International Business Machines Corp. makes no warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

International Business Machines Corp. does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

Address comments to IBM Corporation, 11400 Burnett Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM®
Power ISA
PowerPC®
Power Architecture
PowerPC Architecture
Power Family
RISC/System 6000®
POWER®
POWER2
POWER4
POWER4+
POWER5
POWER5+
POWER6®
POWER7®
System/370
System z

The POWER ARCHITECTURE and POWER.ORG. word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

AltiVec is a trademark of Freescale Semiconductor, Inc. used under license.

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication or disclosure is subject to restrictions set fourth in GSA ADP Schedule Contract with IBM Corporation.

Preface

The roots of the Power ISA (Instruction Set Architecture) extend back over a quarter of a century, to IBM Research. The POWER (Performance Optimization With Enhanced RISC) Architecture was introduced with the RISC System/6000 product family in early 1990. In 1991, Apple, IBM, and Motorola began the collaboration to evolve to the PowerPC Architecture, expanding the architecture's applicability. In 1997, Motorola and IBM began another collaboration, focused on optimizing PowerPC for embedded systems, which produced Book E.

In 2006, Freescale and IBM collaborated on the creation of the Power ISA Version 2.03, which represented the reunification of the architecture by combining Book E content with the more general purpose PowerPC Version 2.02. A significant benefit of the reunification is the establishment of a single, compatible, 64-bit programming model. The combining also extends explicit architectural endorsement and control to Auxiliary Processing Units (APUs), units of function that were originally developed as implementation- or product family-specific extensions in the context of the Book E allocated opcode space. With the resulting architectural superset comes a framework that clearly establishes requirements and identifies options.

To a very large extent, application program compatibility has been maintained throughout the history of the architecture, with the main exception being application exploitation of APUs. The framework identifies the base, pervasive, part of the architecture, and differentiates it from "categories" of optional function (see Section 1.3.5 of Book I). Because of the substantial differences in the supervisor (privileged) architecture that developed as Book E was optimized for embedded systems, the supervisor architectures for embedded and general purpose implementations are represented as mutually exclusive categories. Future versions of the architecture will seek to converge on a common solution where possible.

This document defines the Power ISA Version 2.07. It is comprised of five books and a set of appendices.

Book I, *Power ISA User Instruction Set Architecture*, covers the base instruction set and related facilities available to the application programmer. It includes five chapters derived from APU function, including the vector extension also known as Altivec.

Book II, *Power ISA Virtual Environment Architecture,* defines the storage model and related instructions and facilities available to the application programmer.

Book III-S, *Power ISA Operating Environment Architecture,* defines the supervisor instructions and related facilities used for general purpose implementations.

Book III-E, *Power ISA Operating Environment Architecture,* defines the supervisor instructions and related facilities used for embedded implementations. It was derived from Book E and extended to include APU function.

Book VLE, *Power ISAVariable Length Encoded Instructions Architecture,* defines alternative instruction encodings and definitions intended to increase instruction density for very low end implementations. It was derived from an APU description developed by Freescale Semiconductor.

As used in this document, the term "Power ISA" refers to the instructions and facilities described in Books I, II, III-S, III-E, and VLE.

Usage of the phrase "Book III" refers to both Book III-S and Book III-E. An exception to this rule is when, at the beginning of a Section or Book, it is specified that usage of the phrase "Book III" implies only either "Book III-S" or "Book III-E".

Change bars have been included to indicate changes from the Power ISA Version 2.06B.

Summary of Changes in Power ISA Version 2.07

Version 2.07 of the PowerISA was created by applying the following requests for change (RFCs) to PowerISA Version 2.06B.

Performance Monitor Facility: Adds various performance monitoring facilities and a branch history buffer to Server architecture.

VSX Scalar Single-Precision: Adds support for scalar single-precision to VSX.

Transactional Memory: Adds support for a transactional memory storage model which allows an application to perform a sequence of accesses that appear to occur atomically with respect to other threads.

Processor Control Enhancements: Enables privileged and hypervisor software to send messages to other threads.

Instruction Cache Block Touch: The ***icbt*** instruction has been moved from the Embedded category to the Base category.

Extended Problem State Priority: Provides a mechanism that enables application programs to temporarily boost their priority.

Virtual Page Class Key Extensions for Instructions: Adds a new SPR similar to the AMR that controls whether instructions can be fetched from virtual addresses.

Reserved Bit Behavior Restriction and Processor Compatibility Register: Updates the PCR to accommodate new problem state features for PowerISA Version 2.07. In addition, requirements for reserved bit behaviors have been tightened in order to improve software compatibility across implementations.

Chip Information Register: Adds a privileged SPR which enables software to determine information about the chip on which the processor is implemented.

Crypto Operations: Adds instructions supporting AES, GCM, and SHA encryption and decryption, as well as a variety of CRCs and other finite field arithmetic operations.

VMX 64-Bit Integer Operations: Adds instructions supporting 2-way SIMD 64-bit integer operations.

Vector Miscellaneous Instructions: Adds SIMD count leading zeros instructions and a bit gather instruction.

Cache Hint Indicating Block Not Needed: Enables software to provide a cache hint indicating that it will no longer access a block.

Remove $LPES_1$ from the LPCR: Eliminates the capability to request the processor to behave as previous versions of the architecture required when $LPES_1 = 0$.

Split Endian Control: Adds a new MSR bit (SLE) as well as the ***mtsle*** instruction which allows its manipulation in problem state. This bit now allows the Endian values used for data and instruction storage accesses to be specified independently.

Direct Move Instructions: Adds new VSX instructions which remove the restriction of cross-register file moves.

Move dnh from Enhanced Debug to Embedded Category: The dnh instruction has been moved to the Embedded category.

Real Mode Storage Control for Instruction Fetching and Loosely-Related Caching-Inhibited Load/Store Changes: Defines an alternative to the existing instruction fetch RMSC approach whereby a first access to a region of storage will be performed as guarded, but subsequent accesses will be performed as non-guarded based on the success of the first.

Elemental Memory Barriers: Extends the definition of the existing ***sync*** instruction by adding several memory ordering functions.

Event-Based Branch Facility: A problem-state accessible event-based branch mechanism analogous to the interrupt mechanism is defined.

Stream Prefetch Changes: Allows DSCR access in problem state and adds additional stream prefetch functionality.

Add lqarx/stqcx. Instructions: Adds support for quadword atomic storage operations.

Allow lq/stq in Problem State: Allows problem state software to use the ***lq*** and ***stq*** instructions.

Allow lq/stq in Little-Endian Mode: Removes the restriction that the ***lq*** and ***stq*** instructions can only operate in Big-Endian mode.

Add makeitso Instruction: Adds the ***miso*** extended mnemonic which allows producers in producer-consumer applications to provide a hint to push store data out.

Branch Conditional to Target Address Register: Defines a new instruction which branches conditionally to an address contained in a new SPR.

Remove DABR[X], add DAWR[X] and IABR SPRs: Removes and replaces existing DABR/DABRX SPRs with enhanced watchpoint functionality. Additionally adds instruction address breakpoint capability to the ISA.

Facility Availability Registers and Interrupts: A privileged register and a hypervisor register that enables various facilities are defined.

Reserved SPRs: Defines a set of reserved SPRs treated as no-ops in the current architecture so that

exploitation of new function in future designs can proceed more quickly and pervasively.

Instruction Counter and Virtual Time Base: Two registers, one that counts instructions completed by a thread, and another that counts at the same rate as the Time Base are added.

Architecture Changes to Support Program Portability: Eliminates software exposure to errors caused by variations in behavior due to implementation-dependent bits in CTRL and PPR.

Miscellaneous Changes: Various minor editorial corrections are made.

Interrupts and Relocation, MMIO Emulation: NewLPCR bits enable most interrupts to be taken with relocation on, and virtual page class key faults to cause Hypervisor Data Storage interrupts.

Guest Timer Interrupts and Facilities: Adds guest timer facilities to enable performance measurements for guest operating systems.

VSX Unaligned Vector Storage Accesses: Change VSX vector storage access instructions to support to byte-aligned addresses. Simplify Table 2, “Effect of Operand Placement on Performance,” on page 753 in Book II.

VMX Miscellaneous Operations II: Adds new instructions ***vclzd***, ***vpopcntb***, ***vpopcnth***, ***vpopcntw***, ***vpopcntd***, ***veqv***, ***vnand***, and ***vorc***..

BFP/VSX Miscellaneous Operations: Introduces new VSX instructions to address IEEE-754-2008 compliance when performing simple assignments between single-precision scalar and vector elements.

VMX 32-bit Multiply Operations: Introduces 4-way SIMD 32-bit integer multiply instructions.

VMX Decimal Integer Operations: Introduces new packed decimal add and subtract instructions.

Remove Data Value Compare: Since the data value compare function can be easily emulated within a data address compare handler, the data value compare registers are removed from Book III-E.

VMX 128-bit Integer Operations: Introduces new quadword integer add and subtract instructions.

Cache Lock Query Instructions: Adds instructions to determine whether a cache block has been successfully locked with a preceding cache locking instruction.

Embedded Guest Perforamnce Monitor Interrupt: Introduces a new Performance Monitor Interrupt that enables a Performance Monitor interrupt to be taken in guest state.

Table of Contents

Figures

Book III-E:

Power ISA Operating Environment Architecture - Embedded Environment [Category: Embedded] 1021

Book I:

Power ISA User Instruction Set Architecture

Chapter 1. Introduction

1.1 Overview

This chapter describes computation modes, document conventions, a processor overview, instruction formats, storage addressing, and instruction fetching.

1.2 Instruction Mnemonics and Operands

The description of each instruction includes the mnemonic and a formatted list of operands. Some examples are the following.

```
stw     RS,D(RA)
addis   RT,RA,SI
```

Power ISA-compliant Assemblers will support the mnemonics and operand lists exactly as shown. They should also provide certain extended mnemonics, such as the ones described in Appendix E of Book I.

1.3 Document Conventions

1.3.1 Definitions

The following definitions are used throughout this document.

- **program**
  A sequence of related instructions.
- **application program**
  A program that uses only the instructions and resources described in Books I and II.
- **processor**
  The hardware component that implements the instruction set, storage model, and other facilities defined in the Power ISA architecture, and executes the instructions specified in a program.
- **quadwords, doublewords, words, halfwords, and bytes**
  128 bits, 64 bits, 32 bits, 16 bits, and 8 bits, respectively.
- **positive**
  Means greater than zero.
- **negative**
  Means less than zero.
- **floating-point single format** (or simply **single format**)
  Refers to the representation of a single-precision binary floating-point value in a register or storage.
- **floating-point double format** (or simply **double format**)
  Refers to the representation of a double-precision binary floating-point value in a register or storage.
- **system library program**
  A component of the system software that can be called by an application program using a *Branch* instruction.
- **system service program**
  A component of the system software that can be called by an application program using a *System Call* instruction.
- **system trap handler**
  A component of the system software that receives control when the conditions specified in a *Trap* instruction are satisfied.
- **system error handler**
  A component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of error. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.
- **latency**
  Refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.
- **unavailable**
  Refers to a resource that cannot be used by the program. For example, storage is unavailable if access to it is denied. See Book III.

- **undefined value**
  May vary between implementations, and between different executions on the same implementation, and similarly for register contents, storage contents, etc., that are specified as being undefined.
- **boundedly undefined**
  The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary finite sequence of instructions (none of which yields boundedly undefined results) in the state the processor was in before executing the given instruction. Boundedly undefined results may include the presentation of inconsistent state to the system error handler as described in Section 1.9.1 of Book II. Boundedly undefined results for a given instruction may vary between implementations, and between different executions on the same implementation.
- **"must"**
  If software violates a rule that is stated using the word "must" (e.g., "this field must be set to 0"), the results are boundedly undefined unless otherwise stated.
- **sequential execution model**
  The model of program execution described in Section 2.2, "Instruction Execution Order" on page 29.
- **Auxiliary Processor**
  An implementation-specific processing unit. Previous versions of the architecture use the term Auxiliary Processing Unit (APU) to describe this extension of the architecture. Architectural support for auxiliary processors is part of the Embedded category.
- **virtualized implementation**
  An implementation of the Power Architecture created by hypervisor software. A guest operating system sees a virtualized implementation of the Power ISA. Architectural support for virtualized implementations is part of the Embedded category (see Section 1.3.5, "Categories").

1.3.2 Notation

The following notation is used throughout the Power ISA documents.

- All numbers are decimal unless specified in some special way.
  - 0bnnnn means a number expressed in binary format.
  - 0xnnnn means a number expressed in hexadecimal format.

  Underscores may be used between digits.
- RT, RA, R1, ... refer to General Purpose Registers.
- FRT, FRA, FR1, ... refer to Floating-Point Registers.
- FRTp, FRAp, FRBp, ... refer to an even-odd pair of Floating-Point Registers. Values must be even, otherwise the instruction form is invalid.
- VRT, VRA, VR1, ... refer to Vector Registers.
- (x) means the contents of register x, where x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields. Names such as LR and CTR denote registers, not fields, so parentheses are not used with them. Parentheses are also omitted when register x is the register into which the result of an operation is placed.
- (RA|0) means the contents of register RA if the RA field has the value 1-31, or the value 0 if the RA field is 0.
- Bytes in instructions, fields, and bit strings are numbered from left to right, starting with byte 0 (most significant).
- Bits in registers, instructions, fields, and bit strings are specified as follows. In the last three items (definition of X_p etc.), if X is a field that specifies a GPR, FPR, or VR (e.g., the RS field of an instruction), the definitions apply to the register, not to the field.
  - Bits in instructions, fields, and bit strings are numbered from left to right, starting with bit 0
  - For all registers except the Vector category, bits in registers that are less than 64 bits start with bit number 64-L, where L is the register length; for the Vector category, bits in registers that are less than 128 bits start with bit number 128-L.
  - The leftmost bit of a sequence of bits is the most significant bit of the sequence.
  - X_p means bit p of register/instruction/field/bit_string X.
  - $X_{p:q}$ means bits p through q of register/instruction/field/bit_string X.
  - $X_{p\ q\ ...}$ means bits p, q, ... of register/instruction/field/bit_string X.
- ¬(RA) means the one's complement of the contents of register RA.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution.
- The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.
- x^n means x raised to the n^{th} power.

- ^{n}x means the replication of x, n times (i.e., x concatenated to itself n - 1 times). ^{n}0 and ^{n}1 are special cases:
  - ^{n}0 means a field of n bits with each bit equal to 0. Thus ^{5}0 is equivalent to 0b00000.
  - ^{n}1 means a field of n bits with each bit equal to 1. Thus ^{5}1 is equivalent to 0b11111.
- Each bit and field in instructions, and in status and control registers (e.g., XER, FPSCR) and Special Purpose Registers, is either defined or reserved. Some defined fields contain reserved values. In such cases when this document refers to the specific field, it refers only to the defined values, unless otherwise specified.
- /, //, ///, ... denotes a reserved field, in a register, instruction, field, or bit string.
- ?, ??, ???, ... denotes an implementation-dependent field in a register, instruction, field or bit string.

1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs

Reserved fields in instructions are ignored by the processor. This is a requirement in the Server environment and is being phased into the Embedded environment.

In some cases a defined field of an instruction has certain values that are reserved. This includes cases in which the field is shown in the instruction layout as containing a particular value; in such cases all other values of the field are reserved. In general, if an instruction is coded such that a defined field contains a reserved value the instruction form is invalid; see Section 1.8.2 on page 22. The only exceptions to the preceding rule is that it does not apply to Reserved and Illegal classes of instructions (see Section 1.7) or to portions of defined fields that are specified, in the instruction description, as being treated as reserved fields.

To maximize compatibility with future architecture extensions, software must ensure that reserved fields in instructions contain zero and that defined fields of instructions do not contain reserved values.

The handling of reserved bits in System Registers (e.g., XER, FPSCR) depends on whether the processor is in problem state. Unless otherwise stated, software is permitted to write any value to such a bit. In problem state, a subsequent reading of the bit returns 0 regardless of the value written; in privileged states, a subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

In some cases, a defined field of a System Register has certain values that are reserved. Software must not set a defined field of a System Register to a reserved value. References elsewhere in this document to a defined field (in an instruction or System Register) that has reserved values assume the field does not contain a reserved value, unless otherwise stated or obvious from context.

In some cases, a given bit of a System Register is specified to be set to a constant value by a given instruction or event. Unless otherwise stated or obvious from context, software should not depend on this constant value because the bit may be assigned a meaning in a future version of the architecture.

The reserved SPRs include SPRs 808, 809, 810, and 811. ***mtspr*** and ***mfspr*** instructions specifying these SPRs are treated as noops. Reserved SPRs are provided in the architecture to anticipate the eventual adoption of performance hint functionality that must be controlled by SPRs. Control of these capabilities using reserved SPRs will allow software to use these new capabilities on new implementations that support them while remaining compatible with existing implementations that may not support the new functionality.

Reserved SPRs are not assigned names. There are no individual descriptions of reserved SPRs in this document.

Assembler Note

Assemblers should report uses of reserved values of defined fields of instructions as errors.

Programming Note

It is the responsibility of software to preserve bits that are now reserved in System Registers, because they may be assigned a meaning in some future version of the architecture.

In order to accomplish this preservation in implementation-independent fashion, software should do the following.

- Initialize each such register supplying zeros for all reserved bits.
- Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

The XER and FPSCR are partial exceptions to this recommendation. Software can alter the status bits in these registers, preserving the reserved bits, by executing instructions that have the side effect of altering the status bits. Similarly, software can alter any defined bit in the FPSCR by executing a Floating-Point Status and Control Register instruction. Using such instructions is likely to yield better performance than using the method described in the second item above.

1.3.4 Description of Instruction Operation

Instruction descriptions (including related material such as the introduction to the section describing the instructions) mention that the instruction may cause a system error handler to be invoked, under certain conditions, if and only if the system error handler may treat the case as a programming error. (An instruction may cause a system error handler to be invoked under other conditions as well; see Chapter 6 of Book III-S and Chapter 7 of Book III-E).

A formal description is given of the operation of each instruction. In addition, the operation of most instructions is described by a semiformal language at the register transfer level (RTL). This RTL uses the notation given below, in addition to the notation described in Section 1.3.2. Some of this notation is also used in the formal descriptions of instructions. RTL notation not summarized here should be self-explanatory.

The RTL descriptions cover the normal execution of the instruction, except that "standard" setting of status registers, such as the Condition Register, is not shown. ("Non-standard" setting of these registers, such as the setting of the Condition Register by the *Compare* instructions, is shown.) The RTL descriptions do not cover cases in which the system error handler is invoked, or for which the results are boundedly undefined.

The RTL descriptions specify the architectural transformation performed by the execution of an instruction. They do not imply any particular implementation.

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_{iea}$	Assignment of an instruction effective address. In 32-bit mode the high-order 32 bits of the 64-bit target address are set to 0.
$\neg$	NOT logical operator
+	Two's complement addition
-	Two's complement subtraction, unary minus
$\times$	Multiplication
$\times_{si}$	Signed-integer multiplication
$\times_{ui}$	Unsigned-integer multiplication
/	Division
$\div$	Division, with result truncated to integer
$\sqrt{}$	Square root
=, $\neq$	Equals, Not Equals relations
<, $\leq$, >, $\geq$	Signed comparison relations
$<^u$, $>^u$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
$\oplus$, $\equiv$	Exclusive OR, Equivalence logical operators (($a \equiv b$) = ($a \oplus \neg b$))
ABS(x)	Absolute value of x
BCD_TO_DPD(x)	The low-order 24 bits of x contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the result. See Section B.1, "BCD-to-DPD Translation".
CEIL(x)	Least integer $\geq$ x
DCR(x)	Device Control Register x <E.DC>
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format, using the model shown on page 133
DPD_TO_BCD(x)	The low-order 20 bits of x contain two declets which are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the result. See Section B.2, "DPD-to-BCD Translation".
EXTS(x)	Result of extending x on the left with sign bits
FLOOR(x)	Greatest integer $\leq$ x
GPR(x)	General Purpose Register x

MASK(x, y) Mask having 1s in positions x through y (wrapping if x > y) and 0s elsewhere

MEM(x, y) Contents of a sequence of y bytes of storage. The sequence depends on the byte ordering used for storage access, as follows.
Big-Endian byte ordering:
The sequence starts with the byte at address x and ends with the byte at address x+y-1.
Little-Endian byte ordering:
The sequence starts with the byte at address x+y-1 and ends with the byte at address x.

MEM_DECORATED(x,y,z)
Contents of a sequence of y bytes of storage, where the storage is accessed with decoration z applied. The sequence depends on the byte ordering used for storage access, as follows.
Big-Endian byte ordering:
The sequence starts with the byte at address x and ends with the byte at address x+y-1.
Little-Endian byte ordering:
The sequence starts with the byte at address x+y-1 and ends with the byte at address x.

MEM_NOTIFY(x,z)
The decoration z is sent to storage location x.

$ROTL_{64}$(x, y)
Result of rotating the 64-bit value x left y positions

$ROTL_{32}$(x, y)
Result of rotating the 64-bit value x||x left y positions, where x is 32 bits long

SINGLE(x) Result of converting x from floating-point double format to floating-point single format, using the model shown on page 137

SPR(x) Special Purpose Register x

switch/case/default
switch/case/default statement, indenting shows range. The clause after “switch” specifies the expression to evaluate. The clause after “case” specifies individual values for the expression, followed by a colon, followed by the actions that are taken if the evaluated expression has any of the specified values. “default” is optional. If present, it must follow all the “case” clauses. The clause after “default” starts with a colon, and specifies the actions that are taken if the evaluated expression does not have any of the values specified in the preceding case statements.

TRAP Invoke the system trap handler

characterization
Reference to the setting of status bits, in a standard way that is explained in the text

undefined An undefined value.

CIA Current Instruction Address, which is the 64-bit address of the instruction being described by a sequence of RTL. Used by relative branches to set the Next Instruction Address (NIA), and by Branch instructions with LK=1 to set the Link Register. Does not correspond to any architected register.

NIA Next Instruction Address, which is the 64-bit address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, this is indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching (see Book III), the RTL is similar. For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA+4 (VLE behavior is different; see Book VLE). Does not correspond to any architected register.

if... then... else...
Conditional execution, indenting shows range; else is optional.

do Do loop, indenting shows range. “To” and/or “by” clauses specify incrementing an iteration variable, and a “while” clause gives termination conditions.

leave Leave innermost do loop, or do loop described in leave statement.

for For loop, indenting shows range. Clause after “for” specifies the entities for which to execute the body of the loop.

The precedence rules for RTL operators are summarized in Table 1. Operators higher in the table are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, - associates from left to right, so a-b-c = (a-b)-c.) Parentheses are used to override the evaluation order implied by the table or to increase clarity; parenthesized expressions are evaluated before serving as operands.

Table 1: Operator precedence

Operators	Associativity
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	*right to left*
unary -, ¬	*right to left*
×, ÷	left to right
+, -,	left to right
||	left to right
=, ≠, <, ≤, >, ≥,$<^u$, $>^u$,?	left to right

Table 1: Operator precedence

Operators	Associativity
&, ⊕, ≡	left to right
|	left to right
: (range)	none
←,$\leftarrow_{iea}$	none

1.3.5 Categories

Each facility (including registers and fields therein) and instruction is in exactly one of the categories listed in Figure 1.

A category may be defined as a *dependent category.* These are categories that are supported only if the category they are dependent on is also supported. Dependent categories are identified by the "." in their category name, e.g., if an implementation supports the Floating-Point.Record category, then the Floating-Point category is also supported.

An implementation that supports a facility or instruction in a given category, except for the two categories

Category	Abvr.	Notes
Base	B	Required for all implementations
Server	S	Required for Server implementations
Embedded	E	Required for Embedded implementations
Alternate Time Base	ATB	An additional Time Base; see Book II
Cache Specification	CS	Specify a specific cache for some instructions; see Book II
Decimal Floating-Point	DFP	Decimal Floating-Point facilities
Decorated Storage	DS	Decorated Storage facilities
Elemental Memory Barriers	EMB	More granular memory barrier support
Embedded.Cache Debug	E.CD	Provides direct access to cache data and directory content
Embedded.Cache Initialization	E.CI	Instructions that invalidate the entire cache
Embedded.Device Control	E.DC	Legacy Device Control bus support
Embedded.Enhanced Debug	E.ED	Embedded Enhanced Debug facility; see Book III-E
Embedded.External PID	E.PD	Embedded External PID facility; see Book III-E
Embedded.Hypervisor Embedded.Hypervisor.LRAT	E.HV E.HV.LRAT	Embedded Logical Partitioning and hypervisor facilities Embedded Hypervisor Logical to Real Address Translation facility; see Book III-E
Embedded.Little-Endian	E.LE	Embedded Little-Endian page attribute; see Book III-E
Embedded.MMU Type FSL	E.MMUF	Type FSL Storage Control
Embedded.Page Table	E.PT	Embedded Page Table facility; see Book III-E
Embedded.TLB Write Conditional	E.TWC	Embedded TLB Write Conditional facility; see Book III-E
Embedded.Performance Monitor	E.PM	Embedded Performance Monitor example; see Book III-E
Embedded.Processor Control	E.PC	Embedded Processor Control facility; see Book III-E
Embedded Cache Locking	ECL	Embedded Cache Locking facility; see Book III-E

Figure 1. Category Listing (Sheet 1 of 2)

Category	Abvr.	Notes
Embedded Multi-Threading Embedded Multi-Threading.Thread Management	EM EM.TM	Embedded Multi-Threading; see Book III-E Embedded Multi-Threading Thread Management Facility
External Control	EC	External Control facility; see Book II
External Proxy	EXP	External Proxy facility; see Book III-E
Floating-Point Floating-Point.Record	FP FP.R	Floating-Point Facilities Floating-Point instructions with Rc=1
Legacy Integer Multiply-Accumulate[1]	LMA	*Legacy Integer Multiply-accumulate* instructions
Legacy Move Assist	LMV	*Determine Left most Zero Byte* instruction
Load/Store Quadword	LSQ	*Load/Store Quadword* instructions; see Book III-S
Memory Coherence	MMC	Requirement for Memory Coherence; see Book II
Move Assist	MA	*Move Assist* instructions
Processor Compatibility	PCR	Processor Compatibility Register
Server.Performance Monitor	S.PM	Server Performance Monitor example; see Book III-S
Server.Processor Control	S.PC	Server Processor Control Facility; see Book III-S
Server.Relaxed Page Table Alignment	S.RPTA	HTAB alignment on 256 KB boundary; see Book III-S
Signal Processing Engine[1, 2] SPE.Embedded Float Scalar Double SPE.Embedded Float Scalar Single SPE.Embedded Float Vector	SP SP.FD SP.FS SP.FV	Facility for signal processing GPR-based Floating-Point double-precision instruction set GPR-based Floating-Point single-precision instruction set GPR-based Floating-Point Vector instruction set
Store Conditional Page Mobility	SCPM	*Store Conditional* accounting for page movement; see Book II
Stream	STM	Stream variant of ***dcbt*** instruction; see Book II
Strong Access Order	SAO	Assist for X86 and Sparc emulation; see Book II
Trace	TRC	Trace Facility; see Book III-S
Transactional Memory	TM	Full hardware Transactional Memory support
Variable Length Encoding	VLE	Variable Length Encoding facility; see Book VLE
Vector-Scalar Extension	VSX	Vector-Scalar Extension Requires implementation of Floating-Point and Vector categories
Vector[1] Vector.Little-Endian Vector.Crypto Vector.RAID	V V.LE V.GBB V.XOR	Vector facilities Little-Endian support for Vector storage operations. Cryptographic instruction set Vector Permute-XOR instruction
Wait	WT	***wait*** instruction; see Book II
64-Bit	64	Required for 64-bit implementations; not defined for 32-bit impl's

[1] Because of overlapping opcode usage, SPE is mutually exclusive with Vector and with Legacy Integer Multiply-Accumulate, and Legacy Integer Multiply-Accumulate is mutually exclusive with Vector.
[2] The SPE-dependent Floating-Point categories are collectively referred to as SPE.Embedded Float_* or SP.*.

Figure 1. Category Listing (Sheet 2 of 2)

An instruction in a category that is not supported by the implementation is treated as an illegal instruction or an unimplemented instruction on that implementation (see Section 1.7.2).

For an instruction that is supported by the implementation with field values that are defined by the architecture, the field values defined as part of a category that is not supported by the implementation are treated as reserved values on that implementation (see Section 1.3.3 and Section 1.8.2).

Bits in a register that are in a category that is not supported by the implementation are treated as reserved.

1.3.5.1 Phased-In/Phased-Out

There are two special categories, *Phased-In* and *Phased-Out*, as well as two additional variations of *Phased-In* as defined below. Abbreviations, if applicable, are shown in parentheses.

Phased-In

These are facilities and instructions that, in the next version of the architecture, will be required as part of the category they are dependent on.

Servers do not implement a facility or instruction in this category. Servers that comply with earlier versions of this architecture may have optionally implemented facilities or instructions that were category Phased-In.

Server, Embedded.Phased-In (S,E.PI)

These are facilities and instructions that are part of the Server environment and, in the next version of the architecture, will be required for the Embedded environment.

It is implementation-dependent whether Embedded processors implement a facility or instruction in this category.

Embedded, Server.Phased-In (E,S.PI)

These are facilities and instructions that are part of the Embedded environment and, in the next version of the architecture, will be required for the Server environment.

Servers do not implement a facility or instruction in this category.

Phased-Out

These are facilities and instructions that, in some future version of the architecture, will be dropped out of the architecture. System developers should develop a migration plan to eliminate use of them in new systems.

For Server platforms, Phased-Out facilities and instructions must be implemented if the facility or instruction is part of another category (including the Base category) that is supported by the Server platform.

Programming Note

Warning: Instructions and facilities being phased out of the architecture are likely to perform poorly on future implementations. New programs should not use them.

Programming Note

Facilities are categorized as Phased-In only in cases where there is a difference between the Server and Embedded environments.

1.3.5.2 Corequisite Category

A corequisite category is an additional category that is associated with an instruction or facility, and must be implemented if the instruction or facility is implemented.

1.3.5.3 Category Notation

Instructions and facilities are considered part of the Base category unless otherwise marked. If a section is marked with a specific category tag, all material in that section and its subsections are considered part of the category, unless otherwise marked. Overview sections may contain discussion of instructions and facilities from various categories without being explicitly marked.

An example of a category tag is: [Category: Server]. Alternatively, a shorthand notation of a category tag includes the category name in angled brackets “<>”, such as <E.HV>.

An example of a dependent category is:
[Category: Server.Phased-In]

The shorthand <E> and <S> may also be used for Category: Embedded and Server respectively.

1.3.6 Environments

All implementations support one of the two defined environments, Server or Embedded. Environments refer to common subsets of instructions that are shared across many implementations. The Server environment describes implementations that support Category: Base and Server. The Embedded environment describes implementations that support Category: Base and Embedded.

1.4 Processor Overview

The basic classes of instructions are as follows:

- branch instructions (Chapter 2)
- GPR-based scalar fixed-point instructions (Chapter 3, Chapter 9, and Chapter 11)
- GPR-based vector fixed-point instructions (Chapter 8)
- GPR-based scalar and vector floating-point instructions (Chapter 10)
- FPR-based scalar floating-point instructions (Chapter 4)
- FPR-based scalar decimal floating-point instructions (Chapter 6)
- VR-based vector fixed-point and floating-point instructions (Chapter 5)
- VSR-based scalar and vector floating-point instructions (Chapter 7)

Scalar fixed-point instructions operate on byte, halfword, word, doubleword, and quadword (see Book III-S) operands, where each operand contained in a GPR. Vector fixed-point instructions operate on vectors of byte, halfword, and word operands, where each vector is contained in a GPR or VR. Scalar floating-point instructions operate on single-precision or double-precision floating-point operands, where each operand is contained in a GPR, FPR, or VSR. Vector floating-point instructions operate on vectors of single-precision and double-precision floating-point operands, where each vector is contained in a GPR, VR, or VSR.

The Power ISA uses instructions that are four bytes long and word-aligned (VLE has different instruction characteristics; see Book VLE). It provides for byte, halfword, word, doubleword, and quadword operand loads and stores between storage and a set of 32 General Purpose Registers (GPRs). It provides for word and doubleword operand loads and stores between storage and a set of 32 Floating-Point Registers (FPRs). It also provides for byte, halfword, word, and quadword operand loads and stores between storage and a set of 32 Vector Registers (VRs). It provides for doubleword and quadword operand loads and stores between storage and a set of 64 Vector-Scalar Registers (VSRs).

Signed integers are represented in two's complement form.

There are no computational instructions that modify storage; instructions that reference storage may reformat the data (e.g. load halfword algebraic). To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location. Figure 2 is a logical representation of instruction processing. Figure 3 shows the registers that are defined in Book I. (A few additional registers that are available to application programs are defined in other Books, and are not shown in the figure.)

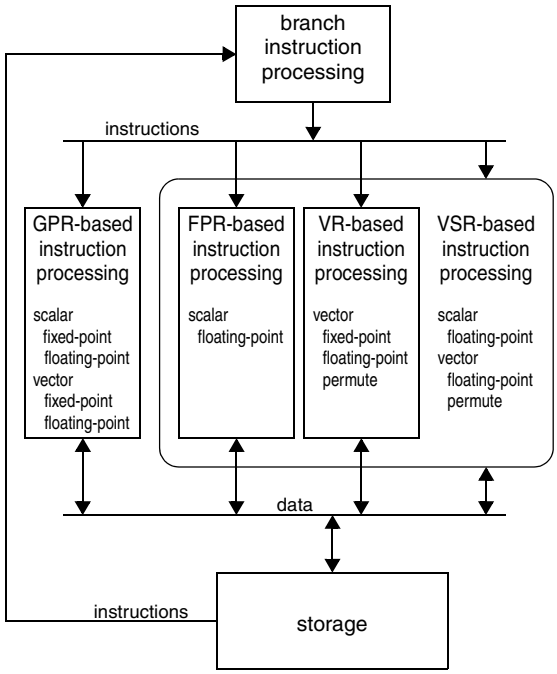
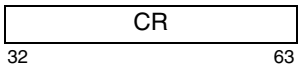
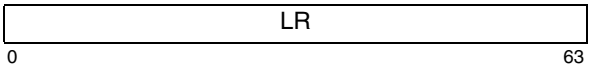


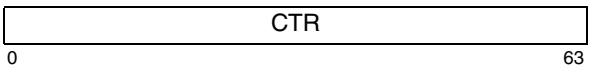
Figure 2. Logical processing model



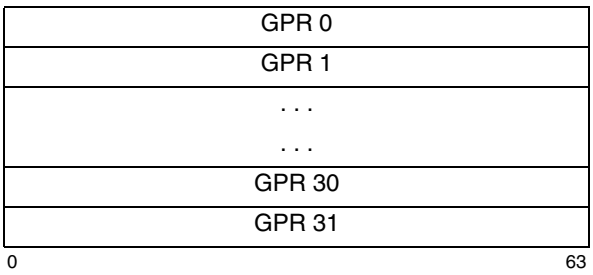
“Condition Register” on page 30



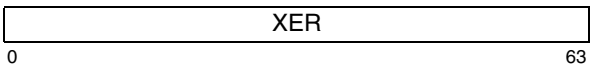
“Link Register” on page 32



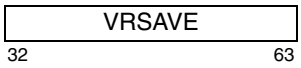
“Count Register” on page 32



“General Purpose Registers” on page 45

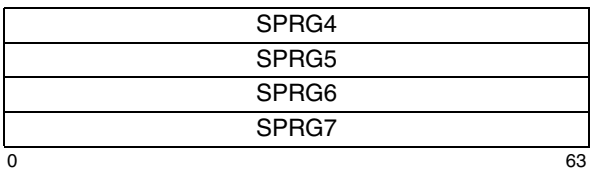


“Fixed-Point Exception Register” on page 45



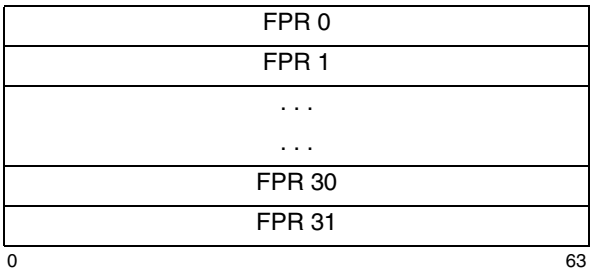
“VR Save Register” on page 173

Category: Embedded:

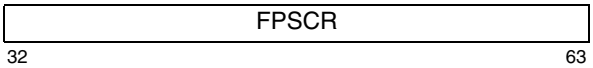


“Software-use SPRs” on page 46.

Category: Floating-Point:

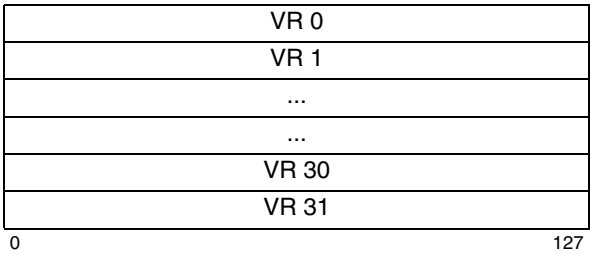


“Floating-Point Registers” on page 116

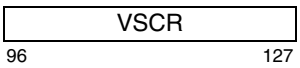


“Floating-Point Status and Control Register” on page 116

Category: Vector:

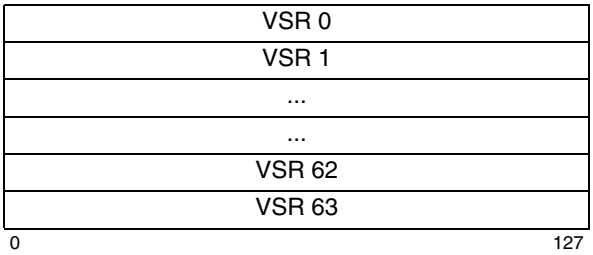


“Vector Registers” on page 172



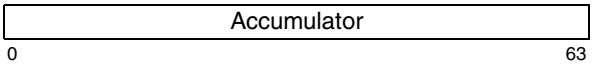
“Vector Status and Control Register” on page 172

Category: Vector-Scalar Extension:

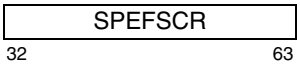


“Vector-Scalar Registers” on page 320

Category: SPE:



“Accumulator” on page 588



“Signal Processing and Embedded Floating-Point Status and Control Register” on page 588

Figure 3. Registers that are defined in Book I

1.5 Computation modes

1.5.1 Modes [Category: Server]

Processors provide two execution modes, 64-bit mode and 32-bit mode. In both of these modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how Condition Register bits and XER bits are set, how the Link Register is set by *Branch* instructions in which LK=1, and how the Count Register is tested by *Branch Conditional* instructions. Nearly all instructions are available in both modes (the only exceptions are a few instructions that are defined in Book III-S). In both modes, effective address computations use all 64 bits of the relevant registers (General Purpose Registers, Link Register, Count Register, etc.) and produce a 64-bit result. However, in 32-bit mode the high-order 32 bits of the computed effective address are ignored for the purpose of addressing storage; see Section 1.10.3 for additional details.

Programming Note

Although instructions that set a 64-bit register affect all 64 bits in both 32-bit and 64-bit modes, operating systems often do not preserve the upper 32-bits of all registers across context switches done in 32-bit mode. For this reason, application programs operating in 32-bit mode should not assume that the upper 32 bits of the GPRs are preserved from instruction to instruction unless the operating system is known to preserve these bits.

1.5.2 Modes [Category: Embedded]

64-bit processors provide 64-bit mode and 32-bit mode. The differences between the two modes are described below. 32-bit processors provide only 32-bit mode, and do so as described at the end of this section.

- In 64-bit mode, the processor behaves as described for 64-bit mode in the Server environment; see Section 1.5.1.
- In 32-bit mode, the processor behavior depends on whether the high-order 32 bits of GPRs are implemented in 32-bit mode, as follows.
  - If these bits are implemented in 32-bit mode, the processor behaves as described for 32-bit mode in the Server environment.
  - If these bits are not implemented in 32-bit mode, the processor behaves as described for 32-bit mode in the Server Environment except for the following.
    - When an effective address is placed in a register other than the Initialize Next Instruction register (see Section 3.7.1 of Book III-E) by an instruction or event, the high-order 32 bits are set to an undefined value (see Section 1.10.3).
    - Except for instructions in the SPE category, instructions that operate on GPRs and SPRs use only the low-order 32 bits of the source GPR or SPR and produce a 32-bit result; the high-order 32 bits of target GPRs are set to an undefined value, and the high-order 32 bits of target SPRs are preserved. The 64-Bit category is not supported.

    Programming Note

    The high-order 32 bits of 64-bit SPRs are not modified in 32-bit mode because for some 64-bit SPRs, such as the Thread Enable Register (see Section 3.3 of Book III-E), these bits control facilities that are active in 32-bit mode. Treating all 64-bit SPRs the same way in this regard simplifies architecture and implementation.

  Implementations may provide a means for selecting between the two treatments of the high-order 32 bits of GPRs in 32-bit mode (i.e., for selecting between the behavior described in the first sub-bullet and the behavior described in the second sub-bullet). The means, if provided, is implementation-specific (including any software synchronization requirements for changing the selection), but must be hypervisor privileged, and the hypervisor must ensure that the selection is constant for a given partition.

32-bit processors provide only 32-bit mode, and provide it as described by the second sub-bullet of the 32-bit mode bullet above.

1.6 Instruction Formats

All instructions are four bytes long and word-aligned (except for VLE instructions; see Book VLE). Thus, whenever instruction addresses are presented to the processor (as in *Branch* instructions) the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address the low-order two bits are zero.

Bits 0:5 always specify the opcode (OPCD, below). Many instructions also have an extended opcode (XO, below). The remaining bits of the instruction contain one or more fields as shown below for the different instruction formats.

The format diagrams given below show horizontally all valid combinations of instruction fields. The diagrams include instruction fields that are used only by instructions defined in Book II or in Book III.

Split Field Notation

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits that are used in permuted order. Such a field is called a *split field*. In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in small letters, once for each of the contiguous sequences. In the RTL description of an instruction having a split field, and in certain other places where individual bits of a split field are identified, the name of the field in small letters represents the concatenation of the sequences from left to right. In all other places, the name of the field is capitalized and represents the concatenation of the sequences in some order, which need not be left to right, as described for each affected instruction.

1.6.1 I-FORM

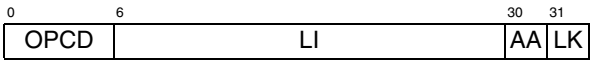


Figure 4. I instruction format

1.6.2 B-FORM

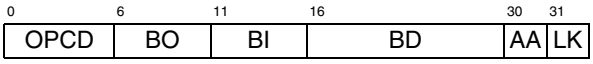


Figure 5. B instruction format

1.6.3 SC-FORM

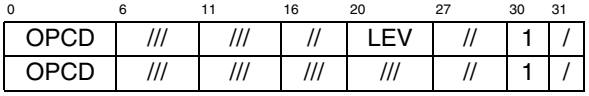


Figure 6. SC instruction format

1.6.4 D-FORM

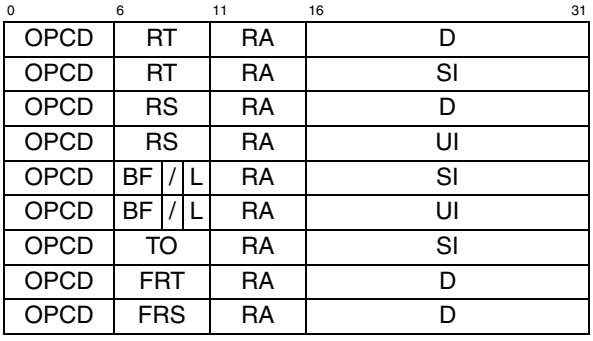


Figure 7. D instruction format

1.6.5 DS-FORM

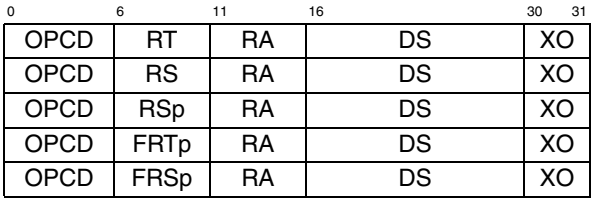


Figure 8. DS instruction format

1.6.6 DQ-FORM

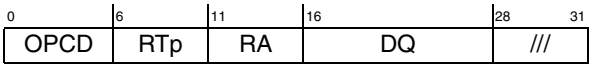


Figure 9. DQ instruction format

1.6.7 X-FORM

0	6	11	16	21	31
OPCD	RT	RA	///	XO	/
OPCD	RT	RA	RB	XO	/
OPCD	RT	RA	RB	XO	EH
OPCD	RT	RA	NB	XO	/
OPCD	RT	/ SR	///	XO	/
OPCD	RT	///	RB	XO	/
OPCD	RT	///	RB	XO	1
OPCD	RT	///	///	XO	/
OPCD	RS	RA	RB	XO	Rc
OPCD	RT	RA	RB	XO	Rc
OPCD	RS	RA	RB	XO	1
OPCD	RS	RA	RB	XO	/
OPCD	RS	RA	NB	XO	/
OPCD	RS	RA	SH	XO	Rc
OPCD	RS	RA	///	XO	Rc
OPCD	RS	RA	///	XO	/
OPCD	RS	/ SR	///	XO	/
OPCD	RS	///	RB	XO	/
OPCD	RS	///	///	XO	/
OPCD	RS	/// L	///	XO	/
OPCD	TH	RA	RB	XO	/
OPCD	BF / L	RA	RB	XO	/
OPCD	BF //	FRA	FRB	XO	/
OPCD	BF //	BFA //	///	XO	/
OPCD	BF //	/// W	U /	XO	Rc
OPCD	BF //	///	///	XO	/
OPCD	TH	RA	RB	XO	/
OPCD	/ CT	///	///	XO	/
OPCD	/ CT	RA	RB	XO	/
OPCD	/// L	RA	RB	XO	/
OPCD	/// L	///	RB	XO	/
OPCD	/// L	///	///	XO	/
OPCD	/// L	/ E	///	XO	/
OPCD	TO	RA	RB	XO	/
OPCD	FRT	RA	RB	XO	/
OPCD	FRT	FRA	FRB	XO	/
OPCD	FRTp	RA	RB	XO	/
OPCD	FRT	///	FRB	XO	Rc
OPCD	FRT	///	FRBp	XO	Rc
OPCD	FRT	///	///	XO	Rc
OPCD	FRTp	///	FRB	XO	Rc
OPCD	FRTp	///	FRBp	XO	Rc
OPCD	FRTp	FRA	FRBp	XO	Rc
OPCD	FRTp	FRAp	FRBp	XO	Rc
OPCD	BF //	FRA	FRBp	XO	/
OPCD	BF //	FRAp	FRBp	XO	/

Figure 10. X Instruction Format

0	6	11	16	21	31
OPCD	FRT	S	FRB	XO	Rc
OPCD	FRTp	S	FRBp	XO	Rc
OPCD	FRS	RA	RB	XO	/
OPCD	FRSp	RA	RB	XO	/
OPCD	BT	///	///	XO	Rc
OPCD	///	RA	RB	XO	/
OPCD	///	///	RB	XO	/
OPCD	///	///	///	XO	/
OPCD	///	///	E ///	XO	/
OPCD	// IH	///	///	XO	/
OPCD	A //	///	///	XO	1
OPCD	A // R	///	///	XO	1
OPCD	///	RA	RB	XO	1
OPCD	/// WC	///	///	XO	/
OPCD	/// T	RA	RB	XO	/
OPCD	VRT	RA	RB	XO	/
OPCD	VRS	RA	RB	XO	/
OPCD	MO	///	///	XO	/

Figure 10. X Instruction Format

1.6.8 XL-FORM

0	6	11	16	21	31
OPCD	BT	BA	BB	XO	/
OPCD	BO	BI	/// BH	XO	LK
OPCD	///		S	XO	/
OPCD	BF //	BFA //	///	XO	/
OPCD	///			XO	/
OPCD	OC			XO	/

Figure 11. XL instruction format

1.6.9 XFX-FORM

0	6	11	21	31
OPCD	RT	spr	XO	/
OPCD	RT	tbr	XO	/
OPCD	RT	0 ///	XO	/
OPCD	RT	1 FXM /	XO	/
OPCD	RT	dcr	XO	/
OPCD	RT	pmrn	XO	/
OPCD	DUI	DUIS	XO	/
OPCD	RS	0 FXM /	XO	/
OPCD	RS	1 FXM /	XO	/
OPCD	RS	spr	XO	/
OPCD	RS	dcr	XO	/
OPCD	RS	pmrn	XO	/

Figure 12. XFX instruction format

1.6.10 XFL-FORM

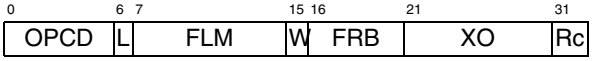


Figure 13. XFL instruction format

1.6.11 XX1-FORM

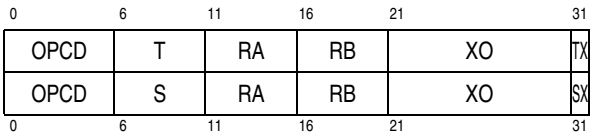


Figure 14. XX1 Instruction Format

1.6.12 XX2-FORM

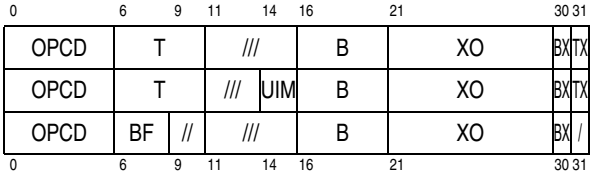


Figure 15. XX2 Instruction Format

1.6.13 XX3-FORM

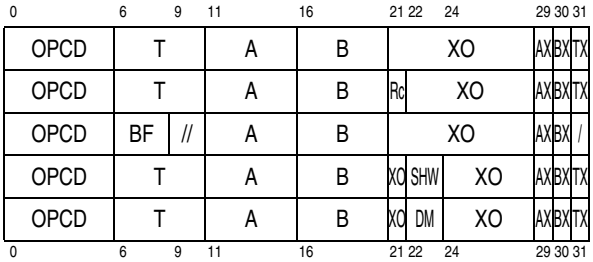


Figure 16. XX3 Instruction Format

1.6.14 XX4-FORM

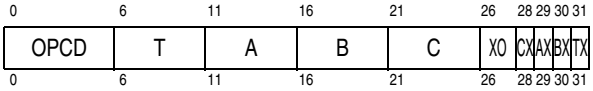


Figure 17. XX4-Form Instruction Format

1.6.15 XS-FORM

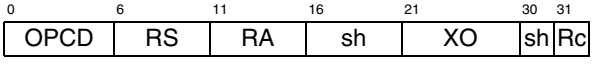


Figure 18. XS instruction format

1.6.16 XO-FORM

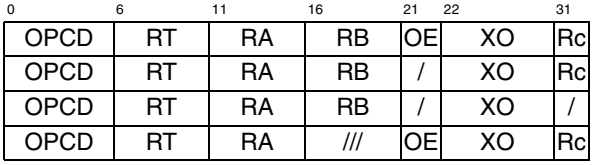


Figure 19. XO instruction format

1.6.17 A-FORM

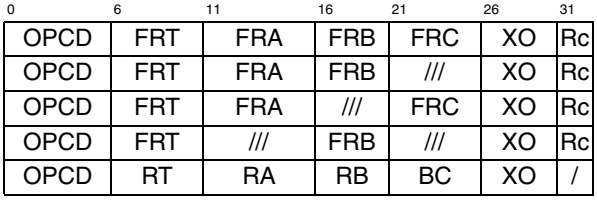


Figure 20. A instruction format

1.6.18 M-FORM

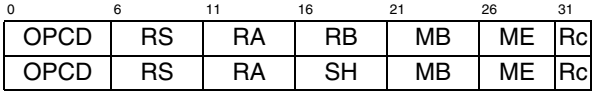


Figure 21. M instruction format

1.6.19 MD-FORM

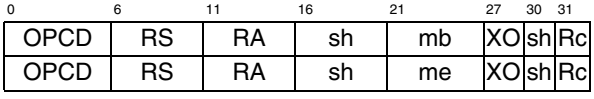


Figure 22. MD instruction format

1.6.20 MDS-FORM

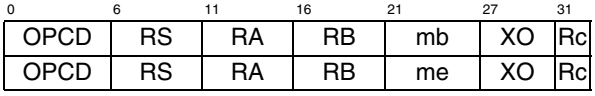


Figure 23. MDS instruction format

1.6.21 VA-FORM

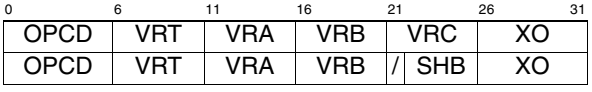


Figure 24. VA instruction format

1.6.22 VC-FORM

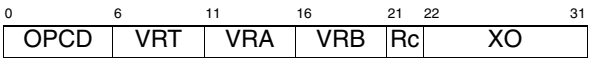


Figure 25. VC instruction format

1.6.23 VX-FORM

0	6	11	16	21	31
OPCD	VRT	VRA	VRB	XO	
OPCD	VRT	///	VRB	XO	
OPCD	VRT	UIM	VRB	XO	
OPCD	VRT	/ UIM	VRB	XO	
OPCD	VRT	// UIM	VRB	XO	
OPCD	VRT	/// UIM	VRB	XO	
OPCD	VRT	SIM	///	XO	
OPCD	VRT	///		XO	
OPCD	///		VRB	XO	

Figure 26. VX instruction format

1.6.24 EVX-FORM

0	6	11	16	21	31
OPCD	RS	RA	RB	XO	
OPCD	RS	RA	UI	XO	
OPCD	RT	///	RB	XO	
OPCD	RT	RA	RB	XO	
OPCD	RT	RA	///	XO	
OPCD	RT	UI	RB	XO	
OPCD	BF //	RA	RB	XO	
OPCD	RT	RA	UI	XO	
OPCD	RT	SI	///	XO	

Figure 27. EVX instruction format

1.6.25 EVS-FORM

0	6	11	16	21	29 31
OPCD	RT	RA	RB	XO	BFA

Figure 28. EVS instruction format

1.6.26 Z22-FORM

0	6	11 15	16	22	31
OPCD	BF //	FRA	DCM	XO	/
OPCD	BF //	FRAp	DCM	XO	/
OPCD	BF //	FRA	DGM	XO	/
OPCD	BF //	FRAp	DGM	XO	/
OPCD	FRT	FRA	SH	XO	Rc
OPCD	FRTp	FRAp	SH	XO	Rc

Figure 29. Z22 instruction format

1.6.27 Z23-FORM

0	6	11	16	21	23	31
OPCD	FRT	TE	FRB	RMC	XO	Rc
OPCD	FRTp	TE	FRBp	RMC	XO	Rc
OPCD	FRT	FRA	FRB	RMC	XO	Rc
OPCD	FRTp	FRA	FRBp	RMC	XO	Rc
OPCD	FRTp	FRAp	FRBp	RMC	XO	Rc
OPCD	FRT	/// R	FRB	RMC	XO	Rc
OPCD	FRTp	/// R	FRBp	RMC	XO	Rc

Figure 30. Z23 instruction format

1.6.28 Instruction Fields

A (6)

Field used by the ***tbegin.*** instruction to specify an implementation-specific function.

Field used by the ***tend.*** instruction to specify the completion of the outer transaction and all nested transactions.

AA (30)

Absolute Address bit.

0 The immediate field represents an address relative to the current instruction address. For I-form branches the effective address of the branch target is the sum of the LI field sign-extended to 64 bits and the address of the branch instruction. For B-form branches the effective address of the branch target is the sum of the BD field sign-extended to 64 bits and the address of the branch instruction.

1 The immediate field represents an absolute address. For I-form branches the effective address of the branch target is the LI field sign-extended to 64 bits. For B-form branches the effective address of the branch target is the BD field sign-extended to 64 bits.

AX (29) & A(11:15)

Fields that are concatenated to specify a VSR to be used as a source.

BA (11:15)

Field used to specify a bit in the CR to be used as a source.

BB (16:20)

Field used to specify a bit in the CR to be used as a source.

BC (21:25)

Field used to specify a bit in the CR to be used as a source.

BD (16:29)
Immediate field used to specify a 14-bit signed two's complement branch displacement which is concatenated on the right with 0b00 and sign-extended to 64 bits.

BF (6:8)
Field used to specify one of the CR fields or one of the FPSCR fields to be used as a target.

BFA (11:13 or 29:31)
Field used to specify one of the CR fields or one of the FPSCR fields to be used as a source.

BH (19:20)
Field used to specify a hint in the *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions. The encoding is described in Section 2.5, "Branch Instructions".

BI (11:15)
Field used to specify a bit in the CR to be tested by a *Branch Conditional* instruction.

BO (6:10)
Field used to specify options for the *Branch Conditional* instructions. The encoding is described in Section 2.5, "Branch Instructions".

BT (6:10)
Field used to specify a bit in the CR or in the FPSCR to be used as a target.

BX (30) & B(16:20)
Fields that are concatenated to specify a VSR to be used as a source.

CT (7:10)
Field used in X-form instructions to specify a cache target (see Section 4.3.2 of Book II).

CX (28) & C(21:25)
Fields that are concatenated to specify a VSR to be used as a source.

D (16:31)
Immediate field used to specify a 16-bit signed two's complement integer which is sign-extended to 64 bits.

DCM (16:21)
Immediate field used as the Data Class Mask.

DCR (11:20)
Field used by the *Move To/From Device Control Register* instructions (see Book III-E).

DGM (16:21)
Immediate field used as the Data Group Mask.

DM (24:25)
Immediate field used by ***xxpermdi*** instruction as doubleword permute control.

DQ (16:27)
Immediate field used to specify a 12-bit signed two's complement integer which is concatenated on the right with 0b0000 and sign-extended to 64 bits.

DS (16:29)
Immediate field used to specify a 14-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

DUI (6:10)
Field used by the ***dnh*** instruction (see Book III-E).

DUIS (11:20)
Field used by the ***dnh*** instruction (see Book III-E).

E (16)
Field used by the *Write MSR External Enable* instruction (see Book III-E).

E (12:15)
Field used to specify the access types ordered by an Elemental Memory Barrier type of ***sync*** instruction.

EH (31)
Field used to specify a hint in the *Load and Reserve* instructions. The meaning is described in Section 4.4.2, "Load and Reserve and Store Conditional Instructions", in Book II.

FLM (7:14)
Field mask used to identify the FPSCR fields that are to be updated by the ***mtfsf*** instruction.

FRA (11:15)
Field used to specify an FPR to be used as a source.

FRAp (11:15)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

FRB (16:20)
Field used to specify an FPR to be used as a source.

FRBp (16:20)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

FRC (21:25)
Field used to specify an FPR to be used as a source.

FRS (6:10)

Field used to specify an FPR to be used as a source.

FRSp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

FRT (6:10)
Field used to specify an FPR to be used as a target.

FRTp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a target.

FXM (12:19)
Field mask used to identify the CR fields that are to be written by the ***mtcrf*** and ***mtocrf*** instructions, or read by the ***mfocrf*** instruction.

IH (8:10)
Field used to specify a hint in the *SLB Invalidate All* instruction. The meaning is described in Section 5.9.3.1, "SLB Management Instructions", in Book III-S.

L (6)
Field used to specify whether the ***mtfsf*** instruction updates the entire FPSCR.

L (10 or 15)
Field used to specify whether a fixed-point Compare instruction is to compare 64-bit numbers or 32-bit numbers.

Field used by the *Data Cache Block Flush* instruction (see Section 4.3.2 of Book II).

Field used by the *Move To Machine State Register* and *TLB Invalidate Entry* instructions (see Book III).

L (9:10)
Field used by the *Data Cache Block Flush* instruction (see Section 4.3.2 of Book II) and also by the *Synchronize* instruction (see Section 4.4.3 of Book II).

LEV (20:26)
Field used by the *System Call* instruction.

LI (6:29)
Immediate field used to specify a 24-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

LK (31)
LINK bit.
- 0 Do not set the Link Register.
- 1 Set the Link Register. The address of the instruction following the *Branch* instruction is placed into the Link Register.

MB (21:25) and ME (26:30)
Fields used in M-form instructions to specify a 64-bit mask consisting of 1-bits from bit MB+32 through bit ME+32 inclusive and 0-bits elsewhere, as described in Section 3.3.14, "Fixed-Point Rotate and Shift Instructions" on page 91.

MB (21:26)
Field used in MD-form and MDS-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, "Fixed-Point Rotate and Shift Instructions" on page 91.

ME (21:26)
Field used in MD-form and MDS-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, "Fixed-Point Rotate and Shift Instructions" on page 91.

MO (6:10)
Field used in X-form instructions to specify a subset of storage accesses.

NB (16:20)
Field used to specify the number of bytes to move in an immediate Move Assist instruction.

OC (6:20)
Field used by the Embedded Hypervisor Privilege instruction.

OPCD (0:5)
Primary opcode field.

OE (21)
Field used by XO-form instructions to enable setting OV and SO in the XER.

PMRN (11:20)
Field used to specify a Performance Monitor Register for the ***mfpmr*** and ***mtpmr*** instructions.

R (10)

Field used by the ***tbegin.*** instruction to specify the start of a ROT.

R (15)
Immediate field that specifies whether the RMC is specifying the primary or secondary encoding

RA (11:15)
Field used to specify a GPR to be used as a source or as a target.

RB (16:20)
Field used to specify a GPR to be used as a source.

Rc (21 OR 31)
RECORD bit.
- 0 Do not alter the Condition Register.

1 Set Condition Register Field 0, Field 1, or Field 6 as described in Section 2.3.1, “Condition Register” on page 30.

RMC (21:22)
Immediate field used for DFP rounding mode control.

RS (6:10)
Field used to specify a GPR to be used as a source.

RSp (6:10)
Field used to specify an even/odd pair of GPRs to be concatenated and used as a source.

RT (6:10)
Field used to specify a GPR to be used as a target.

RTp (6:10)
Field used to specify an even/odd pair of GPRs to be concatenated and used as a target.

S (11 or 20)
Immediate field that specifies signed versus unsigned conversion.
Immediate field that specifies whether or not the ***rfebb*** instruction re-enables event-based branches.

SH (16:20, or 16:20 and 30, or 16:21)
Field used to specify a shift amount.

SHB (22:25)
Field used to specify a shift amount in bytes.

SHW (24:25)
Field used to specify a shift amount in words.

SI (16:31 or 11:15)
Immediate field used to specify a 16-bit signed integer.

SIM (11:15)
Immediate field used to specify a 5-bit signed integer.

SP (11:12)
Immediate field that specifies signed versus unsigned conversion.

SPR (11:20)
Field used to specify a Special Purpose Register for the ***mtspr*** and ***mfspr*** instructions.

SR (12:15)
Field used by the *Segment Register Manipulation* instructions (see Book III-S).

SX (31) & S(6:10)
Fields that are concatenated to specify a VSR to be used as a source.

T(9:10)
Field used to specify the type of invalidation done by a *TLB Invalidate Local* instruction (see Book III-E).

TBR (11:20)
Field used by the *Move From Time Base* instruction (see Section 6.2.1 of Book II).

TE (11:15)
Immediate field that specifies a DFP exponent.

TH (6:10)
Field used by the data stream variant of the ***dcbt*** and ***dcbtst*** instructions (see Section 4.3.2 of Book II).

TO (6:10)
Field used to specify the conditions on which to trap. The encoding is described in Section 3.3.11, “Fixed-Point Trap Instructions” on page 80.

TX (31) & T (6:10)
Fields that are concatenated to specify a VSR to be used as a target.

U (16:19)
Immediate field used as the data to be placed into a field in the FPSCR.

UI (11:15, 16:20, or 16:31)
Immediate field used to specify an unsigned integer.

UIM (11:15, 12:15, 13:15, 14:15)
Immediate field used to specify an unsigned integer.

VRA (11:15)
Field used to specify a VR to be used as a source.

VRB (16:20)
Field used to specify a VR to be used as a source.

VRC (21:25)
Field used to specify a VR to be used as a source.

VRS (6:10)
Field used to specify a VR to be used as a source.

VRT (6:10)
Field used to specify a VR to be used as a target.

W (15)

Field used by the ***mtfsfi*** and ***mtfsf*** instructions to specify the target word in the FPSCR.

WC (9:10)
Field used to specify the condition or conditions that cause instruction execution to resume after executing a ***wait*** [Category: Wait] instruction (see Section 4.4.4 of Book II).

XO (21, 21:28, 21:29, 21:30, 21:31, 22:28, 22:30, 22:31, 23:30, 24:28, 26:27, 26:30, 26:31, 27:29, 27:30, or 30:31)
Extended opcode field.

1.7 Classes of Instructions

An instruction falls into exactly one of the following three classes:

Defined
Illegal
Reserved

The class is determined by examining the opcode, and the extended opcode if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or a reserved instruction, the instruction is illegal.

1.7.1 Defined Instruction Class

This class of instructions contains all the instructions defined in this document.

A defined instruction can have preferred and/or invalid forms, as described in Section 1.8.1, “Preferred Instruction Forms” and Section 1.8.2, “Invalid Instruction Forms”. Instructions that are part of a category that is not supported are treated as illegal instructions.

1.7.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in Appendix D of Book Appendices. Illegal instructions are available for future extensions of the Power ISA ; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction will cause the system illegal instruction error handler to be invoked and will have no other effect.

An instruction consisting entirely of binary 0s is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage will result in the invocation of the system illegal instruction error handler.

1.7.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix E of Book Appendices.

Reserved instructions are allocated to specific purposes that are outside the scope of the Power ISA.

Any attempt to execute a reserved instruction will:

- perform the actions described by the implementation if the instruction is implemented; or
- cause the system illegal instruction error handler to be invoked if the instruction is not implemented.

1.8 Forms of Defined Instructions

1.8.1 Preferred Instruction Forms

Some of the defined instructions have preferred forms. For such an instruction, the preferred form will execute in an efficient manner, but any other form may take significantly longer to execute than the preferred form.

Instructions having preferred forms are:

- the *Condition Register Logical* instructions
- the *Load/Store Multiple* instructions
- the *Load/Store String* instructions
- the *Or Immediate* instruction (preferred form of no-op)
- the *Move To Condition Register Fields* instruction

1.8.2 Invalid Instruction Forms

Some of the defined instructions can be coded in a form that is invalid. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

In general, any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked or yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some instruction forms are invalid because the instruction contains a reserved value in a defined field (see Section 1.3.3 on page 5); these invalid forms are not discussed further. All other invalid forms are identified in the instruction descriptions.

References to instructions elsewhere in this document assume the instruction form is not invalid, unless otherwise stated or obvious from context.

Assembler Note

Assemblers should report uses of invalid instruction forms as errors.

1.8.3 Reserved-no-op Instructions [Category: Phased-In]

Reserved-no-op instructions include the following extended opcodes under primary opcode 31: 530, 562, 594, 626, 658, 690, 722, and 754.

Reserved-no-op instructions are provided in the architecture to anticipate the eventual adoption of performance hint instructions to the architecture. For these instructions, which cause no visible change to architected state, employing a reserved-no-op opcode will allow software to use this new capability on new implementations that support it while remaining compatible with existing implementations that may not support the new function.

When a reserved-no-op instruction is executed, no operation is performed.

Reserved-no-op instructions are not assigned instruction names or mnemonics. There are no individual descriptions of reserved-no-op instructions in this document.

1.9 Exceptions

There are two kinds of exception, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction include the following:

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a “privileged” instruction (see Book III) (system illegal instruction error handler or system privileged instruction error handler)
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler)
- an attempt to execute an instruction that is not provided by the implementation (system illegal instruction error handler)
- an attempt to access a storage location that is unavailable (system instruction storage error handler or system data storage error handler)
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler)
- the execution of a *System Call* instruction (system service program)
- the execution of a *Trap* instruction that traps (system trap handler)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (system floating-point enabled exception error handler)
- the execution of an auxiliary processor instruction that causes an auxiliary processor enabled exception to exist (system auxiliary processor enabled exception error handler)

The exceptions that can be caused by an asynchronous event are described in Book III.

The invocation of the system error handler is precise, except that the invocation of the auxiliary processor enabled exception error handler may be imprecise, and if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 125), then the invocation of the system floating-point enabled exception error handler may also be imprecise. When the system error handler is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the system error handler, has not yet occurred).

Additional information about exception handling can be found in Book III.

1.10 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II and Book III), or when it fetches the next sequential instruction.

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

The byte ordering (Big-Endian or Little-Endian) for a storage access is specified by the operating system. In the Embedded environment this ordering is a page attribute (see Book II) and is specified independently for each virtual page, while in the Server environment it is specified by the two mode bits, LE and SLE (see Book III-S), and applies to all storage.

1.10.1 Storage Operands

A storage operand may be a byte, a halfword, a word, a doubleword, or a quadword, or, for the *Load/Store Multiple* and *Move Assist* instructions, a sequence of bytes (*Move Assist*) or words (*Load/Store Multiple*). For example, the storage operand of a *Load Floating-Point Double Pair* instruction is a quadword, not two doublewords. The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte). An instruction for which the storage operand is a byte is said to cause a byte access, and similarly for halfword, word, doubleword, and quadword.

Operand length is implicit for each instruction.

The storage operand of a single-register *Storage Access* instruction or quadword *Load* or *Store* instruction has a "natural" alignment boundary equal to the operand length. In other words, the "natural" address of such an operand is an integral multiple of the operand length. Such an operand is said to be *aligned* if it is aligned at its natural boundary; otherwise it is said to be *unaligned.* See the following table.

Operand	Length	$Addr_{60:63}$ if aligned
Byte	8 bits	xxxx
Halfword	2 bytes	xxx0
Word	4 bytes	xx00
Doubleword	8 bytes	x000
Quadword	16 bytes	0000

Note: An "x" in an address bit position indicates that the bit can be 0 or 1 independent of the contents of other bits in the address.

The concept of alignment is also applied more generally, to any datum in storage. For example, a 12-byte datum in storage is said to be word-aligned if its address is an integral multiple of 4.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. For single-register *Storage Access* instructions and quadword *Load* and *Store* instructions, the best performance is obtained when storage operands are aligned. Additional effects of data placement on performance are described in Chapter 2 of Book II.

When a storage operand of length N bytes starting at effective address EA is copied between storage and a register that is R bytes long (i.e., the register contains bytes numbered from 0, most significant, through R-1, least significant), the bytes of the operand are placed into the register or into storage in a manner that depends on the byte ordering for the storage access as shown in Figure 31, unless otherwise specified in the instruction description.

Big-Endian Byte Ordering	
Load	Store
for i=0 to N-1: $RT_{(R-N)+i} \leftarrow MEM(EA+i,1)$	for i=0 to N-1: $MEM(EA+i,1) \leftarrow (RS)_{(R-N)+i}$
Little-Endian Byte Ordering	
Load	Store
for i=0 to N-1: $RT_{(R-1)-i} \leftarrow MEM(EA+i,1)$	for i=0 to N-1: $MEM(EA+i,1) \leftarrow (RS)_{(R-1)-i}$

Notes:
1. In this table, subscripts refer to bytes in a register rather than to bits as defined in Section 1.3.2.
2. This table does not apply to the ***lvebx***, ***lvehx***, ***lvewx, stvebx***, ***stvehx***, and ***stvewx*** instructions.

Figure 31. Storage operands and byte ordering

Figure 32 shows an example of a C language structure **s** containing an assortment of scalars and one character string. The value assumed to be in each structure element is shown in hex in the C comments; these values are used below to show how the bytes making up each structure element are mapped into storage. It is assumed that structure s is compiled for

32-bit mode or for a 32-bit implementation. (This affects the length of the pointer to c.)

C structure mapping rules permit the use of padding (skipped bytes) in order to align the scalars on desirable boundaries. Figures 33 and 34 show each scalar aligned at its natural boundary. This alignment introduces padding of four bytes between **a** and **b**, one byte between **d** and **e**, and two bytes between **e** and **f**. The same amount of padding is present for both Big-Endian and Little-Endian mappings.

The Big-Endian mapping of structure **s** is shown in Figure 33. Addresses are shown in hex at the left of each doubleword, and in small figures below each byte. The contents of each byte, as indicated in the C example in Figure 32, are shown in hex (as characters for the elements of the string).

The Little-Endian mapping of structure s is shown in Figure 34. Doublewords are shown laid out from right to left, which is the common way of showing storage maps for processors that implement only Little-Endian byte ordering.

```
struct {
    int      a;      /* 0x1112_1314                  word            */
    double   b;      /* 0x2122_2324_2526_2728        doubleword      */
    char *   c;      /* 0x3132_3334                  word            */
    char     d[7];   /* 'A', 'B', 'C', 'D', 'E', 'F', 'G'  array of bytes  */
    short    e;      /* 0x5152                       halfword        */
    int      f;      /* 0x6162_6364                  word            */
} s;
```

Figure 32. C structure 's', showing values of elements

00	11	12	13	14				
	00	01	02	03	04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	31	32	33	34	'A'	'B'	'C'	'D'
	10	11	12	13	14	15	16	17
18	'E'	'F'	'G'		51	52		
	18	19	1A	1B	1C	1D	1E	1F
20	61	62	63	64				
	20	21	22	23				

Figure 33. Big-Endian mapping of structure 's'

				11	12	13	14	00
07	06	05	04	03	02	01	00	
21	22	23	24	25	26	27	28	08
0F	0E	0D	0C	0B	0A	09	08	
'D'	'C'	'B'	'A'	31	32	33	34	10
17	16	15	14	13	12	11	10	
		51	52		'G'	'F'	'E'	18
1F	1E	1D	1C	1B	1A	19	18	
				61	62	63	64	20
				23	22	21	20	

Figure 34. Little-Endian mapping of structure 's'

1.10.2 Instruction Fetches

Instructions are always four bytes long and word-aligned (except for VLE instructions; see Book VLE).

When an instruction starting at effective address EA is fetched from storage, the relative order of the bytes within the instruction depend on the byte ordering for the storage access as shown in Figure 35.

Big-Endian Byte Ordering
for i=0 to 3: $inst_i \leftarrow$ MEM(EA+i,1)
Little-Endian Byte Ordering
for i=0 to 3: $inst_{3-i} \leftarrow$ MEM(EA+i,1)
Note: In this table, subscripts refer to bytes of the instruction rather than to bits as defined in Section 1.3.2.

Figure 35. Instructions and byte ordering

Figure 36 shows an example of a small assembly language program **p**.

```
loop:
        cmplwi      r5,0
        beq         done
        lwzux       r4,r5,r6
        add         r7,r7,r4
        subi        r5,r5,4
        b           loop

done:
        stw         r7,total
```

Figure 36. Assembly language program 'p'

The Big-Endian mapping of program **p** is shown in Figure 37 (assuming the program starts at address 0).

00	loop: cmplwi r5,0				beq done			
	00	01	02	03	04	05	06	07
08	lwzux r4,r5,r6				add r7,r7,r4			
	08	09	0A	0B	0C	0D	0E	0F
10	subi r5,r5,4				b loop			
	10	11	12	13	14	15	16	17
18	done: stw r7,total							
	18	19	1A	1B	1C	1D	1E	1F

Figure 37. Big-Endian mapping of program 'p'

The Little-Endian mapping of program **p** is shown in Figure 38.

	beq done				loop: cmplwi r5,0			00
	07	06	05	04	03	02	01	00
	add r7,r7,r4				lwzux r4,r5,r6			08
	0F	0E	0D	0C	0B	0A	09	08
	b loop				subi r5,r5,4			10
	17	16	15	14	13	12	11	10
					done: stw r7,total			18
	1F	1E	1D	1C	1B	1A	19	18

Figure 38. Little-Endian mapping of program 'p'

Programming Note

The terms *Big-Endian* and *Little-Endian* come from Part I, Chapter 4, of Jonathan Swift's *Gulliver's Travels*. Here is the complete passage, from the edition printed in 1734 by George Faulkner in Dublin.

> ... our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of *Lilliput* and *Blefuscu*. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of *Blefuscu*; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the *Big-Endians* have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of *Blefuscu* did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet *Lustrog*, in the fifty-fourth Chapter of the *Brundrecal*, (which is their *Alcoran*.) This, however, is thought to be a mere Strain upon the text: For the Words are these; *That all true Believers shall break their Eggs at the convenient End*: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the Power of the chief Magistrate to determine. Now the *Big-Endian* Exiles have found so much Credit in the Emperor of *Blefuscu's* Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

1.10.3 Effective Address Calculation

An effective address is computed by the processor when executing a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II, Book III, and Book VLE) when fetching the next sequential instruction, or when invoking a system error handler. The following provides an overview of this process. More detail is provided in the individual instruction descriptions.

Effective address calculations, for both data and instruction accesses, use 64-bit two's complement addition. All 64 bits of each address component participate in the calculation regardless of mode (32-bit or 64-bit). In this computation one operand is an address (which is by definition an unsigned number) and the second is a signed offset. Carries out of the most significant bit are ignored.

In 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, 2^{64} - 1, to address 0, except that if the current instruction is at effective address 2^{64} - 4 the effective address of the next sequential instruction is undefined.

In 32-bit mode, the low-order 32 bits of the 64-bit result, preceded by 32 0 bits, comprise the 64-bit effective address for the purpose of addressing storage. When an effective address is placed into a register by an instruction or event, the value placed into the high-order 32 bits of the register differs between the Server environment and the Embedded environment.

- Server environment, and Embedded Environment when the high-order 32 bits of GPRs are implemented:
  - Load with Update and Store with Update instructions set the high-order 32 bits of register RA to the high-order 32 bits of the 64-bit result.

- In all other cases (e.g., the Link Register when set by Branch instructions having LK=1, Special Purpose Registers when set to an effective address by invocation of a system error handler) the high-order 32 bits of the register are set to 0s except as described in the last sentence of this paragraph.

- Embedded environment when the high-order 32 bits of GPRs are not implemented for the following cases:
  The high-order 32 bits of the register are set to an undefined value except for the Initialize Next Instruction register [Category: Embedded.Multithreading] (see Section 1.5.2 and Book III), and for the following case. For a register that is loaded with an effective address by the invocation of a system error handler, the high-order 32 bits of the register are set to 0s if the computation mode is 64-bit after the system error is invoked. The 64-bit current instruction address is not affected by a change from 32-bit mode to 64-bit mode, but is affected by a change from 64-bit mode to 32-bit mode. In the latter case, the high-order 32 bits are set to 0. The same is true for the 64-bit next instruction address, except as described in the last item of the list below.

As used to address storage, the effective address arithmetic appears to wrap around from the maximum address, $2^{32} - 1$, to address 0, except that if the current instruction is at effective address $2^{32} - 4$ the effective address of the next sequential instruction is undefined.

RA is a field in the instruction which specifies an address component in the computation of an effective address. A zero in the RA field indicates the absence of the corresponding address component. A value of zero is substituted for the absent component of the effective address computation. This substitution is shown in the instruction descriptions as (RA|0).

Effective addresses are computed as follows. In the descriptions below, it should be understood that “the contents of a GPR” refers to the entire 64-bit contents, independent of mode, but that in 32-bit mode only bits 32:63 of the 64-bit result of the computation are used to address storage.

- With X-form instructions, in computing the effective address of a data element, the contents of the GPR designated by RB (or the value zero for ***lswi*** and ***stswi***) are added to the contents of the GPR designated by RA or to zero if RA=0 or RA is not used in forming the EA.

- With D-form instructions, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.

- With DS-form instructions, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.

- With I-form Branch instructions, the 24-bit LI field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.

- With B-form Branch instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.

- With XL-form Branch instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the target instruction.

- With sequential instruction fetching, the value 4 is added to the address of the current instruction to form the effective address of the next instruction, except that if the current instruction is at the maximum instruction effective address for the mode ($2^{64} - 4$ in 64-bit mode, $2^{32} - 4$ in 32-bit mode) the effective address of the next sequential instruction is undefined. (There is one other exception to this rule; this exception involves changing between 32-bit mode and 64-bit mode and is described in Section 6.3.2 of Book III-E and Section 6.3.2 of Book III-E.)

If the size of the operand of a storage access instruction is more than one byte, the effective address for each byte after the first is computed by adding 1 to the effective address of the preceding byte.

Chapter 2. Branch Facility

2.1 Branch Facility Overview

This chapter describes the registers and instructions that make up the Branch Facility.

2.2 Instruction Execution Order

In general, instructions appear to execute sequentially, in the order in which they appear in storage. The exceptions to this rule are listed below.

- *Branch* instructions for which the branch is taken cause execution to continue at the target address specified by the Branch instruction.
- *Trap* instructions for which the trap conditions are satisfied, and *System Call* instructions, cause the appropriate system handler to be invoked.
- Transaction failure will eventually cause the transaction's failure handler, implied by the ***tbegin.*** instruction, to be invoked. See the programming note following the ***tbegin.*** description in Section 5.5 of Book II.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.9, "Exceptions" on page 22.
- Returning from a system service program, system trap handler, or system error handler causes execution to continue at a specified address.

The model of program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction is called the "sequential execution model". In general, the processor obeys the sequential execution model. For the instructions and facilities defined in this Book, the only exceptions to this rule are the following.

- A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 4.4). The instruction that causes the exception need not complete before the next instruction begins execution, with respect to setting exception bits and (if the exception is enabled) invoking the system error handler.
- A Store instruction modifies one or more bytes in an area of storage that contains instructions that will subsequently be executed. Before an instruction in that area of storage is executed, software synchronization is required to ensure that the instructions executed are consistent with the results produced by the Store instruction.

Programming Note

This software synchronization will generally be provided by system library programs (see Section 1.9 of Book II). Application programs should call the appropriate system library program before attempting to execute modified instructions.

2.3 Branch Facility Registers

2.3.1 Condition Register

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching).

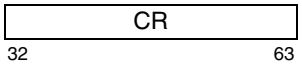


Figure 39. Condition Register

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways.

- Specified fields of the CR can be set by a move to the CR from a GPR (***mtcrf***, ***mtocrf***).
- A specified field of the CR can be set by a move to the CR from another CR field (***mcrf***), from $XER_{32:35}$ (***mcrxr***), or from the FPSCR (***mcrfs***).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 1 can be set as the implicit result of a decimal floating-point instruction.
- CR Field 6 can be set as the implicit result of a vector instruction.
- A specified CR field can be set as the result of a *Compare* instruction or of a ***tcheck*** instruction (see Book II).

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

For all fixed-point instructions in which Rc=1, and for ***addic.***, ***andi.***, and ***andis.***, the first three bits of CR Field 0 (bits 32:34 of the Condition Register) are set by signed comparison of the result to zero, and the fourth bit of CR Field 0 (bit 35 of the Condition Register) is copied from the SO field of the XER. “Result” here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the 64-bit value placed into the target register in 32-bit mode.

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if      (target_register)M:63 < 0 then c ← 0b100
else if (target_register)M:63 > 0 then c ← 0b010
else                                   c ← 0b001
CR0 ← c || XERSO
```

If any portion of the result is undefined, then the value placed into the first three bits of CR Field 0 is undefined.

The bits of CR Field 0 are interpreted as follows.

Bit	Description
0	***Negative*** (LT) The result is negative.
1	***Positive*** (GT) The result is positive.
2	***Zero*** (EQ) The result is zero.
3	***Summary Overflow*** (SO) This is a copy of the contents of XER_{SO} at the completion of the instruction.

With the exception of ***tcheck***, the Transactional Memory instructions set $CR0_{0:2}$ indicating the state of the facility prior to instruction execution, or transaction failure. A complete description of the meaning of these bits is given in the instruction descriptions in Section 5.5 of Book II. These bits are interpreted as follows:

CR0	Description
000 || 0	Transaction state of Non-transactional prior to instruction
010 || 0	Transaction state of Transactional prior to instruction
001 || 0	Transaction state of Suspended prior to instruction
101 || 0	Transaction failure

The ***tcheck*** instruction similarly sets bits 1 and 2 of CR field BF to indicate the transaction state, and additionally sets bit 0 to TDOOMED, as defined in Section 5.5 of Book II.

CR field BF	Description
TDOOMED || 00 || 0	Transaction state of Non-transactional prior to instruction
TDOOMED || 10 || 0	Transaction state of Transactional prior to instruction
TDOOMED || 01 || 0	Transaction state of Suspended prior to instruction

Programming Note

Setting of bit 3 of the specified CR field to zero by ***tcheck*** and of field $CR0_3$ to zero by other TM instructions is intended to preserve these bits for future function. Software should not depend on the bits being zero.

The ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx***., and ***stqcx.*** instructions (see Section 4.4.2, “Load and Reserve and Store

Conditional Instructions", in Book II) also set CR Field 0.

For all floating-point instructions in which Rc=1, CR Field 1 (bits 36:39 of the Condition Register) is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register. This occurs regardless of whether any exceptions are enabled, and regardless of whether the writing of the result is suppressed (see Section 4.4, "Floating-Point Exceptions" on page 124). These bits are interpreted as follows.

Bit	Description
32	***Floating-Point Exception Summary*** (FX) This is a copy of the contents of $FPSCR_{FX}$ at the completion of the instruction.
33	***Floating-Point Enabled Exception Summary*** (FEX) This is a copy of the contents of $FPSCR_{FEX}$ at the completion of the instruction.
34	***Floating-Point Invalid Operation Exception Summary*** (VX) This is a copy of the contents of $FPSCR_{VX}$ at the completion of the instruction.
35	***Floating-Point Overflow Exception*** (OX) This is a copy of the contents of $FPSCR_{OX}$ at the completion of the instruction.

For *Compare* instructions, a specified CR field is set to reflect the result of the comparison. The bits of the specified CR field are interpreted as follows. A complete description of how the bits are set is given in the instruction descriptions in Section 3.3.10, "Fixed-Point Compare Instructions" on page 78, Section 4.6.8, "Floating-Point Compare Instructions" on page 159, and Section 8.3.9, "SPE Instruction Set" on page 594.

Bit	Description
0	***Less Than, Floating-Point Less Than*** (LT, FL) For fixed-point Compare instructions, (RA) < SI or (RB) (signed comparison) or (RA) $<^u$ UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) < (FRB).
1	***Greater Than, Floating-Point Greater Than*** (GT, FG) For fixed-point Compare instructions, (RA) > SI or (RB) (signed comparison) or (RA) $>^u$ UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) > (FRB).
2	***Equal, Floating-Point Equal*** (EQ, FE) For fixed-point Compare instructions, (RA) = SI, UI, or (RB). For floating-point Compare instructions, (FRA) = (FRB).
3	***Summary Overflow, Floating-Point Unordered*** (SO,FU) For fixed-point *Compare* instructions, this is a copy of the contents of XER_{SO} at the completion of the instruction. For floating-point *Compare* instructions, one or both of (FRA) and (FRB) is a NaN.

The *Vector Integer Compare* instructions (see Section 5.9.2, "Vector Integer Compare Instructions") compare two Vector Registers element by element, interpreting the elements as unsigned or signed integers depending on the instruction, and set the corresponding element of the target Vector Register to all 1s if the relation being tested is true and 0s if the relation being tested is false.

If Rc=1, CR Field 6 is set to reflect the result of the comparison, as follows

Bit	Description
0	The relation is true for all element pairs (i.e., VRT is set to all 1s).
1	0
2	The relation is false for all element pairs (i.e., VRT is set to all 0s).
3	0

The *Vector Floating-Point Compare* instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the *Vector Compare Bounds Floating-Point* instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the *Vector Integer Compare* instructions.

Bit	Description
0	The relation is true for all element pairs (i.e., VRT is set to all 1s).
1	0
2	The relation is false for all element pairs (i.e., VRT is set to all 0s).
3	0

The *Vector Compare Bounds Floating-Point* instruction on page 251 sets CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be "within the bounds" specified by a single-precision floating-point value y if $-y \le x \le y$.

Bit	Description
0	0
1	0

2 Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.

3 0

2.3.2 Link Register

The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the *Branch Conditional to Link Register* instruction, and it holds the return address after Branch instructions for which LK=1.

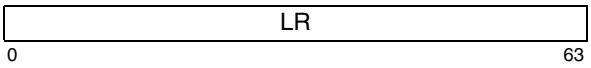


Figure 40. Link Register

2.3.3 Count Register

The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the *Branch Conditional to Count Registe*r instruction.

Figure 41. Count Register

2.3.4 Target Address Register

The Target Address Register (TAR) is a 64-bit register. It can be used to provide bits 0:61 of the branch target address for the *Branch Conditional to Branch Target Address Registe*r instruction. Bits 62:63 are ignored by the hardware but can be set and reset by software.

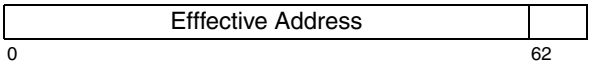


Figure 42. Target Address Register

Programming Note

The TAR is reserved for system software.

2.4 Branch History Rolling Buffer [Category: Server]

The Branch History Rolling Buffer (BHRB) is a buffer containing an implementation-dependent number of entries, referred to as BHRB Entries (BHRBEs), that contain information related to branches that have been taken. Entries are numbered from 0 through n, where n is implementation dependent. Entry 0 is the most-recently written entry.

The system program typically controls the availability of the BHRB as well as the number of entries that it contains. If the BHRB is accessed when it is unavailable, the Facility Unavailable error handler is invoked.

Various events or actions by the system program may result in the BHRB occasionally being cleared. If BHRB entries are read after this has occurred, 0s will be returned. See the description of the ***mfbhrbe*** instruction for additional information.

The BHRB is typically used in conjunction with Performance Monitor event-based branches. (See Chapter 7 of Book II.) When used in conjunction with this facility, $BESCR_{PME}$ is set to 1 to enable Performance Monitor event-based branches, and Performance Monitor alerts are enabled to enable the writing of BHRB entries. (See Section 9.4.4 of Book III-S.) When a Performance Monitor alert occurs, Performance Monitor alerts are disabled, BHRB entries are no longer written, and an event-based branch occurs. (See Chapter 9 of Book III-S for additional information.) The event-based branch handler can then access the contents of the BHRB for analysis.

When the BHRB is written by hardware, only those *Branch* instructions that meet the filtering criterion are written. The filtering criterion is set by the system program. (See Section 9.4.7 of Book III-S.)

The following paragraphs describe the entries written into the BHRB for various types of *Branch* instructions for which the branch was taken. In some circumstances, however, the hardware may be unable to make the entry even though the following paragraphs require it. In such cases, the hardware sets the EA field to 0, and indicates any missed entries using the T and P fields. (See Section 2.4.1.)

When an I-form or B-form *Branch* instruction is entered into the BHRB, bits 0:61 of the effective address of the *Branch* instruction are placed into the next available entry, except that the entry may or may not be written if the instruction following the *Branch* instruction is not another *Branch* instruction, and target address of the *Branch* instruction does not exceed its effective address by more than 8.

When a XL-form *Branch* instruction is entered into the BHRB, bits 0:61 of the branch instruction are written into the next available entry if allowed by the filtering mode; subsequently, bits 0:61 of the effective address of the Branch target is written into the following entry. The BHRB is read by means of the ***mfbhrbe*** instruction.

BHRB entries are written as described above without regard to transactional state and are not removed due to transaction failures.

2.4.1 Branch History Rolling Buffer Entry Format

Branch History Rolling Buffer Data Entries (BHRBEs) have the following format.

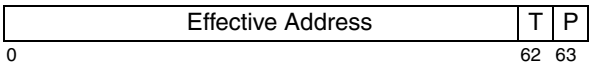


Figure 43. Branch History Rolling Buffer Data

0:61 **Effective Address (EA)**
When this field is set to a non-zero value, it contains bits 0:61 of the effective address of the instruction indicated by the T field; otherwise this field indicates that the entry is a marker with the meaning specified by the T and P fields.

When the EA field contains a non-zero value, bits 62:63 have the following meanings.

62 **Target Address (T)**

0 The EA field contains bits 0:61 of the effective address of a *Branch* instruction for which the branch was taken.
1 The EA field contains bits 0:61 of the branch target effective address corresponding to an XL-form *Branch* instruction.

63 **Prediction (P)**
When T=0, this field has the following meaning.

0 The outcome of the *Branch* instruction was correctly predicted.
1 The outcome of the *Branch* instruction was mispredicted.

When T=1, this field has the following meaning.
0 The Branch instruction was predicted to be taken and the target address was predicted correctly, or the target address was not predicted because the *Branch* instruction was predicted to be not taken.
1 The target address was mispredicted.

When the EA field contains a zero value, either the entry is a marker or is an entry for a *Branch* instruction for which the instruction address or target adress is 0.

Programming Note

It is expected that programs will not contain *Branch* instructions with instruction or target addresses equal to 0. If such instructions exist, however, markers become unusable because BHRB entries for various markers cannot be distinguished from entries for *Branch* instructions with instruction or target addresses equal to 0.

For *Branch* instructions with zero instruction or target addresses, the EA field is set to 0 and bits 62:63 are specified above. For *Branch* instructions with non-zero effective and target addresses, the EA field is set to 0 and bits 62:63 specify the type of marker as described below.

Value	Meaning
00	This entry is either not implemented or has been cleared, and there are no valid entries beyond the current entry.
01	A *Branch* instruction for which the branch was taken was executed, but the hardware was unable to enter its effective address and, for XL-Form *Branch* instructions, its target effective address.
10	Reserved
11	The previous entry contains bits 0:61 of the effective address of an XL-form *Branch* instruction for which the branch was taken, and the filtering mode required bits 0:61 of the current entry to indicate the effective address of the branch target, but the hardware was unable to enter the effective address of the branch target.

2.5 Branch Instructions

The sequence of instruction execution can be changed by the *Branch* instructions. Because all instructions are on word boundaries, bits 62 and 63 of the generated branch target address are ignored by the processor in performing the branch.

The *Branch* instructions compute the effective address (EA) of the target in one of the following five ways, as described in Section 1.10.3, "Effective Address Calculation" on page 26.

1. Adding a displacement to the address of the *Branch* instruction (*Branch* or *Branch Conditiona*l with AA=0).
2. Specifying an absolute address (*Branch* or *Branch Conditional* with AA=1).
3. Using the address contained in the Link Register (*Branch Conditional to Link Register*).
4. Using the address contained in the Count Register (*Branch Conditional to Count Register*).
5. Using the address contained in the Target Address Register (*Branch Conditional to Target Address Register*).

In all five cases, in 32-bit mode the final step in the address computation is setting the high-order 32 bits of the target address to 0.

For the first two methods, the target addresses can be computed sufficiently ahead of the *Branch* instruction that instructions can be prefetched along the target path. For the third through fifth methods, prefetching instructions along the target path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the *Branch* instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the *Branch* instruction is placed into the Link Register after the branch target address has been computed; this is done regardless of whether the branch is taken.

For *Branch Conditional* instructions, the BO field specifies the conditions under which the branch is taken, as shown in Figure 44. In the figure, M=0 in 64-bit mode and M=32 in 32-bit mode.

BO	Description
0000z	Decrement the CTR, then branch if the decremented $CTR_{M:63}\neq 0$ and $CR_{BI}=0$
0001z	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$ and $CR_{BI}=0$
001at	Branch if $CR_{BI}=0$
0100z	Decrement the CTR, then branch if the decremented $CTR_{M:63}\neq 0$ and $CR_{BI}=1$
0101z	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$ and $CR_{BI}=1$
011at	Branch if $CR_{BI}=1$
1a00t	Decrement the CTR, then branch if the decremented $CTR_{M:63}\neq 0$
1a01t	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$
1z1zz	Branch always

Notes:
1. "z" denotes a bit that is ignored.
2. The "a" and "t" bits are used as described below.

Figure 44. BO field encodings

The "a" and "t" bits of the BO field can be used by software to provide a hint about whether the branch is likely to be taken or is likely not to be taken, as shown in Figure 45.

at	Hint
00	No hint is given
01	Reserved
10	The branch is very likely not to be taken
11	The branch is very likely to be taken

Figure 45. "at" bit encodings

Programming Note

Many implementations have dynamic mechanisms for predicting whether a branch will be taken. Because the dynamic prediction is likely to be very accurate, and is likely to be overridden by any hint provided by the "at" bits, the "at" bits should be set to 0b00 unless the static prediction implied by at=0b10 or at=0b11 is highly likely to be correct.

For *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions, the BH field

provides a hint about the use of the instruction, as shown in Figure 46.

BH	Hint
00	***bclr***[***l***]: The instruction is a subroutine return ***bcctr***[***l***] and ***bctar***[***l***]:The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken
01	***bclr***[***l***]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken ***bcctr***[***l***] and ***bctar***[***l***]:Reserved
10	Reserved
11	***bclr***[***l***], ***bcctr***[***l***], and ***bctar***[l]: The target address is not predictable

Figure 46. BH field encodings

Programming Note

The hint provided by the BH field is independent of the hint provided by the "at" bits (e.g., the BH field provides no indication of whether the branch is likely to be taken).

Extended mnemonics for branches

Many extended mnemonics are provided so that *Branch Conditional* instructions can be coded with portions of the BO and BI fields as part of the mnemonic rather than as part of a numeric operand. Some of these are shown as examples with the Branch instructions. See Appendix E for additional extended mnemonics.

Programming Note

The hints provided by the "at" bits and by the BH field do not affect the results of executing the instruction.

The "z" bits should be set to 0, because they may be assigned a meaning in some future version of the architecture.

Programming Note

Many implementations have dynamic mechanisms for predicting the target addresses of ***bclr***[***l***] and ***bcctr***[***l***] instructions. These mechanisms may cache return addresses (i.e., Link Register values set by *Branch* instructions for which LK=1 and for which the branch was taken, other than the special form shown in the first example below) and recently used branch target addresses. To obtain the best performance across the widest range of implementations, the programmer should obey the following rules.

- Use *Branch* instructions for which LK=1 only as subroutine calls (including function calls, etc.), or in the special form shown in the first example below.
- Pair each subroutine call (i.e., each *Branch* instruction for which LK=1 and the branch is taken, other than the special form shown in the first example below) with a ***bclr*** instruction that returns from the subroutine and has BH=0b00.
- Do not use ***bclrl*** as a subroutine call. (Some implementations access the return address cache at most once per instruction; such implementations are likely to treat ***bclrl*** as a subroutine return, and not as a subroutine call.)
- For ***bclr***[***l***] and ***bcctr***[***l***], use the appropriate value in the BH field.

The following are examples of programming conventions that obey these rules. In the examples, BH is assumed to contain 0b00 unless otherwise stated. In addition, the “at” bits are assumed to be coded appropriately.

Let A, B, and Glue be specific programs.

- Obtaining the address of the next instruction:
  Use the following form of *Branch and Link*.
  ```
  bcl   20,31,$+4
  ```
- Loop counts:
  Keep them in the Count Register, and use a ***bc*** instruction (LK=0) to decrement the count and to branch back to the beginning of the loop if the decremented count is nonzero.
- Computed goto’s, case statements, etc.:
  Use the Count Register to hold the address to branch to, and use a ***bcctr*** instruction (LK=0, and BH=0b11 if appropriate) to branch to the selected address.
- Direct subroutine linkage:
  Here A calls B and B returns to A. The two branches should be as follows.
  - A calls B: use a ***bl*** or ***bcl*** instruction (LK=1).
  - B returns to A: use a ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).
- Indirect subroutine linkage:
  Here A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller; the Binder inserts “glue” code to mediate the branch.) The three branches should be as follows.
  - A calls Glue: use a ***bl*** or ***bcl*** instruction (LK=1).
  - Glue calls B: place the address of B into the Count Register, and use a ***bcctr*** instruction (LK=0).
  - B returns to A: use a ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).
- Function call:
  Here A calls a function, the identity of which may vary from one instance of the call to another, instead of calling a specific program B. This case should be handled using the conventions of the preceding two bullets, depending on whether the call is direct or indirect, with the following differences.
  - If the call is direct, place the address of the function into the Count Register, and use a ***bcctrl*** instruction (LK=1) instead of a ***bl*** or ***bcl*** instruction.
  - For the ***bcctr***[***l***] instruction that branches to the function, use BH=0b11 if appropriate.

Compatibility Note

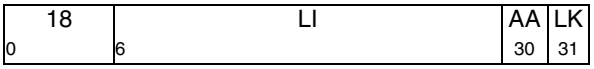
The bits corresponding to the current “a” and “t” bits, and to the current “z” bits except in the “branch always” BO encoding, had different meanings in versions of the architecture that precede Version 2.00.

- The bit corresponding to the “t” bit was called the “y” bit. The “y” bit indicated whether to use the architected default prediction (y=0) or to use the complement of the default prediction (y=1). The default prediction was defined as follows.
  - If the instruction is ***bc***[***l***][***a***] with a negative value in the displacement field, the branch is taken. (This is the only case in which the prediction corresponding to the “y” bit differs from the prediction corresponding to the “t” bit.)
  - In all other cases (***bc***[***l***][***a***] with a nonnegative value in the displacement field, ***bclr***[***l***], or ***bcctr***[***l***]), the branch is not taken.
- The BO encodings that test both the Count Register and the Condition Register had a “y” bit in place of the current “z” bit. The meaning of the “y” bit was as described in the preceding item.
- The “a” bit was a “z” bit.

Because these bits have always been defined either to be ignored or to be treated as hints, a given program will produce the same result on any implementation regardless of the values of the bits. Also, because even the “y” bit is ignored, in practice, by most processors that comply with versions of the architecture that precede Version 2.00, the performance of a given program on those processors will not be affected by the values of the bits.

Branch *I-form*

b	target_addr	(AA=0 LK=0)
ba	target_addr	(AA=1 LK=0)
bl	target_addr	(AA=0 LK=1)
bla	target_addr	(AA=1 LK=1)




```
if AA then NIA ←iea EXTS(LI || 0b00)
else       NIA ←iea CIA + EXTS(LI || 0b00)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

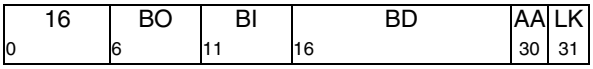
If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Branch Conditional *B-form*

bc	BO,BI,target_addr	(AA=0 LK=0)
bca	BO,BI,target_addr	(AA=1 LK=0)
bcl	BO,BI,target_addr	(AA=0 LK=1)
bcla	BO,BI,target_addr	(AA=1 LK=1)




```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then
  if AA then NIA ←iea EXTS(BD || 0b00)
  else       NIA ←iea CIA + EXTS(BD || 0b00)
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 44. *target_addr* specifies the branch target address.

If AA=0 then the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value BD || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR (if BO_2=0)
LR (if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional*:

Extended:		**Equivalent to:**	
blt	target	bc	12,0,target
bne	cr2,target	bc	4,10,target
bdnz	target	bc	16,0,target

Branch Conditional to Link Register *XL-form*

bclr	BO,BI,BH	(LK=0)
bclrl	BO,BI,BH	(LK=1)

19	BO	BI	///	BH	16	LK
0	6	11	16	19	21	31

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then NIA ←iea LR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 44. The BH field is used as described in Figure 46. The branch target address is $LR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Link Register*:

Extended:		Equivalent to:	
bclr	4,6	bclr	4,6,0
bltlr		bclr	12,0,0
bnelr	cr2	bclr	4,10,0
bdnzlr		bclr	16,0,0

Programming Note

bclr, ***bclrl***, ***bcctr***, and ***bcctrl*** each serve as both a basic and an extended mnemonic. The Assembler will recognize a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with three operands as the basic form, and a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00.

Branch Conditional to Count Register *XL-form*

bcctr	BO,BI,BH	(LK=0)
bcctrl	BO,BI,BH	(LK=1)

19	BO	BI	///	BH	528	LK
0	6	11	16	19	21	31

```
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if cond_ok then NIA ←iea CTR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 44. The BH field is used as described in Figure 46. The branch target address is $CTR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

If the “decrement and test CTR” option is specified (BO_2=0), the instruction form is invalid.

Special Registers Altered:

LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Count Register.*

Extended:		Equivalent to:	
bcctr	4,6	bcctr	4,6,0
bltctr		bcctr	12,0,0
bnectr	cr2	bcctr	4,10,0

Branch Conditional to Branch Target Address Register

XL-form

bctar BO,BI,BH (LK=0)
bctarl BO,BI,BH (LK=1)

19	BO	BI	///	BH	560	LK
0	6	11	16	19	21	31

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then NIA ←iea TAR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 44. The BH field is used as described in Figure 46. The branch target address is $TAR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR (if BO_2=0)
LR (if LK=1)

Programming Note

In some systems, the system program will restrict usage of the ***bctar****[l]* instruction to only selected programs. If an attempt is made to execute the instruction when it is not available, the system error handler will be invoked. See Book III-S for additional information.

2.6 Condition Register Instructions

2.6.1 Condition Register Logical Instructions

The *Condition Register Logical* instructions have preferred forms; see Section 1.8.1. In the preferred forms, the BT and BB fields satisfy the following rule.

- The bit specified by BT is in the same Condition Register field as the bit specified by BB.

Extended mnemonics for Condition Register logical operations

A set of extended mnemonics is provided that allow additional Condition Register logical operations, beyond those provided by the basic *Condition Register Logical* instructions, to be coded easily. Some of these are shown as examples with the *Condition Register Logical* instructions. See Appendix E for additional extended mnemonics.

Condition Register AND *XL-form*

crand BT,BA,BB

19	BT	BA	BB	257	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32}$ & CR_{BB+32}

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register NAND *XL-form*

crnand BT,BA,BB

19	BT	BA	BB	225	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow \neg(CR_{BA+32}$ & $CR_{BB+32})$

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register OR *XL-form*

cror BT,BA,BB

19	BT	BA	BB	449	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \mid CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register OR*:

Extended:	Equivalent to:
crmove Bx,By	cror Bx,By,By

Condition Register XOR *XL-form*

crxor BT,BA,BB

19	BT	BA	BB	193	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \oplus CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register XOR*:

Extended:	Equivalent to:
crclr Bx	crxor Bx,Bx,Bx

Condition Register NOR *XL-form*

crnor BT,BA,BB

19	BT	BA	BB	33	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow \neg(CR_{BA+32} \mid CR_{BB+32})$

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register NOR*:

Extended:	Equivalent to:
crnot Bx,By	crnor Bx,By,By

Condition Register Equivalent *XL-form*

creqv BT,BA,BB

19	BT	BA	BB	289	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \equiv CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register Equivalent*:

Extended:	Equivalent to:
crset Bx	creqv Bx,Bx,Bx

Condition Register AND with Complement *XL-form*

crandc BT,BA,BB

19	BT	BA	BB	129	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \ \& \ \neg CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ANDed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register OR with Complement *XL-form*

crorc BT,BA,BB

19	BT	BA	BB	417	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \mid \neg CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ORed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

2.6.2 Condition Register Field Instruction

Move Condition Register Field *XL-form*

mcrf BF,BFA

19	BF	//	BFA	//	///	0	/
0	6	9	11	14	16	21	31

$CR_{4\times BF+32:4\times BF+35} \leftarrow CR_{4\times BFA+32:4\times BFA+35}$

The contents of Condition Register field BFA are copied to Condition Register field BF.

Special Registers Altered:

CR field BF

2.7 System Call Instruction

This instruction provides the means by which a program can call upon the system to perform a service.

System Call *SC-form*

sc LEV

17	///	///	//	LEV	//	1	/
0	6	11	16	20	27	30	31

This instruction calls the system to perform a service. A complete description of this instruction can be found in Book III.

The use of the LEV field is described in Book III. The LEV values greater than 1 are reserved, and bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

When control is returned to the program that executed the *System Call* instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

This instruction is context synchronizing (see Book III).

Special Registers Altered:

Dependent on the system service

Programming Note

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an ***sc*** mnemonic with one operand as the basic form, and an ***sc*** mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

In application programs the value of the LEV operand for ***sc*** should be 0.

2.8 Branch History Rolling Buffer Instructions

The Branch History Rolling Buffer instructions enable problem state programs to access and clear the BHRB. The availability of these instructions is controlled by the system program. (See Chapter 9 of Book III-S.) When an attempt is made to execute these instructions when they are unavailable, the Facility Unavailable error handler is invoked.

Clear BHRB *X-form*

clrbhrb

31	///	430	/
0	6	21	31

```
for n = 0:1023
BHRB(n) ← 0
```

All BHRB entries are set to 0s.

Special Registers Altered:

None.

Move From Branch History Rolling Buffer *XFX-form*

mfbhrbe RT,BHRBE

31	RT	bhrbe	302	/
0	6	11	21	31

```
n ← bhrbe0:9
If n < number of BHRBEs implemented,
then
 RT ← BHRBE(n)
else
 RT ← 0
```

The BHRBE field denotes an entry in the BHRB. If the designated entry is within the range of BHRB entries implemented, the contents of the designated BHRB entry are placed into register RT; otherwise, 0s are placed into RT.

In order to ensure that the current BHRB contents are read by this instruction, an event-based branch or ***rfebb*** (see Chapter 7 of Book II) or a non-recoverable interrupt or *Return from Interrupt* instruction (see Book III-S Chapter 6) must have occurred prior to this instruction and after all previous *Branch* and ***clrbhrb*** instructions have completed.

Special Registers Altered:

None

Programming Note

In order to read all the BHRB entries containing information about taken branches, software reads the entries starting from entry number 0 and continuing until an entry containing 0s is read or until all implemented BHRB entries have been read.

Programming Note

Since the number of BHRB entries may decrease or the BHRB may be cleared at any time, reading an entry for a second time may result in a zero value being returned even though the program has not executed ***clrbhrb***.

Chapter 3. Fixed-Point Facility

3.1 Fixed-Point Facility Overview

This chapter describes the registers and instructions that make up the Fixed-Point Facility.

3.2 Fixed-Point Facility Registers

3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Facility. The principal storage internal to the Fixed-Point Facility is a set of 32 General Purpose Registers (GPRs). See Figure 47.

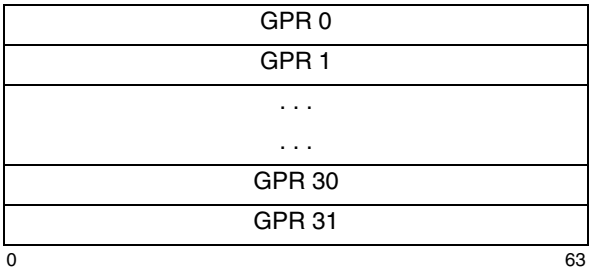


Figure 47. General Purpose Registers

Each GPR is a 64-bit register.

3.2.2 Fixed-Point Exception Register

The Fixed-Point Exception Register (XER) is a 64-bit register.

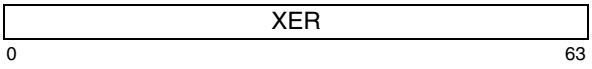


Figure 48. Fixed-Point Exception Register

The bit definitions for the Fixed-Point Exception Register are shown below. Here M=0 in 64-bit mode and M=32 in 32-bit mode.

The bits are set based on the operation of an instruction considered as a whole, not on intermediate results (e.g., the *Subtract From Carrying* instruction, the result of which is specified as the sum of three values, sets bits in the Fixed-Point Exception Register based on the entire operation, not on an intermediate sum).

Bit(s	Description
0:31	Reserved
32	***Summary Overflow*** (SO) The Summary Overflow bit is set to 1 whenever an instruction (except ***mtspr***) sets the Overflow bit. Once set, the SO bit remains set until it is cleared by an ***mtspr*** instruction (specifying the XER) or an ***mcrxr*** instruction. It is not altered by *Compare* instructions, or by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) that cannot overflow. Executing an ***mtspr*** instruction to the XER, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1.
33	***Overflow*** (OV) The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. XO-form *Add*, *Subtract From*, and *Negate* instructions having OE=1 set it to 1 if the carry out of bit M is not equal to the carry out of bit M+1, and set it to 0 otherwise. XO-form *Multiply Low* and *Divide* instructions having OE=1 set it to 1 if the result cannot be represented in 64 bits (***mulld***, ***divd***, ***divde***, ***divdu***, ***divdeu***) or in 32 bits (***mullw***, ***divw***, ***divwe***, ***divwu***, ***divweu***), and set it to 0 otherwise. The OV bit is not altered by *Compare*

instructions, or by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) that cannot overflow.

[Category:
Legacy Integer Multiply-Accumulate]
XO-form *Legacy Integer Multiply-Accumulate* instructions set OV when OE=1 to reflect overflow of the 32-bit result. For signed-integer accumulation, overflow occurs when the add produces a carry out of bit 32 that is not equal to the carry out of bit 33. For unsigned-integer accumulation, overflow occurs when the add produces a carry out of bit 32.

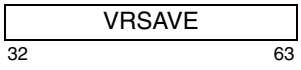
34 ***Carry*** (CA)
The Carry bit is set as follows, during execution of certain instructions. *Add Carrying*, *Subtract From Carrying*, *Add Extended*, and *Subtract From Extended* types of instructions set it to 1 if there is a carry out of bit M, and set it to 0 otherwise. *Shift Right Algebraic* instructions set it to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by *Compare* instructions, or by other instructions (except *Shift Right Algebraic*, ***mtspr*** to the XER, and ***mcrxr***) that cannot carry.

35:56 Reserved

57:63 This field specifies the number of bytes to be transferred by a *Load String Indexed* or *Store String Indexed* instruction.

[Category: Legacy Move Assist]
This field is used as a target by ***dmlzb*** to indicate the byte location of the leftmost zero byte found.

3.2.3 VR Save Register



The VR Save Register (VRSAVE) is a 32-bit register that can be used as a software use SPR; see Sections 3.2.4 and 5.3.3.

Architecture Note

In versions of the Architecture that precede Version 2.05, the VRSAVE Register was in the Vector category (see Section 5.3.3). It is now included in the Base category to simplify operating system support of and migration between heterogeneous systems containing processors with and without support for the Vector category.

3.2.4 Software Use SPRs [Category: Embedded]

Software Use SPRs are 64-bit registers that have no defined functionality. SPRG4-7 can be read by application programs. Additional Software Use SPRs are defined in Book III.

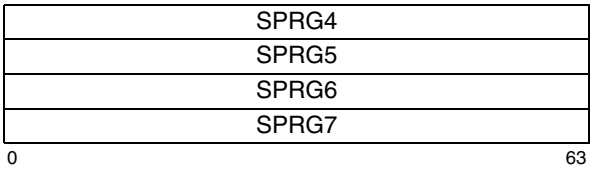


Figure 49. Software-use SPRs

Programming Note

USPRG0 was made a 32-bit register and renamed to VRSAVE; see Sections 3.2.3 and 5.3.3.

3.2.5 Device Control Registers [Category: Embedded.Device Control]

Device Control Registers (DCRs) are on-chip registers that exist architecturally outside the processor and thus are not actually part of the processor architecture. This specification simply defines the existence of a Device Control Register 'address space' and the instructions to access them and does not define the Device Control Registers themselves.

Device Control Registers may control the use of on-chip peripherals, such as memory controllers (the definition of specific Device Control Registers is implementation-dependent).

The contents of user-mode-accessible Device Control Registers can be read using ***mfdcrux*** and written using ***mtdcrux***.

3.3 Fixed-Point Facility Instructions

3.3.1 Fixed-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3 on page 26.

Programming Note

The ***la*** extended mnemonic permits computing an effective address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address.

Programming Note

The DS field in DS-form Storage Access instructions is a word offset, not a byte offset like the D field in D-form *Storage Access* instructions. However, for programming convenience, Assemblers should support the specification of byte offsets for both forms of instruction.

3.3.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

3.3.2 Fixed-Point Load Instructions

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into register RT.

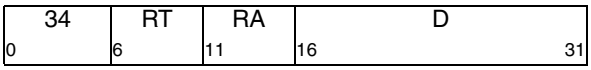
Many of the *Load* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into register RA and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT.

Programming Note

In some implementations, the *Load Algebraic* and *Load with Update* instructions may have greater latency than other types of *Load* instructions. Moreover, *Load with Update* instructions may take longer to execute in some implementations than the corresponding pair of a non-update *Load* instruction and an *Add* instruction.

Load Byte and Zero *D-form*

lbz RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 56 0 || MEM(EA, 1)
```

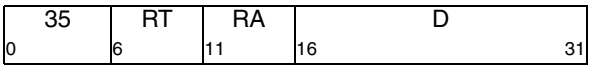
Let the effective address (EA) be the sum (RA|0)+ D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Byte and Zero with Update *D-form*

lbzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 56 0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

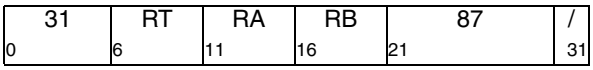
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Byte and Zero Indexed *X-form*

lbzx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 56 0 || MEM(EA, 1)
```

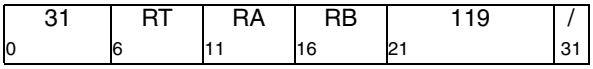
Let the effective address (EA) be the sum (RA|0)+ (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Byte and Zero with Update Indexed *X-form*

lbzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 56 0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword and Zero D-form

lhz RT,D(RA)

40	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 48 0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero Indexed X-form

lhzx RT,RA,RB

31	RT	RA	RB	279	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48 0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero with Update D-form

lhzu RT,D(RA)

41	RT	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
RT ← 48 0 || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword and Zero with Update Indexed X-form

lhzux RT,RA,RB

31	RT	RA	RB	311	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
RT ← 48 0 || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

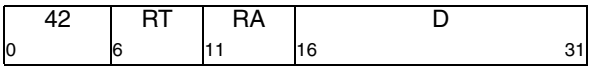
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic D-form

lha RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← EXTS(MEM(EA, 2))
```

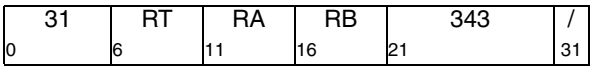
Let the effective address (EA) be the sum (RA|0)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Load Halfword Algebraic Indexed X-form

lhax RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← EXTS(MEM(EA, 2))
```

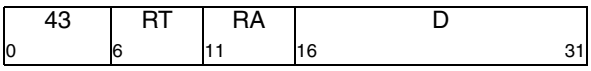
Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Load Halfword Algebraic with Update D-form

lhau RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.

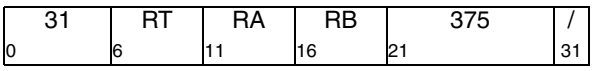
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic with Update Indexed X-form

lhaux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.

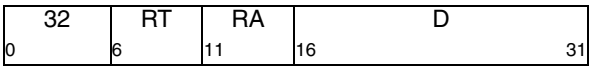
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero D-form

lwz RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 320 || MEM(EA, 4)
```

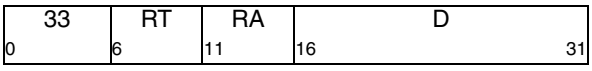
Let the effective address (EA) be the sum (RA|0)+ D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update D-form

lwzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 320 || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

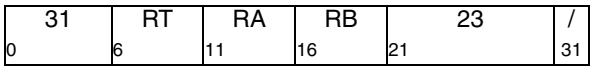
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero Indexed X-form

lwzx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 320 || MEM(EA, 4)
```

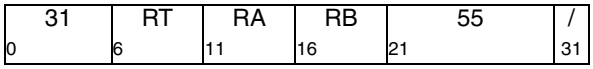
Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update Indexed X-form

lwzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 320 || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

3.3.2.1 64-bit Fixed-Point Load Instructions [Category: 64-Bit]

Load Word Algebraic *DS-form*

lwa RT,DS(RA)

58	RT	RA	DS	2
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DS || 0b00)
RT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Word Algebraic Indexed *X-form*

lwax RT,RA,RB

31	RT	RA	RB	341	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Word Algebraic with Update Indexed *X-form*

lwaux RT,RA,RB

31	RT	RA	RB	373	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
RT ← EXTS(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

EA is placed into register RA.

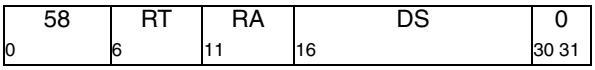
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Doubleword *DS-form*

ld RT,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS || 0b00)
RT ← MEM(EA, 8)
```

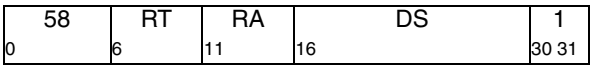
Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

Special Registers Altered:

None

Load Doubleword with Update *DS-form*

ldu RT,DS(RA)




```
EA ← (RA) + EXTS(DS || 0b00)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

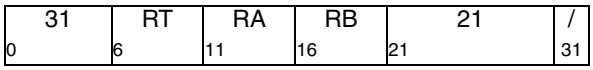
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Doubleword Indexed *X-form*

ldx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)
```

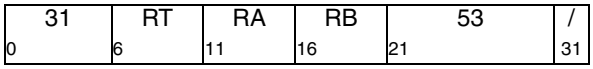
Let the effective address (EA) be the sum (RA|0)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

Special Registers Altered:

None

Load Doubleword with Update Indexed *X-form*

ldux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

3.3.3 Fixed-Point Store Instructions

The contents of register RS are stored into the byte, halfword, word, or doubleword in storage addressed by EA.

Many of the *Store* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, the following rules apply.

- If RA≠0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS are copied to the target storage element and then EA is placed into RA (RS).

Store Byte *D-form*

stb RS,D(RA)

38	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+ D. $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

Special Registers Altered:

None

Store Byte Indexed *X-form*

stbx RS,RA,RB

31	RS	RA	RB	215	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

Special Registers Altered:

None

Store Byte with Update *D-form*

stbu RS,D(RA)

39	RS	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Byte with Update Indexed *X-form*

stbux RS,RA,RB

31	RS	RA	RB	247	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

EA is placed into register RA.

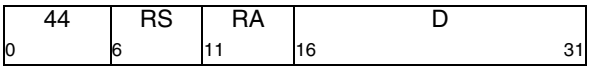
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword *D-form*

sth RS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 2) ← (RS)48:63
```

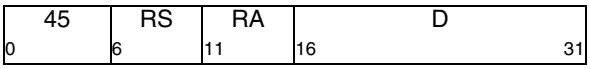
Let the effective address (EA) be the sum (RA|0)+ D. $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Halfword with Update *D-form*

sthu RS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

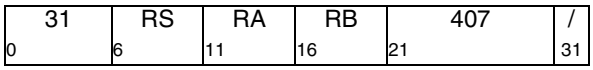
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword Indexed *X-form*

sthx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)48:63
```

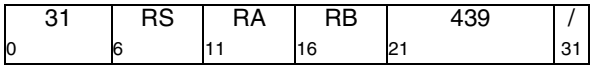
Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Halfword with Update Indexed *X-form*

sthux RS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

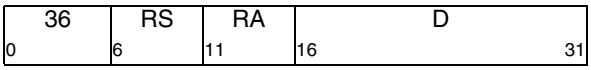
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word *D-form*

stw RS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← (RS)32:63
```

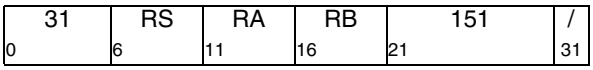
Let the effective address (EA) be the sum (RA|0)+ D. $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Word Indexed *X-form*

stwx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)32:63
```

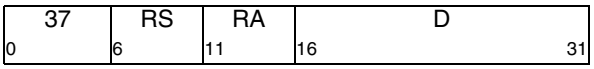
Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Word with Update *D-form*

stwu RS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

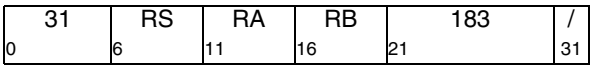
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word with Update Indexed *X-form*

stwux RS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

3.3.3.1 64-bit Fixed-Point Store Instructions [Category: 64-Bit]

Store Doubleword *DS-form*

std RS,DS(RA)

62	RS	RA	DS	0
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Doubleword Indexed *X-form*

stdx RS,RA,RB

31	RS	RA	RB	149	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Doubleword with Update *DS-form*

stdu RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Doubleword with Update Indexed *X-form*

stdux RS,RA,RB

31	RS	RA	RB	181	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

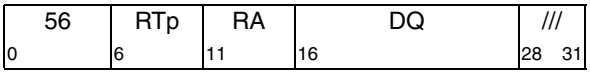
Special Registers Altered:

None

3.3.4 Fixed-Point Load and Store Quadword Instructions [Category: Load/ Store Quadword]

Load Quadword *DQ-form*

lq RTp,DQ(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DQ || 0b0000)
RTp ← MEM(EA, 16)
```

Let the effective address (EA) be the sum (RA|0)+ (DQ||0b0000). The quadword in storage addressed by EA is loaded into register pair RTp.

If RTp is odd or RTp=RA, the instruction form is invalid. If RTp=RA, an attempt to execute this instruction causes an Illegal Instruction type Program interrupt. (The RTp=RA case includes the case of RTp=RA=0.)

The quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In big-endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

Architecture Note

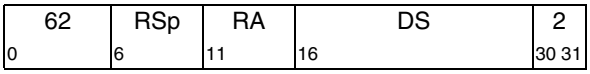
In versions of the architecture prior to 2.07, this instruction was privileged.

Special Registers Altered:

None

Store Quadword *DS-form*

stq RSp,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 16) ← RSp
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). The contents of register pair RSp are stored into the quadword in storage addressed by EA.

If RSp is odd, the instruction form is invalid.

The contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In big-endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

Architecture Note

In versions of the architecture prior to 2.07, this instruction was privileged.

Special Registers Altered:

None

3.3.5 Fixed-Point Load and Store with Byte Reversal Instructions

Programming Note

These instructions have the effect of loading and storing data in the opposite byte ordering from that which would be used by other *Load* and *Store* instructions.

Programming Note

In some implementations, the Load Byte-Reverse instructions may have greater latency than other Load instructions.

Load Halfword Byte-Reverse Indexed X-form

lhbrx RT,RA,RB

31	RT	RA	RB	790	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 2)
RT ← 48 0 || load_data8:15 || load_data0:7
```

Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the halfword in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the halfword in storage addressed by EA are loaded into $RT_{48:55}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Store Halfword Byte-Reverse Indexed X-form

sthbrx RS,RA,RB

31	RS	RA	RB	918	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)56:63 || (RS)48:55
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into bits 0:7 of the halfword in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the halfword in storage addressed by EA.

Special Registers Altered:

None

Load Word Byte-Reverse Indexed X-form

lwbrx RT,RA,RB

31	RT	RA	RB	534	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 4)
RT ← 32 0 || load_data24:31 || load_data16:23
        || load_data8:15 || load_data0:7
```

Let the effective address (EA) be the sum (RA|0)+ (RB). Bits 0:7 of the word in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the word in storage addressed by EA are loaded into $RT_{48:55}$. Bits 16:23 of the word in storage addressed by EA are loaded into $RT_{40:47}$. Bits 24:31 of the word in storage addressed by EA are loaded into $RT_{32:39}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Store Word Byte-Reverse Indexed X-form

stwbrx RS,RA,RB

31	RS	RA	RB	662	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)56:63 || (RS)48:55 || (RS)40:47
             ||(RS)32:39
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into bits 0:7 of the word in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the word in storage addressed by EA. $(RS)_{40:47}$ are stored into bits 16:23 of the word in storage addressed by EA. $(RS)_{32:39}$ are stored into bits 24:31 of the word in storage addressed by EA.

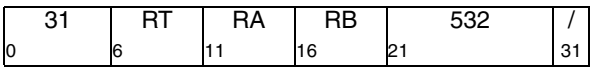
Special Registers Altered:

None

3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions [Category: 64-bit]

Load Doubleword Byte-Reverse Indexed X-form

ldbrx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 8)
RT ←  load_data56:63 || load_data48:55
   || load_data40:47 || load_data32:39
   || load_data24:31 || load_data16:23
   || load_data8:15  || load_data0:7
```

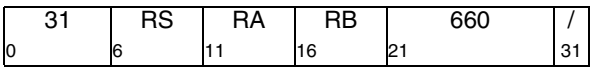
Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the doubleword in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the doubleword in storage addressed by EA are loaded into $RT_{48:55}$. Bits 16:23 of the doubleword in storage addressed by EA are loaded into $RT_{40:47}$. Bits 24:31 of the doubleword in storage addressed by EA are loaded into $RT_{32:39}$. Bits 32:39 of the doubleword in storage addressed by EA are loaded into $RT_{24:31}$. Bits 40:47 of the doubleword in storage addressed by EA are loaded into $RT_{16:23}$. Bits 48:55 of the doubleword in storage addressed by EA are loaded into $RT_{8:15}$. Bits 56:63 of the doubleword in storage addressed by EA are loaded into $RT_{0:7}$.

Special Registers Altered:

None

Store Doubleword Byte-Reverse Indexed X-form

stdbrx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)56:63 || (RS)48:55
          || (RS)40:47 || (RS)32:39
          || (RS)24:31 || (RS)16:23
          || (RS)8:15  || (RS)0:7
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into bits 0:7 of the doubleword in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the doubleword in storage addressed by EA. $(RS)_{40:47}$ are stored into bits 16:23 of the doubleword in storage addressed by EA. $(RS)_{32:39}$ are stored into bits 23:31 of the doubleword in storage addressed by EA. $(RS)_{24:31}$ are stored into bits 32:39 of the doubleword in storage addressed by EA. $(RS)_{16:23}$ are stored into bits 40:47 of the doubleword in storage addressed by EA. $(RS)_{8:15}$ are stored into bits 48:55 of the doubleword in storage addressed by EA. $(RS)_{0:7}$ are stored into bits 56:63 of the doubleword in storage addressed by EA.

Special Registers Altered:

None

3.3.6 Fixed-Point Load and Store Multiple Instructions

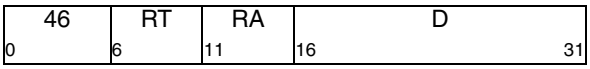
The *Load/Store Multiple* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 22. In the preferred forms, storage alignment satisfies the following rule.

- The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned quadword in storage.

For the Server environment, the *Load/Store Multiple* instructions are not supported in Little-Endian mode. If they are executed in Little-Endian mode, the system alignment error handler is invoked.

Load Multiple Word *D-form*

lmw RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
r ← RT
do while r ≤ 31
   GPR(r) ← 320 || MEM(EA, 4)
   r ← r + 1
   EA ← EA + 4
```

Let n = (32 - RT). Let the effective address (EA) be the sum (RA|0)+ D.

n consecutive words starting at EA are loaded into the low-order 32 bits of GPRs RT through 31. The high-order 32 bits of these GPRs are set to zero.

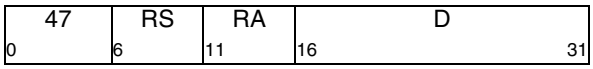
If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Multiple Word *D-form*

stmw RS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
r ← RS
do while r ≤ 31
   MEM(EA, 4) ← GPR(r)32:63
   r ← r + 1
   EA ← EA + 4
```

Let n = (32 - RS). Let the effective address (EA) be the sum (RA|0)+ D.

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs RS through 31.

Special Registers Altered:

None

3.3.7 Fixed-Point Move Assist Instructions [Category: Move Assist]

The *Move Assist* instructions allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

The *Load/Store String* instructions have preferred forms; see Section 1.8.1, “Preferred Instruction Forms” on page 22. In the preferred forms, register usage satisfies the following rules.

- RS = 4 or 5
- RT = 4 or 5
- last register loaded/stored ≤ 12

For some implementations, using GPR 4 for RS and RT may result in slightly faster execution than using GPR 5.

For the Server environment, the *Move Assist* instructions are not supported in Little-Endian mode. If they are executed in Little-Endian mode, the system alignment error handler may be invoked or the instructions may be treated as no-ops if the number of bytes specified by the instruction is 0.

Load String Word Immediate *X-form*

lswi RT,RA,NB

31	RT	RA	NB	597	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RT - 1
i ← 32
do while n > 0
   if i = 32 then
     r ← r + 1 (mod 32)
     GPR(r) ← 0
   GPR(r)i:i+7 ← MEM(EA, 1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load String Word Indexed *X-form*

lswx RT,RA,RB

31	RT	RA	RB	533	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← XER57:63
r ← RT - 1
i ← 32
RT ← undefined
do while n > 0
   if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
   GPR(r)i:i+7 ← MEM(EA, 1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

Let the effective address (EA) be the sum (RA|0)+ (RB). Let $n=XER_{57:63}$; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the contents of register RT are undefined.

If RA or RB is in the range of registers to be loaded, including the case in which RA=0, the instruction is treated as if the instruction form were invalid. If RT=RA or RT=RB, the instruction form is invalid.

Special Registers Altered:

None

Store String Word Immediate *X-form*

stswi RS,RA,NB

31	RS	RA	NB	725	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
   i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to store. Let nr =CEIL(n/4); nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

Special Registers Altered:

None

Store String Word Indexed *X-form*

stswx RS,RA,RB

31	RS	RA	RB	661	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← XER57:63
r ← RS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
    i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be the sum (RA|0)+ (RB). Let n = $XER_{57:63}$; n is the number of bytes to store. Let nr = CEIL(n/4); nr is the number of registers to supply data.

If n>0, n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

If n=0, no bytes are stored.

Special Registers Altered:

None

3.3.8 Other Fixed-Point Instructions

The remainder of the fixed-point instructions use the contents of the General Purpose Registers (GPRs) as source operands, and place results into GPRs, into the Fixed-Point Exception Register (XER), and into Condition Register fields. In addition, the *Trap* instructions test the contents of a GPR or XER bit, invoking the system trap handler if the result of the specified test is true.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The X-form and XO-form instructions with Rc=1, and the D-form instructions ***addic.***, ***andi.***, and ***andis.***, set the first three bits of CR Field 0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to zero. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

Unless otherwise noted and when appropriate, when CR Field 0 and the XER are set they reflect the value placed into the target register.

Programming Note

Instructions with the OE bit set or that set CA may execute slowly or may prevent the execution of subsequent instructions until the instruction has completed.

3.3.9 Fixed-Point Arithmetic Instructions

The XO-form *Arithmetic* instructions with Rc=1, and the D-form *Arithmetic* instruction ***addic.***, set the first three bits of CR Field 0 as described in Section 3.3.8, "Other Fixed-Point Instructions".

addic, ***addic.***, ***subfic***, ***addc***, ***subfc***, ***adde***, ***subfe***, ***addme***, ***subfme***, ***addze***, and ***subfze*** always set CA, to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode. The XO-form *Arithmetic* instructions set SO and OV when OE=1 to reflect overflow of the result. Except for the *Multiply Low* and *Divide* instructions, the setting of these bits is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For XO-form *Multiply Low* and *Divide* instructions, the setting of these bits is mode-independent, and reflects overflow of the 64-bit result for ***mulld***, ***divd***, ***divde***, ***divdu*** and ***divdeu***, and overflow of the low-order 32-bit result for ***mullw***, ***divw***, ***divwe***, ***divwu***, and ***divweu***.

Programming Note

Notice that CR Field 0 may not reflect the "true" (infinitely precise) result if overflow occurs.

Extended mnemonics for addition and subtraction

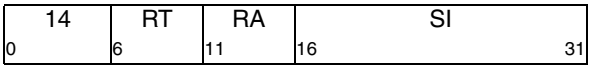
Several extended mnemonics are provided that use the *Add Immediate* and *Add Immediate Shifted* instructions to load an immediate value or an address into a target register. Some of these are shown as examples with the two instructions.

The Power ISA supplies *Subtract From* instructions, which subtract the second operand from the third. A set of extended mnemonics is provided that use the more "normal" order, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate *Add* and *Subtract From* instructions.

See Appendix E for additional extended mnemonics.

Add Immediate — *D-form*

addi RT,RA,SI




```
if RA = 0 then RT ← EXTS(SI)
else           RT ← (RA) + EXTS(SI)
```

The sum (RA|0) + SI is placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate*:

Extended:		Equivalent to:	
li	Rx,value	addi	Rx,0,value
la	Rx,disp(Ry)	addi	Rx,Ry,disp
subi	Rx,Ry,value	addi	Rx,Ry,-value

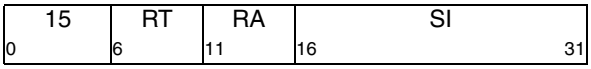
Programming Note

addi, ***addis***, ***add***, and ***subf*** are the preferred instructions for addition and subtraction, because they set few status bits.

Notice that ***addi*** and ***addis*** use the value 0, not the contents of GPR 0, if RA=0.

Add Immediate Shifted — *D-form*

addis RT,RA,SI




```
if RA = 0 then RT ← EXTS(SI || 16 0)
else           RT ← (RA) + EXTS(SI || 16 0)
```

The sum (RA|0) + (SI || 0x0000) is placed into register RT.

Special Registers Altered:

None

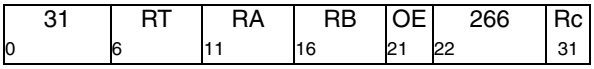
Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate Shifted*:

Extended:		Equivalent to:	
lis	Rx,value	addis	Rx,0,value
subis	Rx,Ry,value	addis	Rx,Ry,-value

Add *XO-form*

add	RT,RA,RB	(OE=0 Rc=0)
add.	RT,RA,RB	(OE=0 Rc=1)
addo	RT,RA,RB	(OE=1 Rc=0)
addo.	RT,RA,RB	(OE=1 Rc=1)



RT ← (RA) + (RB)

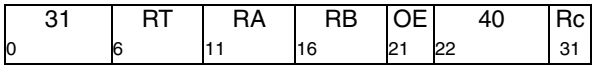
The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Subtract From *XO-form*

subf	RT,RA,RB	(OE=0 Rc=0)
subf.	RT,RA,RB	(OE=0 Rc=1)
subfo	RT,RA,RB	(OE=1 Rc=0)
subfo.	RT,RA,RB	(OE=1 Rc=1)



RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) +1 is placed into register RT.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

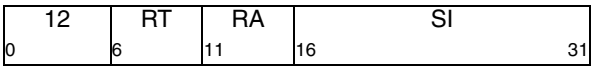
Extended Mnemonics:

Example of extended mnemonics for *Subtract From*:

Extended:		Equivalent to:	
sub	Rx,Ry,Rz	subf	Rx,Rz,Ry

Add Immediate Carrying *D-form*

addic RT,RA,SI



RT ← (RA) + EXTS(SI)

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CA

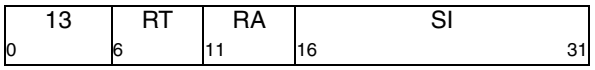
Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying*:

Extended:		Equivalent to:	
subic	Rx,Ry,value	addic	Rx,Ry,-value

Add Immediate Carrying and Record *D-form*

addic. RT,RA,SI



RT ← (RA) + EXTS(SI)

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CR0 CA

Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying and Record*:

Extended:		Equivalent to:	
subic.	Rx,Ry,value	addic.	Rx,Ry,-value

Subtract From Immediate Carrying *D-form*

subfic RT,RA,SI

8	RT	RA	SI
0	6	11	16 31

RT ← ¬(RA) + EXTS(SI) + 1

The sum ¬(RA) + SI + 1 is placed into register RT.

Special Registers Altered:

CA

Add Carrying *XO-form*

addc	RT,RA,RB	(OE=0 Rc=0)
addc.	RT,RA,RB	(OE=0 Rc=1)
addco	RT,RA,RB	(OE=1 Rc=0)
addco.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	10	Rc
0	6	11	16	21	22	31

RT ← (RA) + (RB)

The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Subtract From Carrying *XO-form*

subfc	RT,RA,RB	(OE=0 Rc=0)
subfc.	RT,RA,RB	(OE=0 Rc=1)
subfco	RT,RA,RB	(OE=1 Rc=0)
subfco.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	8	Rc
0	6	11	16	21	22	31

RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) + 1 is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Extended Mnemonics:

Example of extended mnemonics for *Subtract From Carrying*:

Extended:	**Equivalent to:**
subc Rx,Ry,Rz	subfc Rx,Rz,Ry

Add Extended *XO-form*

adde	RT,RA,RB	(OE=0 Rc=0)
adde.	RT,RA,RB	(OE=0 Rc=1)
addeo	RT,RA,RB	(OE=1 Rc=0)
addeo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	138	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + (RB) + CA
```

The sum (RA) + (RB) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Add to Minus One Extended *XO-form*

addme	RT,RA	(OE=0 Rc=0)
addme.	RT,RA	(OE=0 Rc=1)
addmeo	RT,RA	(OE=1 Rc=0)
addmeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	234	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + CA - 1
```

The sum (RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Subtract From Extended *XO-form*

subfe	RT,RA,RB	(OE=0 Rc=0)
subfe.	RT,RA,RB	(OE=0 Rc=1)
subfeo	RT,RA,RB	(OE=1 Rc=0)
subfeo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	136	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + (RB) + CA
```

The sum ¬(RA) + (RB) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Subtract From Minus One Extended *XO-form*

subfme	RT,RA	(OE=0 Rc=0)
subfme.	RT,RA	(OE=0 Rc=1)
subfmeo	RT,RA	(OE=1 Rc=0)
subfmeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	232	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + CA - 1
```

The sum ¬(RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Add to Zero Extended *XO-form*

addze	RT,RA	(OE=0 Rc=0)
addze.	RT,RA	(OE=0 Rc=1)
addzeo	RT,RA	(OE=1 Rc=0)
addzeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	202	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + CA
```

The sum (RA) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Subtract From Zero Extended *XO-form*

subfze	RT,RA	(OE=0 Rc=0)
subfze.	RT,RA	(OE=0 Rc=1)
subfzeo	RT,RA	(OE=1 Rc=0)
subfzeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	200	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + CA
```

The sum ¬(RA) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The setting of CA by the *Add* and *Subtract From* instructions, including the Extended versions thereof, is mode-dependent. If a sequence of these instructions is used to perform extended-precision addition or subtraction, the same mode should be used throughout the sequence.

Negate *XO-form*

neg	RT,RA	(OE=0 Rc=0)
neg.	RT,RA	(OE=0 Rc=1)
nego	RT,RA	(OE=1 Rc=0)
nego.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	104	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + 1
```

The sum ¬(RA) + 1 is placed into register RT.

If the processor is in 64-bit mode and register RA contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number and, if OE=1, OV is set to 1. Similarly, if the processor is in 32-bit mode and $(RA)_{32:63}$ contain the most negative 32-bit number (0x8000_0000), the low-order 32 bits of the result contain the most negative 32-bit number and, if OE=1, OV is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Multiply Low Immediate *D-form*

mulli RT,RA,SI

7	RT	RA	SI
0	6	11	16 31

$prod_{0:127} \leftarrow (RA) \times EXTS(SI)$
$RT \leftarrow prod_{64:127}$

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the SI field. The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

None

Multiply Low Word *XO-form*

mullw	RT,RA,RB	(OE=0 Rc=0)
mullw.	RT,RA,RB	(OE=0 Rc=1)
mullwo	RT,RA,RB	(OE=1 Rc=0)
mullwo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	235	Rc
0	6	11	16	21	22	31

$RT \leftarrow (RA)_{32:63} \times (RB)_{32:63}$

The 32-bit operands are the low-order 32 bits of RA and of RB. The 64-bit product of the operands is placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

For ***mulli*** and ***mullw***, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

For ***mulli*** and ***mulld***, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers. For ***mulli*** and ***mullw***, the low-order 32 bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

Multiply High Word *XO-form*

mulhw	RT,RA,RB	(Rc=0)
mulhw.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	75	Rc
0	6	11	16	21	22	31

$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the operands are placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)

Multiply High Word Unsigned *XO-form*

mulhwu	RT,RA,RB	(Rc=0)
mulhwu.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	11	Rc
0	6	11	16	21	22	31

$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the operands are placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined.

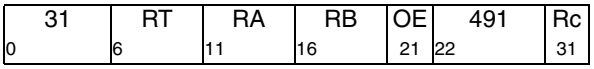
Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)

Divide Word *XO-form*

divw	RT,RA,RB	(OE=0 Rc=0)
divw.	RT,RA,RB	(OE=0 Rc=1)
divwo	RT,RA,RB	(OE=1 Rc=0)
divwo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 32-bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV (if OE=1)

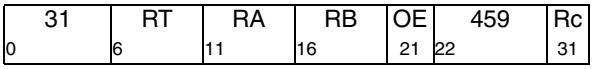
Programming Note

The 32-bit signed remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows, except in the case that $(RA)_{32:63} = -2^{31}$ and $(RB)_{32:63} = -1$.

```
divw   RT,RA,RB      # RT = quotient
mullw  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Word Unsigned *XO-form*

divwu	RT,RA,RB	(OE=0 Rc=0)
divwu.	RT,RA,RB	(OE=0 Rc=1)
divwuo	RT,RA,RB	(OE=1 Rc=0)
divwuo.	RT,RA,RB	(OE=1 Rc=1)



$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 32 bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r <$ divisor.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV (if OE=1)

Programming Note

The 32-bit unsigned remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows.

```
divwu  RT,RA,RB      # RT = quotient
mullw  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Word Extended *XO-form*

divwe RT,RA,RB (OE=0 Rc=0)
divwe. RT,RA,RB (OE=0 Rc=1)
divweo RT,RA,RB (OE=1 Rc=0)
divweo. RT,RA,RB (OE=1 Rc=1)
[Category: Server]
[Category: Embedded.Phased-In]

31	RT	RA	RB	OE	427	Rc
0	6	11	16	21	22	31

$dividend_{0:63} \leftarrow (RA)_{32:63} \,||\, {}^{32}0$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is $(RA)_{32:63} \,||\, {}^{32}0$. The 32-bit divisor is $(RB)_{32:63}$. If the quotient can be represented in 32 bits, it is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If the quotient cannot be represented in 32 bits, or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV (if OE=1)

Divide Word Extended Unsigned *XO-form*

divweu RT,RA,RB (OE=0 Rc=0)
divweu. RT,RA,RB (OE=0 Rc=1)
divweuo RT,RA,RB (OE=1 Rc=0)
divweuo. RT,RA,RB (OE=1 Rc=1)
[Category: Server]
[Category: Embedded.Phased-In]

31	RT	RA	RB	OE	395	Rc
0	6	11	16	21	22	31

$dividend_{0:63} \leftarrow (RA)_{32:63} \,||\, {}^{32}0$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is $(RA)_{32:63} \,||\, {}^{32}0$. The 32-bit divisor is $(RB)_{32:63}$. If the quotient can be represented in 32 bits, it is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r <$ divisor.

If (RA) ≥ (RB), or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV (if OE=1)

Programming Note

Unsigned long division of a 64-bit dividend contained in two 32-bit registers by a 32-bit divisor can be computed as follows. The algorithm is shown first, followed by Assembler code that implements the algorithm. The dividend is Dh || Dl, the divisor is Dv, and the quotient and remainder are Q and R respectively, where these variables and all intermediate variables represent unsigned 32-bit integers. It is assumed that Dv > Dh, and that assigning a value to an intermediate variable assigns the low-order 32 bits of the value and ignores any higher-order bits of the value. (In both the algorithm and the Assembler code, “r1” and “r2” refer to “remainder 1” and “remainder 2”, rather than to GPRs 1 and 2.)

Algorithm:

3. q1 ← **divweu** Dh, Dv
4. r1 ← -(q1 × Dv) # remainder of step 1 divide operation (see Note 1)
5. q2 ← **divwu** Dl, Dv
6. r2 ← Dl - (q2 × Dv) # remainder of step 2 divide operation
7. Q ← q1 + q2
8. R ← r1 + r2
9. if (R < r2) | (R ≥ Dv) then # (see Note 2)
   Q ← Q + 1 # increment quotient
   R ← R - Dv # decrement rem’der

Assembler Code:

```
# Dh in r4, Dl in r5
# Dv in r6
divweu  r3,r4,r6     # q1
divwu   r7,r5,r6     # q2
mullw   r8,r3,r6     # -r1 = q1 * Dv
mullw   r0,r7,r6     # q2 * Dv
subf    r10,r0,r5    # r2 = Dl - (q2 * Dv)
add     r3,r3,r7     # Q = q1 + q2
subf    r4,r8,r10    # R = r1 + r2
cmplw   r4,r10       # R < r2 ?
blt     *+12         # must adjust Q and R if yes
cmplw   r4,r6        # R ≥ Dv ?
blt     *+12         # must adjust Q and R if yes
addi    r3,r3,1      # Q = Q + 1
subf    r4,r6,r4     # R = R - Dv
# Quotient in r3
# Remainder in r4
```

Notes:

1. The remainder is Dh || $^{32}0$ - (q1 × Dv). Because the remainder must be less than Dv and Dv < 2^{32}, the remainder is representable in 32 bits. Because the low-order 32 bits of Dh || $^{32}0$ are 0s, the remainder is therefore equal to the low-order 32 bits of -(q1 × Dv). Thus assigning -(q1 × Dv) to r1 yields the correct remainder.
2. R is less than r2 (and also less than r1) if and only if the addition at step 6 carried out of 32 bits — i.e., if and only if the correct sum could not be represented in 32 bits — in which case the correct sum is necessarily greater than Dv.
3. For additional information see the book *Hacker's Delight*, by Henry S. Warren, Jr., as potentially amended at the web site http://www.hackersdelight.org.

3.3.9.1 64-bit Fixed-Point Arithmetic Instructions [Category: 64-Bit]

Multiply Low Doubleword *XO-form*

mulld RT,RA,RB (OE=0 Rc=0)
mulld. RT,RA,RB (OE=0 Rc=1)
mulldo RT,RA,RB (OE=1 Rc=0)
mulldo. RT,RA,RB (OE=1 Rc=1)

31	RT	RA	RB	OE	233	Rc
0	6	11	16	21	22	31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{64:127}$

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Programming Note

The XO-form *Multiply* instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

Multiply High Doubleword Unsigned *XO-form*

mulhdu RT,RA,RB (Rc=0)
mulhdu. RT,RA,RB (Rc=1)

31	RT	RA	RB	/	9	Rc
0	6	11	16	21	22	31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:

CR0 (if Rc=1)

Multiply High Doubleword *XO-form*

mulhd RT,RA,RB (Rc=0)
mulhd. RT,RA,RB (Rc=1)

31	RT	RA	RB	/	73	Rc
0	6	11	16	21	22	31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 (if Rc=1)

Divide Doubleword *XO-form*

divd	RT,RA,RB	(OE=0 Rc=0)
divd.	RT,RA,RB	(OE=0 Rc=1)
divdo	RT,RA,RB	(OE=1 Rc=0)
divdo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	489	Rc
0	6	11	16	21	22	31

$dividend_{0:63}$ ← (RA)
$divisor_{0:63}$ ← (RB)
RT ← dividend ÷ divisor

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000_0000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The 64-bit signed remainder of dividing (RA) by (RB) can be computed as follows, except in the case that (RA) = -2^{63} and (RB) = -1.

```
divd   RT,RA,RB      # RT = quotient
mulld  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Doubleword Unsigned *XO-form*

divdu	RT,RA,RB	(OE=0 Rc=0)
divdu.	RT,RA,RB	(OE=0 Rc=1)
divduo	RT,RA,RB	(OE=1 Rc=0)
divduo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	457	Rc
0	6	11	16	21	22	31

$dividend_{0:63}$ ← (RA)
$divisor_{0:63}$ ← (RB)
RT ← dividend ÷ divisor

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The 64-bit unsigned remainder of dividing (RA) by (RB) can be computed as follows.

```
divdu  RT,RA,RB      # RT = quotient
mulld  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Doubleword Extended *XO-form*

divde RT,RA,RB (OE=0 Rc=0)
divde. RT,RA,RB (OE=0 Rc=1)
divdeo RT,RA,RB (OE=1 Rc=0)
divdeo. RT,RA,RB (OE=1 Rc=1)
[Category: Server]
[Category: Embedded.Phased-In]

31	RT	RA	RB	OE	425	Rc
0	6	11	16	21	22	31

$dividend_{0:127} \leftarrow (RA) \,||\, {}^{64}0$
$divisor_{0:63} \leftarrow (RB)$
$RT \leftarrow dividend \div divisor$

The 128-bit dividend is (RA) || $^{64}0$. The 64-bit divisor is (RB). If the quotient can be represented in 64 bits, it is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Divide Doubleword Extended Unsigned *XO-form*

divdeu RT,RA,RB (OE=0 Rc=0)
divdeu. RT,RA,RB (OE=0 Rc=1)
divdeuo RT,RA,RB (OE=1 Rc=0)
divdeuo. RT,RA,RB (OE=1 Rc=1)
[Category: Server]
[Category: Embedded.Phased-In]

31	RT	RA	RB	OE	393	Rc
0	6	11	16	21	22	31

$dividend_{0:127} \leftarrow (RA) \,||\, {}^{64}0$
$divisor_{0:63} \leftarrow (RB)$
$RT \leftarrow dividend \div divisor$

The 128-bit dividend is (RA) || $^{64}0$. The 64-bit divisor is (RB). If the quotient can be represented in 64 bits, it is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r <$ divisor.

If (RA) ≥ (RB), or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 (if Rc=1)
SO OV (if OE=1)

Programming Note

Unsigned long division of a 128-bit dividend contained in two 64-bit registers by a 64-bit divisor can be accomplished using the technique described in the Programming Note with the ***divweu*** instruction description: ***divd[e]u*** would be used instead of ***divw[e]u*** (and ***cmpld*** instead of ***cmplw***, etc.).

3.3.10 Fixed-Point Compare Instructions

The fixed-point *Compare* instructions compare the contents of register RA with (1) the sign-extended value of the SI field, (2) the zero-extended value of the UI field, or (3) the contents of register RB. The comparison is signed for ***cmpi*** and ***cmp***, and unsigned for ***cmpli*** and ***cmpl***.

The L field controls whether the operands are treated as 64-bit or 32-bit quantities, as follows:

L	Operand length
0	32-bit operands
1	64-bit operands

L=1 is part of Category: 64-Bit.

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

The *Compare* instructions set one bit in the leftmost three bits of the designated CR field to 1, and the other two to 0. XER_{SO} is copied to bit 3 of the designated CR field.

The CR field is set as follows

.

Bit	Name	Description
0	LT	(RA) < SI or (RB) (signed comparison) (RA) $<^u$ UI or (RB) (unsigned comparison)
1	GT	(RA) > SI or (RB) (signed comparison) (RA) $>^u$ UI or (RB) (unsigned comparison)
2	EQ	(RA) = SI, UI, or (RB)
3	SO	Summary Overflow from the XER

Extended mnemonics for compares

A set of extended mnemonics is provided so that compares can be coded with the operand length as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Compare* instructions. See Appendix E for additional extended mnemonics.

Compare Immediate — *D-form*

cmpi BF,L,RA,SI

11	BF	/	L	RA	SI
0	6	9	10	11	16 31

```
if L = 0 then a ← EXTS((RA)32:63)
         else a ← (RA)
if      a < EXTS(SI) then c ← 0b100
else if a > EXTS(SI) then c ← 0b010
else                      c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ sign-extended to 64 bits if L=0) are compared with the sign-extended value of the SI field, treating the operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Extended Mnemonics:

Examples of extended mnemonics for Compare Immediate:

Extended:		Equivalent to:	
cmpdi	Rx,value	cmpi	0,1,Rx,value
cmpwi	cr3,Rx,value	cmpi	3,0,Rx,value

Compare — *X-form*

cmp BF,L,RA,RB

31	BF	/	L	RA	RB	0	/
0	6	9	10	11	16	21	31

```
if L = 0 then a ← EXTS((RA)32:63)
                b ← EXTS((RB)32:63)
           else a ← (RA)
                b ← (RB)
if      a < b then c ← 0b100
else if a > b then c ← 0b010
else               c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

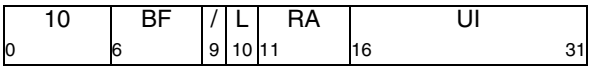
Extended Mnemonics:

Examples of extended mnemonics for Compare:

Extended:		Equivalent to:	
cmpd	Rx,Ry	cmp	0,1,Rx,Ry
cmpw	cr3,Rx,Ry	cmp	3,0,Rx,Ry

Compare Logical Immediate *D-form*

cmpli BF,L,RA,UI




```
if L = 0 then a ← 32 0 || (RA)32:63
         else a ← (RA)
if      a <u (48 0 || UI) then c ← 0b100
else if a >u (48 0 || UI) then c ← 0b010
else                           c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ zero-extended to 64 bits if L=0) are compared with $^{48}0$ || UI, treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

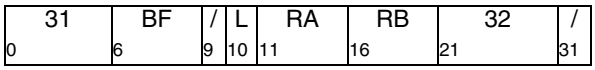
Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical Immediate*:

Extended:		Equivalent to:	
cmpldi	Rx,value	cmpli	0,1,Rx,value
cmplwi	cr3,Rx,value	cmpli	3,0,Rx,value

Compare Logical *X-form*

cmpl BF,L,RA,RB




```
if L = 0 then a ← 32 0 || (RA)32:63
              b ← 32 0 || (RB)32:63
         else a ← (RA)
              b ← (RB)
if      a <u b then c ← 0b100
else if a >u b then c ← 0b010
else                c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical*:

Extended:		Equivalent to:	
cmpld	Rx,Ry	cmpl	0,1,Rx,Ry
cmplw	cr3,Rx,Ry	cmpl	3,0,Rx,Ry

3.3.11 Fixed-Point Trap Instructions

The *Trap* instructions are provided to test for a specified set of conditions. If any of the conditions tested by a *Trap* instruction are met, the system trap handler is invoked. If none of the tested conditions are met, instruction execution continues normally.

The contents of register RA are compared with either the sign-extended value of the SI field or the contents of register RB, depending on the *Trap* instruction. For ***tdi*** and ***td***, the entire contents of RA (and RB) participate in the comparison; for ***twi*** and ***tw***, only the contents of the low-order 32 bits of RA (and RB) participate in the comparison.

This comparison results in five conditions which are ANDed with TO. If the result is not 0 the system trap handler is invoked. These conditions are as follows.

TO Bit	ANDed with Condition
0	Less Than, using signed comparison
1	Greater Than, using signed comparison
2	Equal
3	Less Than, using unsigned comparison
4	Greater Than, using unsigned comparison

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the Trap instructions. See Appendix E for additional extended mnemonics.

Trap Word Immediate ***D-form***

twi TO,RA,SI

3	TO	RA	SI
0	6	11	16 31

```
a ← EXTS((RA)32:63)
if (a < EXTS(SI)) & TO0  then TRAP
if (a > EXTS(SI)) & TO1  then TRAP
if (a = EXTS(SI)) & TO2  then TRAP
if (a <u EXTS(SI)) & TO3 then TRAP
if (a >u EXTS(SI)) & TO4 then TRAP
```

The contents of $RA_{32:63}$ are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Word Immediate*:

Extended:		Equivalent to:	
twgti	Rx,value	twi	8,Rx,value
twllei	Rx,value	twi	6,Rx,value

Trap Word ***X-form***

tw TO,RA,RB

31	TO	RA	RB	4	/
0	6	11	16	21	31

```
a ← EXTS((RA)32:63)
b ← EXTS((RB)32:63)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of $RA_{32:63}$ are compared with the contents of $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

Extended Mnemonics:

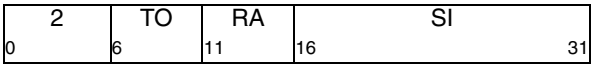
Examples of extended mnemonics for *Trap Word*:

Extended:		Equivalent to:	
tweq	Rx,Ry	tw	4,Rx,Ry
twlge	Rx,Ry	tw	5,Rx,Ry
trap		tw	31,0,0

3.3.11.1 64-bit Fixed-Point Trap Instructions [Category: 64-Bit]

Trap Doubleword Immediate *D-form*

tdi TO,RA,SI




```
a ← (RA)
b ← EXTS(SI)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4  then TRAP
```

The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

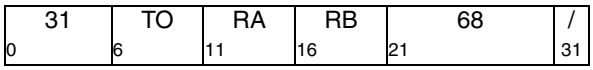
Extended Mnemonics:

Examples of extended mnemonics for *Trap Doubleword Immediate*:

Extended:		Equivalent to:	
tdlti	Rx,value	tdi	16,Rx,value
tdnei	Rx,value	tdi	24,Rx,value

Trap Doubleword *X-form*

td TO,RA,RB




```
a ← (RA)
b ← (RB)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4  then TRAP
```

The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

Extended Mnemonics:

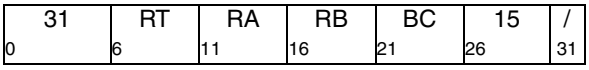
Examples of extended mnemonics for *Trap Doubleword*:

Extended:		Equivalent to:	
tdge	Rx,Ry	td	12,Rx,Ry

3.3.12 Fixed-Point Select [Category: Phased-In (sV2.06)]

Integer Select *A-form*

isel RT,RA,RB,BC




```
if RA=0 then a ←0 else a ← (RA)
if CRBC+32=1 then RT ← a
else              RT ← (RB)
```

If the contents of bit BC+32 of the Condition Register are equal to 1, then the contents of register RA (or 0) are placed into register RT. Otherwise, the contents of register RB are placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Integer Select*:

Extended:		Equivalent to:	
isellt	Rx,Ry,Rz	isel	Rx,Ry,Rz,0
iselgt	Rx,Ry,Rz	isel	Rx,Ry,Rz,1
iseleq	Rx,Ry,Rz	isel	Rx,Ry,Rz,1

3.3.13 Fixed-Point Logical Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands.

The X-form Logical instructions with Rc=1, and the D-form *Logical* instructions ***andi.*** and ***andis.***, set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 65. The Logical instructions do not change the SO, OV, and CA bits in the XER.

Extended mnemonics for logical operations

Extended mnemonics are provided that generate two different types of “no-ops” (instructions that do nothing). The first type is the preferred form, which is optimized to minimize its use of the processor's execution resources. This form is based on the *OR Immediate* instruction. The second type is the executed form, which is intended to consume the same amount of the processor's execution resources as if it were not a no-op. This form is based on the *XOR Immediate* instruction. (There are also no-ops that have other uses, such as affecting program priority, for which extended mnemonics have not been defined.)

Extended mnemonics are provided that use the *OR* and *NOR* instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

See Appendix E, “Assembler Extended Mnemonics” on page 709 for additional extended mnemonics.

Programming Note

Warning: Some forms of no-op may have side effects such as affecting program priority. Programmers should use the preferred no-op unless the side effects of some other form of no-op are intended.

AND Immediate *D-form*

andi. RA,RS,UI

28	RS	RA	UI
0	6	11	16 31

RA ← (RS) & ($^{48}0$ || UI)

The contents of register RS are ANDed with $^{48}0$ || UI and the result is placed into register RA.

Special Registers Altered:
CR0

AND Immediate Shifted *D-form*

andis. RA,RS,UI

29	RS	RA	UI
0	6	11	16 31

RA ← (RS) & ($^{32}0$ || UI || $^{16}0$)

The contents of register RS are ANDed with $^{32}0$ || UI || $^{16}0$ and the result is placed into register RA.

Special Registers Altered:
CR0

OR Immediate *D-form*

ori RA,RS,UI

24	RS	RA	UI
0	6	11	16 31

RA ← (RS) | ($^{48}0$ || UI)

The contents of register RS are ORed with $^{48}0$ || UI and the result is placed into register RA.

The preferred “no-op” (an instruction that does nothing) is:

```
ori    0,0,0
```

Special Registers Altered:
None

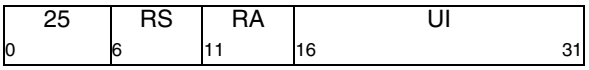
Extended Mnemonics:

Example of extended mnemonics for *OR Immediate*:

Extended:	Equivalent to:
no-op	ori 0,0,0

OR Immediate Shifted *D-form*

oris RA,RS,UI



RA ← (RS) | (^{32}0 || UI || ^{16}0)

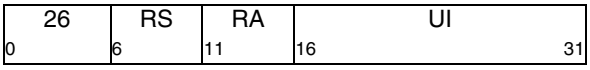
The contents of register RS are ORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:

None

XOR Immediate *D-form*

xori RA,RS,UI



RA ← (RS) XOR (^{48}0 || UI)

The contents of register RS are XORed with ^{48}0 || UI and the result is placed into register RA.

The executed form of a “no-op” (an instruction that does nothing, but consumes execution resources nevertheless) is:

```
xori   0,0,0
```

Special Registers Altered:

None

Extended Mnemonics:

Example of extended mnemonics for *XOR Immediate*:

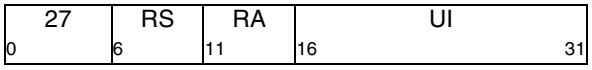
Extended:	Equivalent to:
xnop	xori 0,0,0

Programming Note

The executed form of no-op should be used only when the intent is to alter the timing of a program.

XOR Immediate Shifted *D-form*

xoris RA,RS,UI



RA ← (RS) XOR (^{32}0 || UI || ^{16}0)

The contents of register RS are XORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:

None

AND *X-form*

and RA,RS,RB (Rc=0)
and. RA,RS,RB (Rc=1)

31	RS	RA	RB	28	Rc
0	6	11	16	21	31

RA ← (RS) & (RB)

The contents of register RS are ANDed with the contents of register RB and the result is placed into register RA.

<S> Some forms of ***and*** Rx, Rx, Rx may provide special functions, see Section 9.3 of Book III-S.

Special Registers Altered:
CR0 (if Rc=1)

XOR *X-form*

xor RA,RS,RB (Rc=0)
xor. RA,RS,RB (Rc=1)

31	RS	RA	RB	316	Rc
0	6	11	16	21	31

RA ← (RS) ⊕ (RB)

The contents of register RS are XORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

NAND *X-form*

nand RA,RS,RB (Rc=0)
nand. RA,RS,RB (Rc=1)

31	RS	RA	RB	476	Rc
0	6	11	16	21	31

RA ← ¬((RS) & (RB))

The contents of register RS are ANDed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Programming Note

nand or ***nor*** with RS=RB can be used to obtain the one's complement.

OR *X-form*

or RA,RS,RB (Rc=0)
or. RA,RS,RB (Rc=1)

31	RS	RA	RB	444	Rc
0	6	11	16	21	31

RA ← (RS) | (RB)

The contents of register RS are ORed with the contents of register RB and the result is placed into register RA.

Some forms of *or Rx,Rx,Rx* provide special functions. See Section 3.2, ""or" Instruction", in Book II.

Special Registers Altered:
CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *OR*:

Extended:		**Equivalent to:**	
mr	Rx,Ry	or	Rx,Ry,Ry

NOR *X-form*

nor RA,RS,RB (Rc=0)
nor. RA,RS,RB (Rc=1)

31	RS	RA	RB	124	Rc
0	6	11	16	21	31

```
RA ← ¬((RS) | (RB))
```

The contents of register RS are ORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *NOR*:

Extended:		Equivalent to:	
not	Rx,Ry	nor	Rx,Ry,Ry

Equivalent *X-form*

eqv RA,RS,RB (Rc=0)
eqv. RA,RS,RB (Rc=1)

31	RS	RA	RB	284	Rc
0	6	11	16	21	31

```
RA ← (RS) ≡ (RB)
```

The contents of register RS are XORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

AND with Complement *X-form*

andc RA,RS,RB (Rc=0)
andc. RA,RS,RB (Rc=1)

31	RS	RA	RB	60	Rc
0	6	11	16	21	31

```
RA ← (RS) & ¬(RB)
```

The contents of register RS are ANDed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

OR with Complement *X-form*

orc RA,RS,RB (Rc=0)
orc. RA,RS,RB (Rc=1)

31	RS	RA	RB	412	Rc
0	6	11	16	21	31

```
RA ← (RS) | ¬(RB)
```

The contents of register RS are ORed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Extend Sign Byte *X-form*

extsb RA,RS (Rc=0)
extsb. RA,RS (Rc=1)

31	RS	RA	///	954	Rc
0	6	11	16	21	31

$s \leftarrow (RS)_{56}$
$RA_{56:63} \leftarrow (RS)_{56:63}$
$RA_{0:55} \leftarrow {}^{56}s$

$(RS)_{56:63}$ are placed into $RA_{56:63}$. $RA_{0:55}$ are filled with a copy of $(RS)_{56}$.

Special Registers Altered:
CR0 (if Rc=1)

Extend Sign Halfword *X-form*

extsh RA,RS (Rc=0)
extsh. RA,RS (Rc=1)

31	RS	RA	///	922	Rc
0	6	11	16	21	31

$s \leftarrow (RS)_{48}$
$RA_{48:63} \leftarrow (RS)_{48:63}$
$RA_{0:47} \leftarrow {}^{48}s$

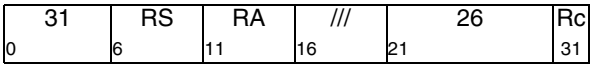
$(RS)_{48:63}$ are placed into $RA_{48:63}$. $RA_{0:47}$ are filled with a copy of $(RS)_{48}$.

Special Registers Altered:
CR0 (if Rc=1)

Count Leading Zeros Word *X-form*

cntlzw RA,RS (Rc=0)

cntlzw. RA,RS (Rc=1)




```
n ← 32
do while n < 64
   if (RS)n = 1 then leave
   n ← n + 1
RA ← n - 32
```

A count of the number of consecutive zero bits starting at bit 32 of register RS is placed into register RA. This number ranges from 0 to 32, inclusive.

If Rc=1, CR Field 0 is set to reflect the result.

Special Registers Altered:

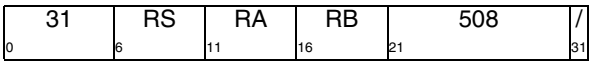
CR0 (if Rc=1)

Programming Note

For both *Count Leading Zeros* instructions, if Rc=1 then LT is set to 0 in CR Field 0.

Compare Bytes *X-form*

cmpb RA,RS,RB




```
do n = 0 to 7
   if RS8×n:8×n+7 = (RB)8×n:8×n+7 then
      RA8×n:8×n+7 ← 81
   else
      RA8×n:8×n+7 ← 80
```

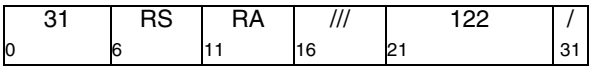
Each byte of the contents of register RS is compared to each corresponding byte of the contents in register RB. If they are equal, the corresponding byte in RA is set to 0xFF. Otherwise the corresponding byte in RA is set to 0x00.

Special Registers Altered:

None

Population Count Bytes *X-form*

popcntb RA, RS




```
do i = 0 to 7
   n ← 0
   do j = 0 to 7
      if (RS)(i×8)+j = 1 then
         n ← n+1
   RA(i×8):(i×8)+7 ← n
```

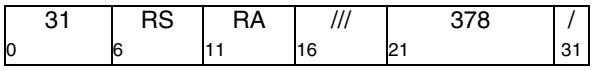
A count of the number of one bits in each byte of register RS is placed into the corresponding byte of register RA. This number ranges from 0 to 8, inclusive.

Special Registers Altered:

None

Population Count Words *X-form*

popcntw RA, RS
[Category: Server]
[Category: Embedded.Phased-In]




```
do i = 0 to 1
   n ← 0
   do j = 0 to 31
      if (RS)(i×32)+j = 1 then
         n ← n+1
   RA(i×32):(i×32)+31 ← n
```

A count of the number of one bits in each word of register RS is placed into the corresponding word of register RA. This number ranges from 0 to 32, inclusive.

Special Registers Altered:

None

Parity Doubleword *X-form*

prtyd RA,RS
[Category: 64-bit]

31	RS	RA	///	186	/
0	6	11	16	21	31

```
s ← 0
do i = 0 to 7
   s ← s ⊕ (RS)ix8+7
RA ← 63 0 || s
```

The least significant bit in each byte of the contents of register RS is examined. If there is an odd number of one bits the value 1 is placed into register RA; otherwise the value 0 is placed into register RA.

Special Registers Altered:

None

Parity Word *X-form*

prtyw RA,RS

31	RS	RA	///	154	/
0	6	11	16	21	31

```
s ← 0
t ← 0
do i = 0 to 3
   s ← s ⊕ (RS)ix8+7
do i = 4 to 7
   t ← t ⊕ (RS)ix8+7
RA0:31 ← 31 0 || s
RA32:63 ← 31 0 || t
```

The least significant bit in each byte of $(RS)_{0:31}$ is examined. If there is an odd number of one bits the value 1 is placed into $RA_{0:31}$; otherwise the value 0 is placed into $RA_{0:31}$. The least significant bit in each byte of $(RS)_{32:63}$ is examined. If there is an odd number of one bits the value 1 is placed into $RA_{32:63}$; otherwise the value 0 is placed into $RA_{32:63}$.

Special Registers Altered:

None

Programming Note

The *Parity* instructions are designed to be used in conjunction with the *Population Count* instruction to compute the parity of words or a doubleword. The parity of the upper and lower words in (RS) can be computed as follows.

```
popcntb  RA, RS
prtyw    RA, RA
```

The parity of (RS) can be computed as follows.

```
popcntb  RA, RS
prtyd    RA, RA
```

3.3.13.1 64-bit Fixed-Point Logical Instructions [Category: 64-Bit]

Extend Sign Word *X-form*

extsw RA,RS (Rc=0)
extsw. RA,RS (Rc=1)

31	RS	RA	///	986	Rc
0	6	11	16	21	31

```
s ← (RS)32
RA32:63 ← (RS)32:63
RA0:31 ← 32s
```

$(RS)_{32:63}$ are placed into $RA_{32:63}$. $RA_{0:31}$ are filled with a copy of $(RS)_{32}$.

Special Registers Altered:
CR0 (if Rc=1)

Count Leading Zeros Doubleword X-form

cntlzd RA,RS (Rc=0)
cntlzd. RA,RS (Rc=1)

31	RS	RA	///	58	Rc
0	6	11	16	21	31

```
n ← 0
do while n < 64
  if (RS)n = 1 then leave
  n ← n + 1
RA ← n
```

A count of the number of consecutive zero bits starting at bit 0 of register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

If Rc=1, CR Field 0 is set to reflect the result.

Special Registers Altered:
CR0 (if Rc=1)

Population Count Doubleword X-form

popcntd RA, RS
[Category: Server.64-bit]
[Category: Embedded.64-bit.Phased-In]

31	RS	RA	///	506	/
0	6	11	16	21	31

```
n ← 0
do i = 0 to 63
   if (RS)i = 1 then
   n ← n+1
RA ← n
```

A count of the number of one bits in register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None

Bit Permute Doubleword *X-form*

bpermd RA,RS,RB
[Category: Embedded.Phased-in, Server]

31	RS	RA	RB	252	/
0	6	11	16	21	31

```
For i = 0 to 7
   index ← (RS)8*i:8*i+7
   If index < 64
      then permi ← (RB)index
      else permi ← 0
RA ← 560 || perm0:7
```

Eight permuted bits are produced. For each permuted bit i where i ranges from 0 to 7 and for each byte i of RS, do the following.

If byte i of RS is less than 64, permuted bit i is set to the bit of RB specified by byte i of RS; otherwise permuted bit i is set to 0.

The permuted bits are placed in the least-significant byte of RA, and the remaining bits are filled with 0s.

Special Registers Altered:

None

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 63 permits the permuted bits to be selected from a 128-bit quantity, using a single index register. For example, assume that the 128-bit quantity Q, from which the permuted bits are to be selected, is in registers r2 (high-order 64 bits of Q) and r3 (low-order 64 bits of Q), that the index values are in register r1, with each byte of r1 containing a value in the range 0:127, and that each byte of register r4 contains the value 64. The following code sequence selects eight permuted bits from Q and places them into the low-order byte of r6.

```
bpermd    r6,r1,r2     # select from high-
                         order half of Q
xor       r0,r1,r4     # adjust index values
bpermd    r5,r0,r3     # select from low-
                         order half of Q
or        r6,r6,r5     # merge the two
                         selections
```

3.3.14 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Facility performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted $rotate_{64}$ or $ROTL_{64}$, the value rotated is the given 64-bit value. The $rotate_{64}$ operation is used to rotate a given 64-bit quantity.

For the second type, denoted $rotate_{32}$ or $ROTL_{32}$, the value rotated consists of two copies of bits 32:63 of the given 64-bit value, one copy in bits 0:31 and the other in bits 32:63. The $rotate_{32}$ operation is used to rotate a given 32-bit quantity.

The *Rotate* and *Shift* instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, *mstart*, through and including a stop bit, *mstop*, and 0-bits elsewhere. The values of *mstart* and *mstop* range from 0 to 63. If *mstart* > *mstop*, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```
if mstart ≤ mstop then
   mask_mstart:mstop = ones
   mask_all other bits = zeros
else
   mask_mstart:63 = ones
   mask_0:mstop = ones
   mask_all other bits = zeros
```

There is no way to specify an all-zero mask.

For instructions that use the $rotate_{32}$ operation, the mask start and stop positions are always in the low-order 32 bits of the mask.

The use of the mask is described in following sections.

The *Rotate* and *Shift* instructions with Rc=1 set the first three bits of CR field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 65. *Rotate* and *Shift* instructions do not change the OV and SO bits. *Rotate* and *Shift* instructions, except algebraic right shifts, do not change the CA bit.

Extended mnemonics for rotates and shifts

The *Rotate* and *Shift* instructions, while powerful, can be complicated to code (they have up to five operands). A set of extended mnemonics is provided that allow simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and performing simple rotates and shifts. Some of these are shown as examples with the *Rotate* instructions. See Appendix E, “Assembler Extended Mnemonics” on page 709 for additional extended mnemonics.

3.3.14.1 Fixed-Point Rotate Instructions

These instructions rotate the contents of a register. The result of the rotation is

- inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged); or
- ANDed with a mask before being placed into the target register.

The *Rotate Left* instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64 - n, where n is the number of bits by which to rotate right. They allow right-rotation of the contents of the low-order 32 bits of a register to be performed (in concept) by a left-rotation of 32 - n, where n is the number of bits by which to rotate right.

Rotate Left Word Immediate then AND with Mask — *M-form*

rlwinm RA,RS,SH,MB,ME (Rc=0)
rlwinm. RA,RS,SH,MB,ME (Rc=1)

21	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are $rotated_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Word Immediate then AND with Mask*:

Extended:		Equivalent to:	
extlwi	Rx,Ry,n,b	rlwinm	Rx,Ry,b,0,n-1
srwi	Rx,Ry,n	rlwinm	Rx,Ry,32-n,n,31
clrrwi	Rx,Ry,n	rlwinm	Rx,Ry,0,0,31-n

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

rlwinm can be used to extract an n-bit field that starts at bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b+n, MB=32-n, and ME=31. It can be used to extract an n-bit field that starts at bit position b in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b, MB = 0, and ME=n-1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by n bits, by setting SH=n (32-n), MB=0, and ME=31. It can be used to shift the contents of the low-order 32 bits of a register right by n bits, by setting SH=32-n, MB=n, and ME=31. It can be used to clear the high-order b bits of the low-order 32 bits of the contents of a register and then shift the result left by n bits, by setting SH=n, MB=b-n, and ME=31-n. It can be used to clear the low-order n bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31-n.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for all of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.

Rotate Left Word then AND with Mask M-form

rlwnm RA,RS,RB,MB,ME (Rc=0)
rlwnm. RA,RS,RB,MB,ME (Rc=1)

23	RS	RA	RB	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are rotated$_{32}$ left the number of bits specified by (RB)$_{59:63}$. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word then AND with Mask*:

Extended:		Equivalent to:	
rotlw	Rx,Ry,Rz	rlwnm	Rx,Ry,Rz,0,31

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

rlwnm can be used to extract an n-bit field that starts at variable bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting RB$_{59:63}$=b+n, MB=32-n, and ME=31. It can be used to extract an n-bit field that starts at variable bit position b in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting RB$_{59:63}$=b, MB = 0, and ME=n-1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by variable n bits, by setting RB$_{59:63}$=n (32-n), MB=0, and ME=31.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for some of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.

Rotate Left Word Immediate then Mask Insert *M-form*

rlwimi RA,RS,SH,MB,ME (Rc=0)
rlwimi. RA,RS,SH,MB,ME (Rc=1)

20	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word Immediate then Mask Insert*:

Extended:	Equivalent to:
inslwi Rx,Ry,n,b	rlwimi Rx,Ry,32-b,b,b+n-1

Programming Note

Let RAL represent the low-order 32 bits of register RA, with the bits numbered from 0 through 31.

rlwimi can be used to insert an n-bit field that is left-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting SH=32-b, MB=b, and ME=(b+n)-1. It can be used to insert an n-bit field that is right-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting SH=32-(b+n), MB=b, and ME=(b+n)-1.

Extended mnemonics are provided for both of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.

3.3.14.1.1 64-bit Fixed-Point Rotate Instructions [Category: 64-Bit]

Rotate Left Doubleword Immediate then Clear Left *MD-form*

rldicl RA,RS,SH,MB (Rc=0)
rldicl. RA,RS,SH,MB (Rc=1)

30	RS	RA	sh	mb	0	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Left*:

Extended:		Equivalent to:	
extrdi	Rx,Ry,n,b	rldicl	Rx,Ry,b+n,64-n
srdi	Rx,Ry,n	rldicl	Rx,Ry,64-n,n
clrldi	Rx,Ry,n	rldicl	Rx,Ry,0,n

Programming Note

rldicl can be used to extract an n-bit field that starts at bit position b in register RS, right-justified into register RA (clearing the remaining 64-n bits of RA), by setting SH=b+n and MB=64-n. It can be used to rotate the contents of a register left (right) by n bits, by setting SH=n (64-n) and MB=0. It can be used to shift the contents of a register right by n bits, by setting SH=64-n and MB=n. It can be used to clear the high-order n bits of a register, by setting SH=0 and MB=n.

Extended mnemonics are provided for all of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.

Rotate Left Doubleword Immediate then Clear Right *MD-form*

rldicr RA,RS,SH,ME (Rc=0)
rldicr. RA,RS,SH,ME (Rc=1)

30	RS	RA	sh	me	1	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
e ← me5 || me0:4
m ← MASK(0, e)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Right*:

Extended:		Equivalent to:	
extldi	Rx,Ry,n,b	rldicr	Rx,Ry,b,n-1
sldi	Rx,Ry,n	rldicr	Rx,Ry,n,63-n
clrrdi	Rx,Ry,n	rldicr	Rx,Ry,0,63-n

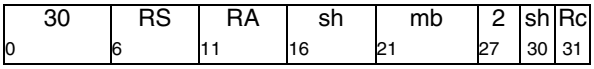
Programming Note

rldicr can be used to extract an n-bit field that starts at bit position b in register RS, left-justified into register RA (clearing the remaining 64-n bits of RA), by setting SH=b and ME=n-1. It can be used to rotate the contents of a register left (right) by n bits, by setting SH=n (64-n) and ME=63. It can be used to shift the contents of a register left by n bits, by setting SH=n and ME=63-n. It can be used to clear the low-order n bits of a register, by setting SH=0 and ME=63-n.

Extended mnemonics are provided for all of these uses (some devolve to ***rldicl***); see Appendix E, “Assembler Extended Mnemonics” on page 709.

Rotate Left Doubleword Immediate then Clear MD-form

rldic RA,RS,SH,MB (Rc=0)
rldic. RA,RS,SH,MB (Rc=1)




```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63 - SH and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Clear*:

Extended:		Equivalent to:	
clrlsldi	Rx,Ry,b,n	rldic	Rx,Ry,n,b - n

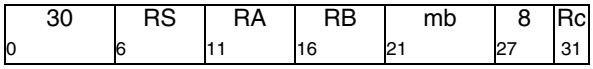
Programming Note

rldic can be used to clear the high-order b bits of the contents of a register and then shift the result left by n bits, by setting SH=n and MB=b - n. It can be used to clear the high-order n bits of a register, by setting SH=0 and MB=n.

Extended mnemonics are provided for both of these uses (the second devolves to ***rldicl***); see Appendix E, “Assembler Extended Mnemonics” on page 709.

Rotate Left Doubleword then Clear Left MDS-form

rldcl RA,RS,RB,MB (Rc=0)
rldcl. RA,RS,RB,MB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword then Clear Left*:

Extended:		Equivalent to:	
rotld	Rx,Ry,Rz	rldcl	Rx,Ry,Rz,0

Programming Note

rldcl can be used to extract an n-bit field that starts at variable bit position b in register RS, right-justified into register RA (clearing the remaining 64 - n bits of RA), by setting RB$_{58:63}$=b+n and MB=64 - n. It can be used to rotate the contents of a register left (right) by variable n bits, by setting RB$_{58:63}$=n (64 - n) and MB=0.

Extended mnemonics are provided for some of these uses; see Appendix E, “Assembler Extended Mnemonics” on page 709.

Rotate Left Doubleword then Clear Right *MDS-form*

rldcr RA,RS,RB,ME (Rc=0)
rldcr. RA,RS,RB,ME (Rc=1)

30	RS	RA	RB	me	9	Rc
0	6	11	16	21	27	31

```
n ← (RB)58:63
r ← ROTL64((RS), n)
e ← me5 || me0:4
m ← MASK(0, e)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Programming Note

rldcr can be used to extract an n-bit field that starts at variable bit position b in register RS, left-justified into register RA (clearing the remaining 64-n bits of RA), by setting RB$_{58:63}$=b and ME=n-1. It can be used to rotate the contents of a register left (right) by variable n bits, by setting RB$_{58:63}$=n (64-n) and ME=63.

Extended mnemonics are provided for some of these uses (some devolve to ***rldcl***); see Appendix E, “Assembler Extended Mnemonics” on page 709.

Rotate Left Doubleword Immediate then Mask Insert *MD-form*

rldimi RA,RS,SH,MB (Rc=0)
rldimi. RA,RS,SH,MB (Rc=1)

30	RS	RA	sh	mb	3	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63-SH and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Mask Insert*:

Extended:	**Equivalent to:**
insrdi Rx,Ry,n,b	rldimi Rx,Ry,64-(b+n),b

Programming Note

rldimi can be used to insert an n-bit field that is right-justified in register RS, into register RA starting at bit position b, by setting SH=64-(b+n) and MB=b.

An extended mnemonic is provided for this use; see Appendix E, “Assembler Extended Mnemonics” on page 709.

3.3.14.2 Fixed-Point Shift Instructions

The instructions in this section perform left and right shifts.

Extended mnemonics for shifts

Immediate-form logical (unsigned) shift operations are obtained by specifying appropriate masks and shift values for certain *Rotate* instructions. A set of extended mnemonics is provided to make coding of such shifts simpler and easier to understand. Some of these are shown as examples with the *Rotate* instructions. See Appendix E, "Assembler Extended Mnemonics" on page 709 for additional extended mnemonics.

Programming Note

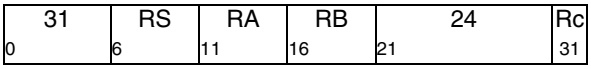
Any Shift Right Algebraic instruction, followed by ***addze***, can be used to divide quickly by 2^n. The setting of the CA bit by the Shift Right Algebraic instructions is independent of mode.

Programming Note

Multiple-precision shifts can be programmed as shown in Section F.1, "Multiple-Precision Shifts" on page 723.

Shift Left Word — *X-form*

slw RA,RS,RB (Rc=0)
slw. RA,RS,RB (Rc=1)




```
n ← (RB)59:63
r ← ROTL32((RS)32:63, n)
if (RB)58 = 0 then
     m ← MASK(32, 63-n)
else m ← 640
RA ← r & m
```

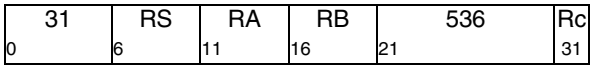
The contents of the low-order 32 bits of register RS are shifted left the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Word — *X-form*

srw RA,RS,RB (Rc=0)
srw. RA,RS,RB (Rc=1)




```
n ← (RB)59:63
r ← ROTL32((RS)32:63, 64-n)
if (RB)58 = 0 then
    m ← MASK(n+32, 63)
else m ← 640
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Algebraic Word Immediate *X-form*

srawi RA,RS,SH (Rc=0)
srawi. RA,RS,SH (Rc=1)

31	RS	RA	SH	824	Rc
0	6	11	16	21	31

```
n ← SH
r ← ROTL32((RS)32:63, 64-n)
m ← MASK(n+32, 63)
s ← (RS)32
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)32:63≠0)
```

The contents of the low-order 32 bits of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA is set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive EXTS($(RS)_{32:63}$), and CA to be set to 0.

Special Registers Altered:

CA
CR0 (if Rc=1)

Shift Right Algebraic Word *X-form*

sraw RA,RS,RB (Rc=0)
sraw. RA,RS,RB (Rc=1)

31	RS	RA	RB	792	Rc
0	6	11	16	21	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, 64-n)
if (RB)58 = 0 then
    m ← MASK(n+32, 63)
else m ← 640
s ← (RS)32
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)32:63≠0)
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA is set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive EXTS($(RS)_{32:63}$), and CA to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive the sign bit of $(RS)_{32:63}$.

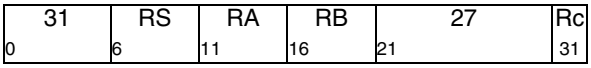
Special Registers Altered:

CA
CR0 (if Rc=1)

3.3.14.2.1 64-bit Fixed-Point Shift Instructions [Category: 64-Bit]

Shift Left Doubleword *X-form*

sld RA,RS,RB (Rc=0)
sld. RA,RS,RB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
if (RB)57 = 0 then
     m ← MASK(0, 63-n)
else m ← 640
RA ← r & m
```

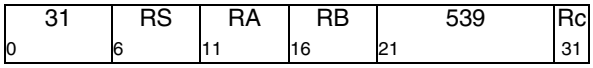
The contents of register RS are shifted left the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Doubleword *X-form*

srd RA,RS,RB (Rc=0)
srd. RA,RS,RB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 640
RA ← r & m
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Algebraic Doubleword Immediate XS-form

sradi RA,RS,SH (Rc=0)
sradi. RA,RS,SH (Rc=1)

31	RS	RA	sh	413	sh	Rc
0	6	11	16	21	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), 64-n)
m ← MASK(n, 63)
s ← (RS)0
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)≠0)
```

The contents of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0.

Special Registers Altered:
CA
CR0 (if Rc=1)

Shift Right Algebraic Doubleword X-form

srad RA,RS,RB (Rc=0)
srad. RA,RS,RB (Rc=1)

31	RS	RA	RB	794	Rc
0	6	11	16	21	31

```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 640
s ← (RS)0
RA ← r&m | (64s)&¬m
CA ← s & ((r&¬m)≠0)
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0. Shift amounts from 64 to 127 give a result of 64 sign bits in RA, and cause CA to receive the sign bit of (RS).

Special Registers Altered:
CA
CR0 (if Rc=1)

3.3.15 Binary Coded Decimal (BCD) Assist Instructions [Category: Embedded.Phased-in, Server]

The *Binary Coded Decimal Assist* instructions operate on Binary Coded Decimal operands (***cbcdtd*** and ***addg6s***) and Decimal Floating-Point operands (***cdtbcd***) See Chapter 6. for additional information.

Convert Declets To Binary Coded Decimal X-form

cdtbcd RA, RS

31	RS	RA	///	282	/
0	6	11	16	21	31

```
do i = 0 to 1
  n ← i x 32
  RA[n+0:n+7] ← 0
  RA[n+8:n+19] ← DPD_TO_BCD( (RS)[n+12:n+21] )
  RA[n+20:n+31] ← DPD_TO_BCD( (RS)[n+22:n+31] )
```

The low-order 20 bits of each word of register RS contain two declets which are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the corresponding word in RA. The high-order 8 bits in each word of RA are set to 0.

Special Registers Altered:

None

Convert Binary Coded Decimal To Declets X-form

cbcdtd RA, RS

31	RS	RA	///	314	/
0	6	11	16	21	31

```
do i = 0 to 1
  n ← i x 32
  RA[n+0:n+11] ← 0
  RA[n+12:n+21] ← BCD_TO_DPD( (RS)[n+8:n+19] )
  RA[n+22:n+31] ← BCD_TO_DPD( (RS)[n+20:n+31] )
```

The low-order 24 bits of each word of register RS contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the corresponding word in RA. The high-order 12 bits in each word of RA are set to 0.

If a 4-bit BCD field has a value greater than 9 the results are undefined.

Special Registers Altered:

None

Add and Generate Sixes XO-form

addg6s RT,RA,RB

31	RT	RA	RB	/	74	/
0	6	11	16	21	22	31

```
do i = 0 to 15
   dc_i ← carry_out(RA[4xi:63] + RB[4xi:63])
c ← 4(dc_0) || 4(dc_1) || ... || 4(dc_15)
RT ← (¬c) & 0x6666_6666_6666_6666
```

The contents of register RA are added to the contents of register RB. Sixteen carry bits are produced, one for each carry out of decimal position n (bit position 4xn).

A doubleword is composed from the 16 carry bits, and placed into RT. The doubleword consists of a decimal six (0b0110) in every decimal digit position for which the corresponding carry bit is 0, and a zero (0b0000) in every position for which the corresponding carry bit is 1.

Special Registers Altered:

None

Programming Note

addg6s can be used to add or subtract two BCD operands. In these examples it is assumed that r0 contains 0x666...666. (BCD data formats are described in Section 6.3.)

Addition of the unsigned BCD operand in register RA to the unsigned BCD operand in register RB can be accomplished as follows.

```
add       r1,RA,r0
add       r2,r1,RB
addg6s    RT,r1,RB
subf      RT,RT,r2# RT = RA +BCD RB
```

Subtraction of the unsigned BCD operand in register RA from the unsigned BCD operand in register RB can be accomplished as follows. (In this example it is assumed that RB is not register 0.)

```
addi      r1,RB,1
nor       r2,RA,RA# one's complement of RA
add       r3,r1,r2
addg6s    RT,r1,r2
subf      RT,RT,r3# RT = RB -BCD RA
```

Additional instructions are needed to handle signed BCD operands, and BCD operands that occupy more than one register (e.g., unsigned BCD operands that have more than 16 decimal digits).

3.3.16 Move To/From System Register Instructions

The *Move To Condition Register Fields* instruction has a preferred form; see Section 1.8.1, “Preferred Instruction Forms” on page 22. In the preferred form, the FXM field satisfies the following rule.

- Exactly one bit of the FXM field is set to 1.

Extended mnemonics

Extended mnemonics are provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. An extended mnemonic is provided for the ***mtcrf*** instruction for compatibility with old software (written for a version of the architecture that precedes Version 2.00) that uses it to set the entire Condition Register. Some of these extended mnemonics are shown as examples with the relevant instructions. See Appendix E, “Assembler Extended Mnemonics” on page 709 for additional extended mnemonics.

Move To Special Purpose Register *XFX-form*

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(13): see Book III-S
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      SPR(n) ← (RS)
    else
      SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs". Otherwise, unless the SPR field contains 13 (denoting the AMR<S>), the contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

The AMR (Authority Mask Register) is used for "storage protection" in the Server environment. This use, and operation of ***mtspr*** for the AMR, are described in Book III-S.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
3	00000 00011	DSCR[5]
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR[3]
128	00100 00000	TFHAR[4]
129	00100 00001	TFIAR[4]
130	00100 00010	TEXASR[4]
131	00100 00011	TEXASRU[4]
256	01000 00000	VRSAVE
512	10000 00000	SPEFSCR[2]
769	11000 00001	MMCR2[3]
770	11000 00010	MMCRA[3]
771	11000 00011	PMC1[3]
772	11000 00100	PMC2[3]
773	11000 00101	PMC3[3]
774	11000 00110	PMC4[3]
775	11000 00111	PMC5[3]
776	11000 01000	PMC6[3]
779	11000 01011	MMCR0[3]
800	11001 00000	BESCRS[7]
801	11001 00001	BESCRSU[7]
802	11001 00010	BESCRR[7]
803	11001 00011	BESCRRU[7]
804	11001 00100	EBBHR[7]
805	11001 00101	EBBRR[7]
806	11001 00110	BESCR[7]
808	11001 01000	reserved[6]
809	11001 01001	reserved[6]
810	11001 01010	reserved[6]
811	11001 01011	reserved[6]
815	11001 01111	TAR[3]
896	11100 00000	PPR[7]
898	11100 00010	PPR32

1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 Category: SPE.
3 Category: Server; see Book III-S.
4 Category: Transactional Memory. See Chapter 5 of Book II.
5 Category: Stream.
6 Accesses to these registers are noops; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs"
7 Category: Server; see Book II.

If execution of this instruction is attempted specifying an SPR number that is not shown above, or an SPR number that is shown above but is in a category that is not supported by the implementation, one of the following occurs.

- If spr_0 = 0, the illegal instruction error handler is invoked.
- If spr_0 = 1, the system privileged instruction error handler is invoked.

If an attempt is made to execute ***mtspr*** specifying a TM SPR in other than Non-transactional state, a TM Bad Thing type Program interrupt is generated.

A complete description of this instruction can be found in Book III.

Special Registers Altered:

See above

Extended Mnemonics:

Examples of extended mnemonics for *Move To Special Purpose Register*:

Extended:		Equivalent to:	
mtxer	Rx	mtspr	1,Rx
mtlr	Rx	mtspr	8,Rx

Extended:		Equivalent to:	
mtctr	Rx	mtspr	9,Rx
mtppr	Rx	mtspr	896,Rx
mtppr32	Rx	mtspr	898,Rx

Programming Note

The AMR is part of the "context" of the program (see Book III-S). Therefore modification of the AMR requires "synchronization" by software. For this reason, most operating systems provide a system library program that application programs can use to modify the AMR.

Compiler and Assembler Note

For the ***mtspr*** and ***mfspr*** instructions, the SPR number coded in Assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(129): see Book III-S
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      RT ← SPR(n)
    else
      RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains 129, the instruction references the Transaction Failure Instruction Address Register (TFIAR)<TM> and the result is dependent on the privilege with which it is executed. See Book III-S. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a noop; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs". Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
3	00000 00011	DSCR[8]
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR[6]
128	00100 00000	TFHAR[7]
129	00100 00001	TFIAR[7]
130	00100 00010	TEXASR[7]
131	00100 00011	TEXASRU[7]
136	00100 01000	CTRL
256	01000 00000	VRSAVE
259	01000 00011	SPRG3
260	01000 00100	SPRG4[2]
261	01000 00101	SPRG5[2]
262	01000 00110	SPRG6[2]
263	01000 00111	SPRG7[2]
268	01000 01100	TB[3]
269	01000 01101	TBU[3]
512	10000 00000	SPEFSCR[4]
526	10000 01110	ATB[3,5]
527	10000 01111	ATBU[3,5]
768	11000 00000	SIER[6]
769	11000 00001	MMCR2[6]
770	11000 00010	MMCRA[6]
771	11000 00011	PMC1[6]
772	11000 00100	PMC2[6]
773	11000 00101	PMC3[6]
774	11000 00110	PMC4[6]
775	11000 00111	PMC5[6]
776	11000 01000	PMC6[6]
779	11000 01011	MMCR0[6]
780	11000 01100	SIAR[6]
781	11000 01101	SDAR[6]
782	11000 01110	MMCR1[6]
800	11001 00000	BESCRS[10]
801	11001 00001	BESCRSU[10]
802	11001 00010	BESCRR[10]
803	11001 00011	BESCRRU[10]
804	11001 00100	EBBHR[10]
805	11001 00101	EBBRR[10]
806	11001 00110	BESCR[10]
808	11001 01000	reserved[9]
809	11001 01001	reserved[9]
810	11001 01010	reserved[9]
811	11001 01011	reserved[9]
815	11001 01111	TAR[6]
896	11100 00000	PPR[10]
898	11100 00010	PPR32

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] Category: Embedded.
[3] See Chapter 6 of Book II.
[4] Category: SPE.
[5] Category: Alternate Time Base.
[6] Category: Server; see Book III-S.
[7] Category: Transactional Memory. See Chapter 5 of Book II.
[8] Category: Stream.
[9] Accesses to these SPRs are noops; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs".
[10] Category: Server; see Book II.

If execution of this instruction is attempted specifying an SPR number that is not shown above, or an SPR number that is shown above but is in a category that is not supported by the implementation, one of the following occurs.

- If spr_0 = 0, the illegal instruction error handler is invoked.
- If spr_0 = 1, the system privileged instruction error handler is invoked.

A complete description of this instruction can be found in Book III.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for Move From Special Purpose Register:

Extended:		Equivalent to:	
mfxer	Rx	mfspr	Rx,1
mflr	Rx	mfspr	Rx,8
mfctr	Rx	mfspr	Rx,9

Note

See the Notes that appear with ***mtspr***.

Move To Condition Register Fields
XFX-form

mtcrf FXM,RS

31	RS	0	FXM	/	144	/
0	6	11	12	20	21	31

$mask \leftarrow {}^{4}(FXM_0) \,||\, {}^{4}(FXM_1) \,||\, \ldots\, {}^{4}(FXM_7)$
$CR \leftarrow ((RS)_{32:63}\ \&\ mask) \mid (CR\ \&\ \neg mask)$

The contents of bits 32:63 of register RS are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FXM_i=1 then CR field i (CR bits 4×i+32:4×i+35) is set to the contents of the corresponding field of the low-order 32 bits of RS.

Special Registers Altered:

CR fields selected by mask

Extended Mnemonics:

Example of extended mnemonics for *Move To Condition Register Fields*:

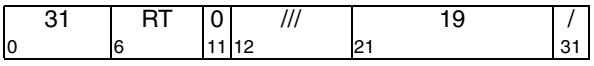
Extended:		Equivalent to:	
mtcr	Rx	mtcrf	0xFF,Rx

Programming Note

In the preferred form of this instruction (***mtocrf***), only one Condition Register field is updated.

Move From Condition Register XFX-form

mfcr RT



RT ← $^{32}0$ || CR

The contents of the Condition Register are placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

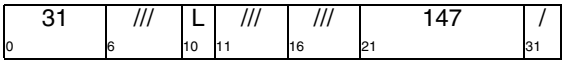
Special Registers Altered:

None

Move To Split Little Endian X-form

mtsle L

[Category: Server]



SLE ← L

The value specified in the L field is placed into SLE.

L=0

SLE is set to 0. The Endian mode used for both data and instruction storage accesses is represented by the value specified by LE.

L=1

SLE is set to 1. The Endian mode used for instruction storage accesses is represented by the value of LE. The Endian mode used for data storage accesses is represented by the value of ¬LE.

The effects of executing ***mtsle*** are immediately effective for subsequent storage accesses.

Special Registers Altered:

MSR (see Book III-S)

Move From VSR Doubleword XX1-form
[Category: Vector-Scalar]

mfvsrd RA,XS

31	S	RA	///	51	SX
0	6	11	16	21	31

```
XS ← SX || S
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()
GPR[RA] ← VSR[XS].doubleword[0]
```

Let XS be the value SX concatenated with S.

The contents of doubleword element 0 of VSR[XS] are placed into GPR[RA].

For SX=0, ***mfvsrd*** is treated as a *Floating-Point* instruction in terms of resource availability.

For SX=1, ***mfvsrd*** is treated as a *Vector* instruction in terms of resource availability.

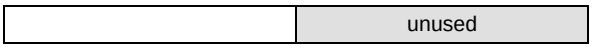
Extended Mnemonics		***Equivalent To***	
mffprd	RA,FRS	mfvsrd	RA,FRS
mfvrd	RA,VRS	mfvsrd	RA,VRS+32

Special Registers Altered

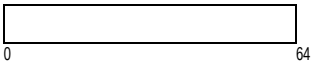
None

Data Layout for mfvsrd

src = VSR[XS]



tgt = GPR[RA]



Move From VSR Word and Zero XX1-form
[Category: Vector-Scalar]

mfvsrwz RA,XS (0x7C00_00E6)

31	S	RA	///	115	SX
0	6	11	16	21	31

```
XS ← SX || S
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()
GPR[RA] ← ExtendZero(VSR[XS].word[1])
```

Let XS be the value SX concatenated with S.

The contents of word element 1 of VSR[XS] are placed into bits 32:63 of GPR[RA]. The contents of bits 0:31 of GPR[RA] are set to 0.

For SX=0, ***mfvsrwz*** is treated as a *Floating-Point* instruction in terms of resource availability.

For SX=1, ***mfvsrwz*** is treated as a *Vector* instruction in terms of resource availability.

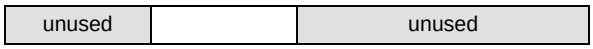
Extended Mnemonics		***Equivalent To***	
mffprwz	RA,FRS	mfvsrwz	RA,FRS
mfvrwz	RA,VRS	mfvsrwz	RA,VRS+32

Special Registers Altered

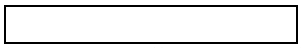
None

Data Layout for mfvsrwz

src = VSR[XS]

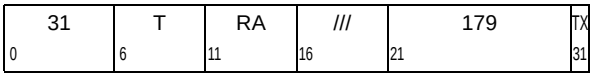


tgt = GPR[RA]



Move To VSR Doubleword XX1-form
[Category: Vector-Scalar]

mtvsrd XT,RA




```
XT ← TX || T
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()
VSR[XT].doubleword[0] ← GPR[RA]
VSR[XT].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.

The contents of GPR[RA] are placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrd*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrd*** is treated as a *Vector* instruction in terms of resource availability.

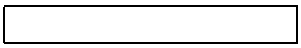
Extended Mnemonics		***Equivalent To***	
mtfprd	FRT,RA	mtvsrd	FRT,RA
mtvrd	VRT,RA	mtvsrd	VRT+32,RA

Special Registers Altered

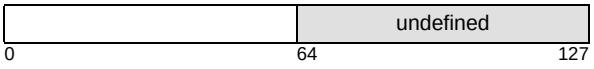
None

Data Layout for mtvsrd

src = GPR[RA]

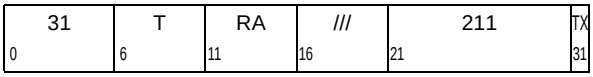


tgt = VSR[XT]



Move To VSR Word Algebraic XX1-form
[Category: Vector-Scalar]

mtvsrwa XT,RA




```
XT ← TX || T
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()
VSR[XT].doubleword[0] ← ExtendSign(GPR[RA].bit[32:63])
VSR[XT].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.

The two's-complement integer in bits 32:63 of GPR[RA] is sign-extended to 64 bits and placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrwa*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrwa*** is treated as a *Vector* instruction in terms of resource availability.

Extended Mnemonics		***Equivalent To***	
mtfprwa	FRT,RA	mtvsrwa	FRT,RA
mtvrwa	VRT,RA	mtvsrwa	VRT+32,RA

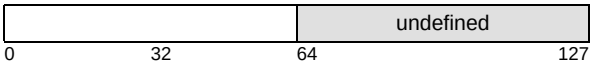
Special Registers Altered

None

Data Layout for mtvsrwa

src = GPR[RA]

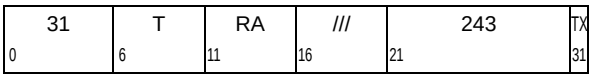
tgt = VSR[XT]



Move To VSR Word and Zero XX1-form

[Category: Vector-Scalar]

mtvsrwz XT,RA




```
XT ← TX || T
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()
VSR[XT].doubleword[0] ← ExtendZero(GPR[RA].word[1])
VSR[XT].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.

The contents of bits 32:63 of GPR[RA] are placed into word element 1 of VSR[XT]. The contents of word element 0 of VSR[XT] are set to 0.

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrwz*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrwz*** is treated as a *Vector* instruction in terms of resource availability.

Extended Mnemonics		***Equivalent To***	
mtfprwz	FRT,RA	mtvsrwz	FRT,RA
mtvrwz	VRT,RA	mtvsrwz	VRT+32,RA

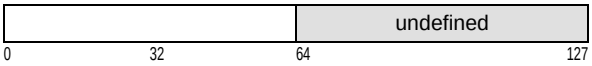
Special Registers Altered

None

Data Layout for mtvsrwz

src = GPR[RA]

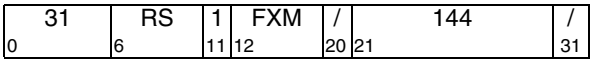
tgt = VSR[XT]



3.3.16.1 Move To/From One Condition Register Field Instructions

Move To One Condition Register Field *XFX-form*

mtocrf FXM,RS




```
count ← 0
do i = 0 to 7
  if FXM_i = 1 then
    n ← i
    count ← count + 1
if count = 1 then
   CR_{4×n+32:4×n+35} ← (RS)_{4×n+32:4×n+35}
else CR ← undefined
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \le n \le 7$). The contents of bits 4×n+32:4×n+35 of register RS are placed into CR field n (CR bits 4×n+32:4×n+35). Otherwise, the contents of the Condition Register are undefined.

Special Registers Altered:

CR field selected by FXM

Programming Note

These forms of the ***mtcrf*** and ***mfcr*** instructions are intended to replace the old forms of the instructions (the forms shown in page 107), which will eventually be phased out of the architecture. The new forms are backward compatible with most processors that comply with versions of the architecture that precede Version 2.00. On those processors, the new forms are treated as the old forms.

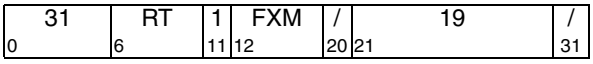
However, on some processors that comply with versions of the architecture that precede Version 2.00 the new forms may be treated as follows:

mtocrf: may cause the system illegal instruction error handler to be invoked

mfocrf: may place an undefined value into register RT

Move From One Condition Register Field *XFX-form*

mfocrf RT,FXM




```
RT ← undefined
count ← 0
do i = 0 to 7
  if FXM_i = 1 then
    n ← i
    count ← count + 1
if count = 1 then
   RT_{4×n+32:4×n+35} ← CR_{4×n+32:4×n+35}
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \le n \le 7$). The contents of CR field n (CR bits 4×n+32:4×n+35) are placed into bits 4×n+32:4×n+35 of register RT and the contents of the remaining bits of register RT are undefined. Otherwise, the contents of register RT are undefined.

Special Registers Altered:

None

3.3.16.2 Move To/From System Registers [Category: Embedded]

Move to Condition Register from XER *X-form*

mcrxr BF

31	BF	//	///	///	512	/
0	6	9	11	16	21	31

$CR_{4\times BF+32:4\times BF+35} \leftarrow XER_{32:35}$
$XER_{32:35} \leftarrow 0b0000$

The contents of $XER_{32:35}$ are copied to Condition Register field BF. $XER_{32:35}$ are set to zero.

Special Registers Altered:

CR field BF $XER_{32:35}$

Move To Device Control Register User-mode Indexed *X-form*

mtdcrux RS,RA
[Category: Embedded.Device Control]

31	RS	RA	///	419	/
0	6	11	16	21	31

DCRN ← (RA)
DCR(DCRN) ← RS

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of bits 32:63 of RS are placed into the Device Control Register.

See “Move To Device Control Register Indexed X-form” on page 1054 in Book III for more information on this instruction.

Special Registers Altered:

Implementation-dependent

Move From Device Control Register User-mode Indexed *X-form*

mfdcrux RT,RA
[Category: Embedded.Device Control]

31	RT	RA	///	291	/
0	6	11	16	21	31

DCRN ← (RA)
RT ← DCR(DCRN)

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into RT. For 32-bit Device Control Registers, the contents of bits 32:63 of the designated Device Control Register are placed into RT.

See “Move From Device Control Register Indexed X-form” on page 1055 in Book III for more information on this instruction.

Special Registers Altered:

Implementation-dependent

Chapter 4. Floating-Point Facility [Category: Floating-Point]

4.1 Floating-Point Facility Overview

This chapter describes the registers and instructions that make up the Floating-Point Facility.

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic" (hereafter referred to as "the IEEE standard"). That standard defines certain required "operations" (addition, subtraction, etc.). Herein, the term "floating-point operation" is used to refer to one of these required operations and to additional operations defined (e.g., those performed by *Multiply-Add* or *Reciprocal Estimate* instructions). A Non-IEEE mode is also provided. This mode, which may produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the Floating-Point Status and Control Register explicitly.

These instructions are divided into two categories.

- computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.6 through 4.6.8.

- non-computational instructions

  The non-computational instructions are those that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the Floating-Point Status and Control Register explicitly, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations. With the exception of the instructions that manipulate the Floating-Point Status and Control Register explicitly, they do not alter the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.2 through 4.6.5, and 4.6.10.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values that are "Not a Number" (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to the Floating-Point Facility: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the Floating-Point Status and Control Register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - Infinity - Infinity (VXISI)
  - Infinity÷Infinity (VXIDI)
  - Zero÷Zero (VXZDZ)
  - Infinity×Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)

Invalid Integer Convert (VXCVI)
- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XX)

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 4.2.2, "Floating-Point Status and Control Register" on page 116 for a description of these exception and enable bits, and Section 4.4, "Floating-Point Exceptions" on page 124 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

4.2 Floating-Point Facility Registers

4.2.1 Floating-Point Registers

Implementations of this architecture provide 32 floating-point registers (FPRs). The floating-point instruction formats provide 5-bit fields for specifying the FPRs to be used in the execution of the instruction. The FPRs are numbered 0-31. See Figure 50 on page 116.

Each FPR contains 64 bits that support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the *Move* and *Select* instructions, operate on data located in FPRs and, with the exception of the *Compare* instructions, place the result value into an FPR and optionally (when Rc=1) place status information into the Condition Register. Instruction forms with Rc=1 are part of Category: Floating-Point.Record.

Load Double and *Store Double* instructions are provided that transfer 64 bits of data between storage and the FPRs with no conversion. *Load Single* instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. *Store Single* instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Instructions are provided that manipulate the Floating-Point Status and Control Register and the Condition Register explicitly. Some of these instructions copy data from an FPR to the Floating-Point Status and Control Register or vice versa.

The computational instructions and the *Select* instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

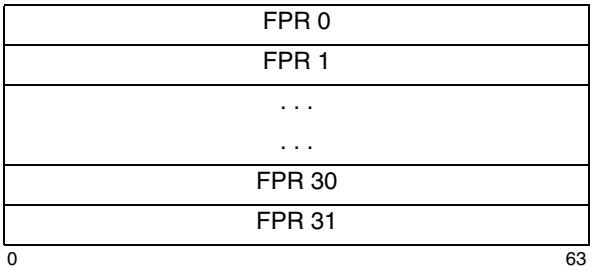


Figure 50. Floating-Point Registers

4.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 32:55 are status bits. Bits 56:63 are control bits.

The exception bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mcrfs***, ***mtfsfi***, ***mtfsf***, or ***mtfsb0*** instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be "exception bits", and only FX is sticky.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

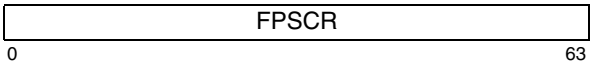


Figure 51. Floating-Point Status and Control Register

The bit definitions for the FPSCR are as follows.

Bit(s)	Description
0:31	Reserved
32	***Floating-Point Exception Summary*** (FX) Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets $FPSCR_{FX}$ to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter $FPSCR_{FX}$ explicitly.

Programming Note

$FPSCR_{FX}$ is defined not to be altered implicitly by ***mtfsfi*** and ***mtfsf*** because permitting these instructions to alter $FPSCR_{FX}$ implicitly could cause a paradox. An example is an ***mtfsfi*** or ***mtfsf*** instruction that supplies 0 for $FPSCR_{FX}$ and 1 for $FPSCR_{OX}$, and is executed when $FPSCR_{OX}$=0. See also the Programming Notes with the definition of these two instructions.

33 ***Floating-Point Enabled Exception Summary*** (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{FEX}$ explicitly.

34 ***Floating-Point Invalid Operation Exception Summary*** (VX)
This bit is the OR of all the Invalid Operation exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{VX}$ explicitly.

35 ***Floating-Point Overflow Exception*** (OX)
See Section 4.4.3, “Overflow Exception” on page 127.

36 ***Floating-Point Underflow Exception*** (UX)
See Section 4.4.4, “Underflow Exception” on page 128.

37 ***Floating-Point Zero Divide Exception*** (ZX)
See Section 4.4.2, “Zero Divide Exception” on page 126.

38 ***Floating-Point Inexact Exception*** (XX)
See Section 4.4.5, “Inexact Exception” on page 128.

$FPSCR_{XX}$ is a sticky version of $FPSCR_{FI}$ (see below). Thus the following rules completely describe how $FPSCR_{XX}$ is set by a given instruction.

- If the instruction affects $FPSCR_{FI}$, the new value of $FPSCR_{XX}$ is obtained by ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
- If the instruction does not affect $FPSCR_{FI}$, the value of $FPSCR_{XX}$ is unchanged.

39 ***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN)
See Section 4.4.1, “Invalid Operation Exception” on page 126.

40 ***Floating-Point Invalid Operation Exception (∞ - ∞)*** (VXISI)
See Section 4.4.1.

41 ***Floating-Point Invalid Operation Exception (∞ ÷ ∞)*** (VXIDI)
See Section 4.4.1.

42 ***Floating-Point Invalid Operation Exception (0 ÷ 0)*** (VXZDZ)
See Section 4.4.1.

43 ***Floating-Point Invalid Operation Exception (∞ × 0)*** (VXIMZ)
See Section 4.4.1.

44 ***Floating-Point Invalid Operation Exception (Invalid Compare)*** (VXVC)
See Section 4.4.1.

45 ***Floating-Point Fraction Rounded*** (FR)
The last *Arithmetic* or *Rounding and Conversion* instruction incremented the fraction during rounding. See Section 4.3.6, “Rounding” on page 123. This bit is not sticky.

46 ***Floating-Point Fraction Inexact*** (FI)
The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 4.3.6. This bit is not sticky.

See the definition of $FPSCR_{XX}$, above, regarding the relationship between $FPSCR_{FI}$ and $FPSCR_{XX}$.

47:51 ***Floating-Point Result Flags*** (FPRF)
Arithmetic, rounding, and *Convert From Integer* instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined. Floating-point *Compare* instructions set this field based on the relative values of the operands being compared. For *Convert To Integer* instructions, the value placed into FPRF is undefined. Additional details are given below.

Programming Note

A single-precision operation that produces a denormalized result sets FPRF to indicate a denormalized number. When possible, single-precision denormalized numbers are represented in normalized double format in the target register.

47 ***Floating-Point Result Class Descriptor*** (C)
Arithmetic, rounding, and *Convert From Integer* instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 52 on page 119.

48:51 ***Floating-Point Condition Code*** (FPCC)
Floating-point *Compare* instructions set one of

the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and *Convert From Integer* instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 52 on page 119. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 ***Floating-Point Less Than or Negative*** (FL or <)

49 ***Floating-Point Greater Than or Positive*** (FG or >)

50 ***Floating-Point Equal or Zero*** (FE or =)

51 ***Floating-Point Unordered or NaN*** (FU or ?)

52 Reserved

53 ***Floating-Point Invalid Operation Exception (Software-Defined Condition)*** (VXSOFT)
This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 4.4.1.

Programming Note

$FPSCR_{VXSOFT}$ can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation Exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.

54 ***Floating-Point Invalid Operation Exception (Invalid Square Root)*** (VXSQRT)
See Section 4.4.1.

55 ***Floating-Point Invalid Operation Exception (Invalid Integer Convert)*** (VXCVI)
See Section 4.4.1.

56 ***Floating-Point Invalid Operation Exception Enable*** (VE)
See Section 4.4.1.

57 ***Floating-Point Overflow Exception Enable*** (OE)
See Section 4.4.3, “Overflow Exception” on page 127.

58 ***Floating-Point Underflow Exception Enable*** (UE)
See Section 4.4.4, “Underflow Exception” on page 128.

59 ***Floating-Point Zero Divide Exception Enable*** (ZE)
See Section 4.4.2, “Zero Divide Exception” on page 126.

60 ***Floating-Point Inexact Exception Enable*** (XE)
See Section 4.4.5, “Inexact Exception” on page 128.

61 ***Floating-Point Non-IEEE Mode*** (NI)
Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved, and the remainder of the definition of this bit does not apply.

If floating-point non-IEEE mode is implemented, this bit has the following meaning.

0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard).
1 The processor is in floating-point non-IEEE mode.

When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits may have meanings different from those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a given floating-point instruction with $FPSCR_{NI}$=1, and any additional requirements for using non-IEEE mode, are implementation-dependent. The results of executing a given instruction in non-IEEE mode may vary between implementations, and between different executions on the same implementation.

Programming Note

When the processor is in floating-point non-IEEE mode, the results of floating-point operations may be approximate, and performance for these operations may be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation may return 0 instead of a denormalized number, and may return a large number instead of an infinity.

62:63 ***Floating-Point Rounding Control*** (RN) See Section 4.3.6, “Rounding” on page 123.

00 Round to Nearest
01 Round toward Zero
10 Round toward +Infinity
11 Round toward -Infinity

Result Flags					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	- Infinity
0	1	0	0	0	- Normalized Number
1	1	0	0	0	- Denormalized Number
1	0	0	1	0	- Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Denormalized Number
0	0	1	0	0	+ Normalized Number
0	0	1	0	1	+ Infinity

Figure 52. Floating-Point Result Flags

4.3 Floating-Point Data

4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in floating-point registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below.

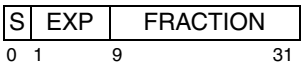


Figure 53. Floating-point single format

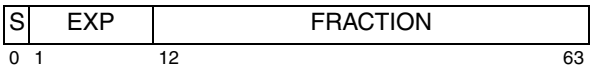


Figure 54. Floating-point double format

Values in floating-point format are composed of three fields:

S	sign bit
EXP	exponent+bias
FRACTION	fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (i.e., the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Figure 55.

	Format	
	Single	Double
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	-126	-1022
Widths (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

Figure 55. IEEE floating-point fields

The architecture requires that the FPRs of the Floating-Point Facility support the floating-point double format only.

4.3.2 Value Representation

This architecture defines numeric and non-numeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 56.

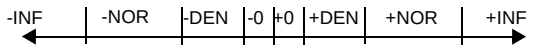


Figure 56. Approximation to real numbers

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers
Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (± NOR)
These are values that have a biased exponent value in the range:

1 to 254 in single format
1 to 2046 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$NOR = (-1)^s \times 2^E \times (1.fraction)$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

Single Format:

$1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$

Double Format:

$2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$

Zero values (± 0)
These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to -0).

Denormalized numbers (± DEN)
These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

$DEN = (-1)^s \times 2^{Emin} \times (0.fraction)$

where Emin is the minimum representable exponent value (-126 for single-precision, -1022 for double-precision).

Infinities (± ∞)
These are values that have the maximum biased exponent value:

255 in single format
2047 in double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

$$-\infty < \text{every finite number} < +\infty$$

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 4.4.1, “Invalid Operation Exception” on page 126.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.

Not a Numbers (NaNs)
These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (i.e., NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0 then the NaN is a *Signaling NaN*; otherwise it is a *Quiet NaN*.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled ($FPSCR_{VE}$=0). Quiet NaNs propagate through all floating-point operations except ordered comparison, *Floating Round to Single-Precision*, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if (FRA) is a NaN
   then FRT ← (FRA)
   else if (FRB) is a NaN
      then if instruction is frsp
           then FRT ← (FRB)0:34 || 29 0
           else FRT ← (FRB)
      else if (FRC) is a NaN
           then FRT ← (FRC)
           else if generated QNaN
                then FRT ← generated QNaN
```

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is ***frsp***. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as the result of a disabled Invalid Operation

Exception generates this QNaN (i.e., 0x7FF8_0000_0000_0000).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

4.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation x-y is the same as the sign of the result of the add operation x+(-y).

  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward -Infinity, in which mode the sign is negative.

- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is -Infinity.

- The sign of the result of a *Round to Single-Precision,* or *Convert From Integer*, or *Round to Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

4.3.4 Normalization and Denormalization

The intermediate result of an arithmetic or ***frsp*** instruction may require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 4.5.1, "Execution Model for IEEE Operations" on page 129) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be "Tiny" and the stored result is determined by the rules described in Section 4.4.4, "Underflow Exception". These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process then "Loss of Accuracy" has occurred (See Section 4.4.4, "Underflow Exception" on page 128) and Underflow Exception is signaled.

4.3.5 Data Handling and Precision

Most of the *Floating-Point Facility Architecture*, including all computational, *Move*, and *Select* instructions, use the floating-point double format to represent data in the FPRs. Single-precision and integer-valued operands may be manipulated using double-precision operations. Instructions are provided to coerce these values from a double format operand. Instructions are also provided for manipulations which do not require double-precision. In addition, instructions are provided to access a true single-precision representation in storage, and a fixed-point integer representation in GPRs.

4.3.5.1 Single-Precision Operands

For single format data, a format conversion from single to double is performed when loading from storage into an FPR and a format conversion from double to single is performed when storing from an FPR to storage. No floating-point exceptions are caused by these instructions. An instruction is provided to explicitly convert a double format operand in an FPR to single-precision. Floating-point single-precision is enabled with four types of instruction.

1. Load Floating-Point Single

   This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into an FPR. No floating-point exceptions are caused by these instructions.

2. Round to Floating-Point Single-Precision

   The Floating Round to Single-Precision instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR in double format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the Floating Round to Single-Precision instruction, this operation does not alter the value.

3. Single-Precision Arithmetic Instructions

   This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

   If any input value is not representable in single format and either OE=1 or UE=1, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

   For ***fres***[**.**] or ***frsqrtes***[**.**], if the input value is finite and has an unbiased exponent greater than +127, the input value is interpreted as an Infinity.

4. Store Floating-Point Single

   This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

When the result of a *Load Floating-Point Single*, *Floating Round to Single-Precision*, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 FRACTION bits are zero.

Programming Note

The *Floating Round to Single-Precision* instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions and by ***fcfid***) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a *Floating Round to Single-Precision* instruction.

Programming Note

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

4.3.5.2 Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and fixed-Point facilities, instructions are provided to convert between floating-point double format and fixed-point integer format in an FPR. Computation on integer-valued operands may be performed using arithmetic instructions of the required precision. (The results may not be integer values.) The two groups of instructions provided specifically to support integer-valued operands are described below.

1. Floating Round to Integer

   The *Floating Round to Integer* instructions round a double-precision operand to an integer value in floating-point double format. These instructions may cause Invalid Operation (VXSNAN) exceptions. See Sections 4.3.6 and 4.5.1 for more information about rounding.

2. Floating Convert To/From Integer

   The *Floating Convert To Integer* instructions convert a double-precision operand to a 32-bit or 64-bit signed fixed-point integer format. Variants are provided both to perform rounding based on

the value of $FPSCR_{RN}$ and to round toward zero. These instructions may cause Invalid Operation (VXSNaN, VXCVI) and Inexact exceptions. The *Floating Convert From Integer* instruction converts a 64-bit signed fixed-point integer to a double-precision floating-point integer. Because of the limitations of the source format, only an Inexact exception may be generated.

4.3.6 Rounding

The material in this section applies to operations that have numeric operands (i.e., operands that are not infinities or NaNs). Rounding the intermediate result of such an operation may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 4.3.2, “Value Representation” and Section 4.4, “Floating-Point Exceptions” for the cases not covered here.

The *Arithmetic* and *Rounding and Conversion* instructions round their intermediate results. With the exception of the *Estimate* instructions, these instructions produce an intermediate result that can be regarded as having infinite precision and unbounded exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target FPR in double format. The *Floating Round to Integer* and *Floating Convert To Integer* instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for *Floating Round to Integer* is normalized and put in double format, and for *Floating Convert To Integer* is converted to a signed fixed-point integer.

FPSCR bits FR and FI generally indicate the results of rounding. Each of the instructions which rounds its intermediate result sets these bits. If the fraction is incremented during rounding then FR is set to 1, otherwise FR is set to 0. If the result is inexact then FI is set to 1, otherwise FI is set to zero. The *Round to Integer* instructions are exceptions to this rule, setting FR and FI to 0. The *Estimate* instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and FI.

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 4.2.2, “Floating-Point Status and Control Register”. These are encoded as follows.

RN	Rounding Mode
00	Round to Nearest
01	Round toward Zero
10	Round toward +Infinity
11	Round toward -Infinity

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 57 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. “LSB” means “least significant bit”.

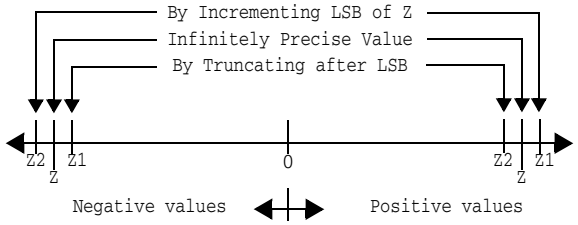


Figure 57. Selection of Z1 and Z2

Round to Nearest
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit 0).

Round toward Zero
Choose the smaller in magnitude (Z1 or Z2).

Round toward +Infinity
Choose Z1.

Round toward -Infinity
Choose Z2.

See Section 4.5.1, “Execution Model for IEEE Operations” on page 129 for a detailed explanation of rounding.

4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
  - Infinity - Infinity
  - Infinity÷Infinity
  - Zero÷Zero
  - Infinity×Zero
  - Invalid Compare
  - Software-Defined Condition
  - Invalid Square Root
  - Invalid Integer Convert
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions, other than Invalid Operation Exception due to Software-Defined Condition, may occur during execution of computational instructions. An Invalid Operation Exception due to Software-Defined Condition occurs when a *Move To FPSCR* instruction sets $FPSCR_{VXSOFT}$ to 1.

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 125), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) is set with Invalid Operation Exception ($\infty\times0$) for *Multiply-Add* instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Integer Convert) for *Convert To Integer* instructions.

When an exception occurs the writing of a result to the target register may be suppressed or a result may be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case; the expectation is that the exception will be detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case; the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled float-

ing-point exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1; the *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.) The effects of the four possible settings of these bits are as follows.

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.
1	0	**Imprecise Recoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction and there is only one such instruction. Otherwise it has not begun execution (or may have been partially executed in some cases, as described in Book III).

Programming Note

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. (It always applies in the latter case.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

4.4.1 Invalid Operation Exception

4.4.1.1 Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- Any floating-point operation on a Signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and nonzero) number (Invalid Square Root)
- Integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (Invalid Integer Convert)

An Invalid Operation Exception also occurs when an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1 (Software-Defined Condition).

4.4.1.2 Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled ($FPSCR_{VE}$=1) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty - \infty$)
   - $FPSCR_{VXIDI}$ (if $\infty \div \infty$)
   - $FPSCR_{VXZDZ}$ (if $0 \div 0$)
   - $FPSCR_{VXIMZ}$ (if $\infty \times 0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if sfw-def cond)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
2. If the operation is an arithmetic, *Floating Round to Single-Precision*, *Floating Round to Integer,* or convert to integer operation,
   - the target FPR is unchanged
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is unchanged
3. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
4. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1,
   - The FPSCR is set as specified in the instruction description.

When Invalid Operation Exception is disabled ($FPSCR_{VE}$=0) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty - \infty$)
   - $FPSCR_{VXIDI}$ (if $\infty \div \infty$)
   - $FPSCR_{VXZDZ}$ (if $0 \div 0$)
   - $FPSCR_{VXIMZ}$ (if $\infty \times 0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if sfw-def cond)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
2. If the operation is an arithmetic or *Floating Round to Single-Precision* operation,
   - the target FPR is set to a Quiet NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is set to indicate the class of the result (Quiet NaN)
3. If the operation is a convert to 64-bit integer operation,
   - the target FPR is set as follows:
     - FRT is set to the most positive 64-bit integer if the operand in FRB is a positive number or + ∞, and to the most negative 64-bit integer if the operand in FRB is a negative number, - ∞, or NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
4. If the operation is a convert to 32-bit integer operation,
   - the target FPR is set as follows:
     - $FRT_{0:31} \leftarrow$ undefined
     - $FRT_{32:63}$ are set to the most positive 32-bit integer if the operand in FRB is a positive number or +infinity, and to the most negative 32-bit integer if the operand in FRB is a negative number, -infinity, or NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
5. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
6. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1,
   - The FPSCR is set as specified in the instruction description.

4.4.2 Zero Divide Exception

4.4.2.1 Definition

A Zero Divide Exception occurs when a *Divide* instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a *Reciprocal Estimate* instruction (***fre***[***s***] or ***frsqrte***[***s***]) is executed with an operand value of zero.

4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled ($FPSCR_{ZE}$=1) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is unchanged
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is unchanged

When Zero Divide Exception is disabled ($FPSCR_{ZE}$=0) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is set to ± Infinity, where the sign is determined by the XOR of the signs of the operands
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Infinity)

4.4.3 Overflow Exception

4.4.3.1 Definition

An Overflow Exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

4.4.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled ($FPSCR_{OE}$=1) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled ($FPSCR_{OE}$=0) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
3. The result is determined by the rounding mode ($FPSCR_{RN}$) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store ± Infinity, where the sign is the sign of the intermediate result
   - Round toward Zero
     Store the format's largest finite number with the sign of the intermediate result
   - Round toward + Infinity
     For negative overflow, store the format's most negative finite number; for positive overflow, store +Infinity
   - Round toward - Infinity
     For negative overflow, store - Infinity; for positive overflow, store the format's largest finite number
4. The result is placed into the target FPR
5. $FPSCR_{FR}$ is undefined
6. $FPSCR_{FI}$ is set to 1
7. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Infinity or ± Normal Number)

4.4.4 Underflow Exception

4.4.4.1 Definition

Underflow Exception is defined separately for the enabled and disabled states:

- Enabled:
  Underflow occurs when the intermediate result is "Tiny".
- Disabled:
  Underflow occurs when the intermediate result is "Tiny" and there is "Loss of Accuracy".

A "Tiny" result is detected before rounding, when a non-zero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is "Tiny" and Underflow Exception is disabled ($FPSCR_{UE}$=0) then the intermediate result is denormalized (see Section 4.3.4, "Normalization and Denormalization" on page 121) and rounded (see Section 4.3.6, "Rounding" on page 123) before being placed into the target FPR.

"Loss of Accuracy" is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}$=1) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normalized Number)

Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow Exception, to simulate a "trap disabled" environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled ($FPSCR_{UE}$=0) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The rounded result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

4.4.5 Inexact Exception

4.4.5.1 Definition

An Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)
2. The rounded result overflows and Overflow Exception is disabled.

4.4.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When an Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

4.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 4.3.2 and Section 4.4 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. The Power ISA follows these guidelines; double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions and ***fcfid*** produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

4.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

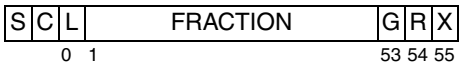


Figure 58. IEEE 64-bit execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due either to shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Figure 59 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

G R X	Interpretation
0 0 0	IR is exact
0 0 1	IR closer to NL
0 1 0	
0 1 1	
1 0 0	IR midway between NL and NH
1 0 1	IR closer to NH
1 1 0	
1 1 1	

Figure 59. Interpretation of G, R, and X bits

Figure 60 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figure 58.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26:52, G, R, X

Figure 60. Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction. Four user-selectable rounding modes are provided through $FPSCR_{RN}$ as described in Section 4.3.6, “Rounding” on page 123. Using Z1 and Z2 as defined on page 123, the rules for rounding in each mode are as follows.

- ***Round to Nearest***

  Guard bit = 0
  The result is truncated. (Result exact (GRX=000) or closest to next lower value in magnitude (GRX=001, 010, or 011))

  Guard bit = 1
  Depends on Round and Sticky bits:

  Case a
  If the Round or Sticky bit is 1 (inclusive), the result is incremented. (Result closest to next higher value in magnitude (GRX=101, 110, or 111))

  Case b
  If the Round and Sticky bits are 0 (result midway between closest representable values), then if the low-order bit of the result is 1 the result is incremented. Otherwise (the low-order bit of the result is 0) the result is truncated (this is the case of a tie rounded to even).

- ***Round toward Zero***
  Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.

- ***Round toward + Infinity***
  Choose Z1.

- ***Round toward - Infinity***
  Choose Z2.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. If any of the Guard, Round, or Sticky bits is nonzero, then the result is also inexact. Fraction bits are stored to the target FPR. For *Floating Round to Integer*, *Floating Round to Single-Precision*, and single-precision arithmetic instructions, low-order zeros must be appended as appropriate to fill out the double-precision fraction.

4.5.2 Execution Model for Multiply-Add Type Instructions

The Power ISA provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

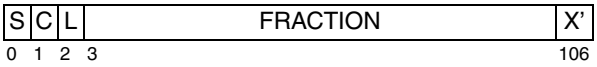


Figure 61. Multiply-add 64-bit execution model

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 62 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X'
Single	24	25	OR of 26:105, X'

Figure 62. Location of the Guard, Round, and Sticky bits in the multiply-add execution model

The rules for rounding the intermediate result are the same as those given in Section 4.5.1.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract*, the final result is negated.

4.6 Floating-Point Facility Instructions

For each instruction in this section that defines the use of an Rc bit, the behavior defined for the instruction corresponding to Rc=1 is considered part of the Floating-Point.Record category.

4.6.1 Floating-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 26.

Programming Note

The ***la*** extended mnemonic permits computing an effective address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address. This extended mnemonic is described in Section E.10, “Miscellaneous Mnemonics” on page 719.

4.6.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

4.6.2 Floating-Point Load Instructions

There are three basic forms of load instruction: single-precision, double-precision, and integer. The integer form is provided by the *Load Floating-Point as Integer Word Algebraic* instruction, described on page 136. Because the FPRs support only floating-point double format, single-precision *Load Floating-Point* instructions convert single-precision data to double format prior to loading the operand into the target FPR. The conversion and loading steps are as follows.

Let $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_{2} \leftarrow \neg WORD_{1}$
  $FRT_{3} \leftarrow \neg WORD_{1}$
  $FRT_{4} \leftarrow \neg WORD_{1}$
  $FRT_{5:63} \leftarrow WORD_{2:31} \,||\, {}^{29}0$

Denormalized Operand

  if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
  sign $\leftarrow WORD_{0}$
  exp $\leftarrow$ -126
  $frac_{0:52} \leftarrow 0b0 \,||\, WORD_{9:31} \,||\, {}^{29}0$
  normalize the operand
    do while $frac_{0} = 0$
      $frac_{0:52} \leftarrow frac_{1:52} \,||\, 0b0$
      exp $\leftarrow$ exp - 1
  $FRT_{0} \leftarrow$ sign
  $FRT_{1:11} \leftarrow$ exp + 1023
  $FRT_{12:63} \leftarrow frac_{1:52}$

Zero / Infinity / NaN

if $WORD_{1:8} = 255$ or $WORD_{1:31} = 0$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_{2} \leftarrow WORD_{1}$
  $FRT_{3} \leftarrow WORD_{1}$
  $FRT_{4} \leftarrow WORD_{1}$
  $FRT_{5:63} \leftarrow WORD_{2:31} \,||\, {}^{29}0$

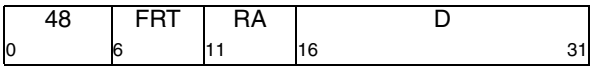
For double-precision *Load Floating-Point* instructions and for the *Load Floating-Point as Integer Word Algebraic* instruction no conversion is required, as the data from storage are copied directly into the FPR.

Many of the *Load Floating-Point* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA and the storage element (word or doubleword) addressed by EA is loaded into FRT.

Note: Recall that RA and RB denote General Purpose Registers, while FRT denotes a Floating-Point Register.

Load Floating-Point Single D-form

lfs FRT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+D.

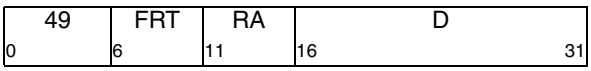
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.

Special Registers Altered:

None

Load Floating-Point Single with Update D-form

lfsu FRT,D(RA)




```
EA ← (RA) + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.

EA is placed into register RA.

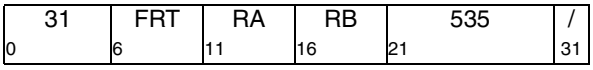
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Single Indexed X-form

lfsx FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← DOUBLE(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+(RB).

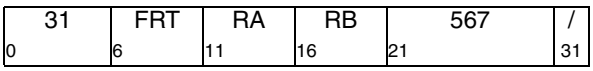
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.

Special Registers Altered:

None

Load Floating-Point Single with Update Indexed X-form

lfsux FRT,RA,RB




```
EA ← (RA) + (RB)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 133) and placed into register FRT.

EA is placed into register RA.

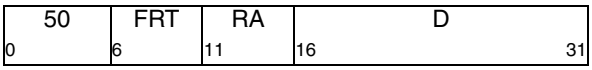
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double D-form

lfd FRT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
FRT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+D.

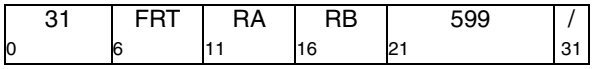
The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:

None

Load Floating-Point Double Indexed X-form

lfdx FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

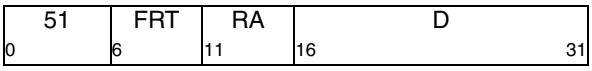
The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:

None

Load Floating-Point Double with Update D-form

lfdu FRT,D(RA)




```
EA ← (RA) + EXTS(D)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The doubleword in storage addressed by EA is loaded into register FRT.

EA is placed into register RA.

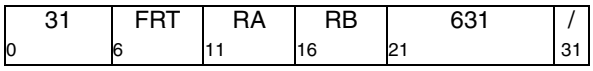
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double with Update Indexed X-form

lfdux FRT,RA,RB




```
EA ← (RA) + (RB)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The doubleword in storage addressed by EA is loaded into register FRT.

EA is placed into register RA.

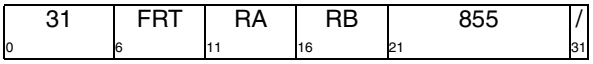
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point as Integer Word Algebraic Indexed X-form

lfiwax FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+(RB).

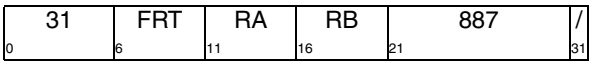
The word in storage addressed by EA is loaded into $FRT_{32:63}$. $FRT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Floating-Point as Integer Word and Zero Indexed X-form

lfiwzx FRT,RA,RB
[Category: Floating-Point.Phased-in]




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← 32 0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

The word in storage addressed by EA is loaded into $FRT_{32:63}$. $FRT_{0:31}$ are set to 0.

Special Registers Altered:

None

4.6.3 Floating-Point Store Instructions

There are three basic forms of store instruction: single-precision, double-precision, and integer. The integer form is provided by the *Store Floating-Point as Integer Word* instruction, described on page 140. Because the FPRs support only floating-point double format for floating-point data, single-precision *Store Floating-Point* instructions convert double-precision data to single format prior to storing the operand into storage. The conversion steps are as follows.

Let $WORD_{0:31}$ be the word in storage written to.

No Denormalization Required (includes Zero / Infinity / NaN)

if $FRS_{1:11} > 896$ or $FRS_{1:63} = 0$ then
    $WORD_{0:1} \leftarrow FRS_{0:1}$
    $WORD_{2:31} \leftarrow FRS_{5:34}$

Denormalization Required

if $874 \leq FRS_{1:11} \leq 896$ then
    sign $\leftarrow FRS_0$
    exp $\leftarrow FRS_{1:11}$ - 1023
    $frac_{0:52} \leftarrow 0b1 \,||\, FRS_{12:63}$
    denormalize operand
        do while exp < -126
            $frac_{0:52} \leftarrow 0b0 \,||\, frac_{0:51}$
            exp $\leftarrow$ exp + 1
    $WORD_0 \leftarrow$ sign
    $WORD_{1:8} \leftarrow 0x00$
    $WORD_{9:31} \leftarrow frac_{1:23}$
else WORD $\leftarrow$ undefined

Notice that if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision *Load Floating-Point* from WORD will not compare equal to the contents of the original source register).

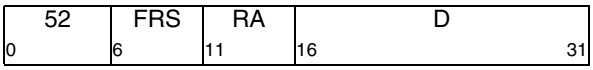
For double-precision *Store Floating-Point* instructions and for the *Store Floating-Point as Integer Word* instruction no conversion is required, as the data from the FPR are copied directly into storage.

Many of the *Store Floating-Point* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA.

Note: Recall that RA and RB denote General Purpose Registers, while FRS denotes a Floating-Point Register.

Store Floating-Point Single D-form

stfs FRS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
```

Let the effective address (EA) be the sum (RA|0)+D.

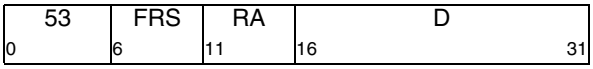
The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Single with Update D-form

stfsu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

EA is placed into register RA.

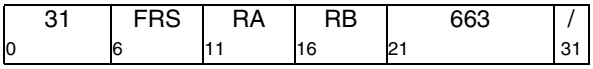
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Single Indexed X-form

stfsx FRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← SINGLE((FRS))
```

Let the effective address (EA) be the sum (RA|0)+(RB).

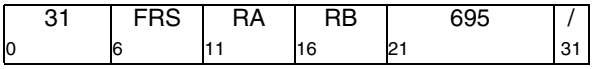
The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Single with Update Indexed X-form

stfsux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The contents of register FRS are converted to single format (see page 137) and stored into the word in storage addressed by EA.

EA is placed into register RA.

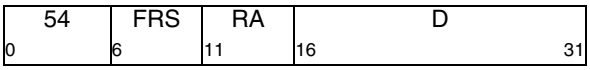
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double D-form

stfd FRS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 8) ← (FRS)
```

Let the effective address (EA) be the sum (RA|0)+D.

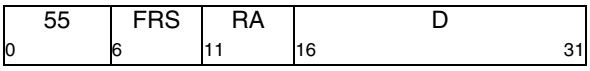
The contents of register FRS are stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Double with Update D-form

stfdu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 8) ← (FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

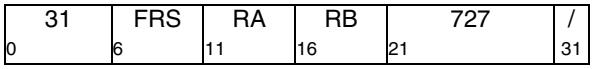
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double Indexed X-form

stfdx FRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (FRS)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

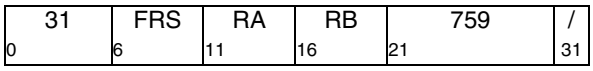
The contents of register FRS are stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Double with Update Indexed X-form

stfdux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 8) ← (FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point as Integer Word Indexed X-form

stfiwx FRS,RA,RB

31	FRS	RA	RB	983	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (FRS)32:63
```

Let the effective address (EA) be the sum (RA|0)+(RB).

$(FRS)_{32:63}$ are stored, without conversion, into the word in storage addressed by EA.

If the contents of register FRS were produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a single-precision *Arithmetic* instruction, or ***frsp***, then the value stored is undefined. (The contents of register FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of register FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more *Floating-Point Move* instructions, with the input to the sequence having been produced directly by such an instruction.)

Special Registers Altered:

None

4.6.4 Floating-Point Load Store Doubleword Pair Instructions [Category: Floating-Point.Phased-Out]

For ***lfdp[x]***, the doubleword-pair in storage addressed by EA is loaded into an even-odd pair of FPRs with the even-numbered FPR being loaded with the leftmost doubleword from storage and the odd-numbered FPR being loaded with the rightmost doubleword.

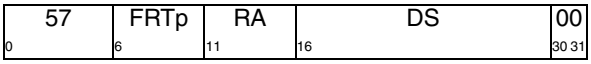
For ***stfdp[x]***, the content of an even-odd pair of FPRs is stored into the doubleword-pair in storage addressed by EA, with the even-numbered FPR being stored into the leftmost doubleword in storage and the odd-numbered FPR being stored into the rightmost doubleword.

Programming Note

The instructions described in this section should not be used to access an operand in DFP128 format when LE ≠ SLE.

Load Floating-Point Double Pair DS-form

lfdp FRTp,DS(RA)




```
if RA = 0 then b ← 0
else          b ←(RA)
EA ← b + EXTS(DS||0b00)
FRTp ← MEM(EA, 16)
```

Let the effective address (EA) be the sum (RA|0) + (DS||0b00). The doubleword-pair in storage addressed by EA is placed into register-pair FRTp.

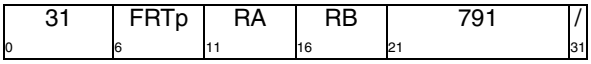
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double Pair Indexed X-form

lfdpx FRTp,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRTp ← MEM(EA, 16)
```

Let the effective address (EA) be the sum (RA|0) + (RB). The doubleword-pair in storage addressed by EA is placed into register-pair FRTp.

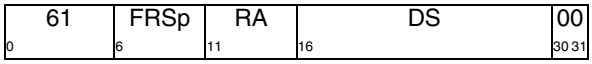
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double Pair DS-form

stfdp FRSp,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS||0b00)
MEM(EA, 16) ← FRSp
```

Let the effective address (EA) be the sum (RA|0) + (DS||0b00). The contents of register-pair FRSp are stored into the doubleword-pair in storage addressed by EA.

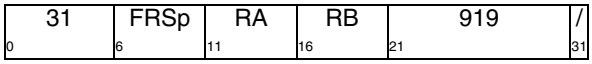
If FRSp is odd, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double Pair Indexed X-form

stfdpx FRSp,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 16) ← FRSp
```

Let the effective address (EA) be the sum (RA|0) + (RB). The contents of register-pair FRSp are stored into the doubleword-pair in storage addressed by EA.

If FRSp is odd, the instruction form is invalid.

Special Registers Altered:

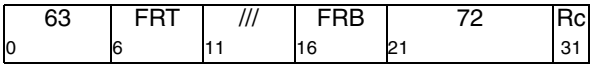
None

4.6.5 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for ***fneg***, ***fabs***, ***fnabs***, and ***fcpsgn***. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of a NaN may be altered by ***fneg***, ***fabs***, ***fnabs***, and ***fcpsgn***). These instructions do not alter the FPSCR.

Floating Move Register X-form

fmr FRT,FRB (Rc=0)
fmr. FRT,FRB (Rc=1)

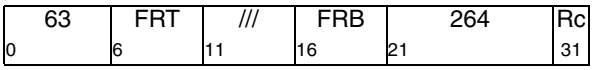


The contents of register FRB are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Absolute Value X-form

fabs FRT,FRB (Rc=0)
fabs. FRT,FRB (Rc=1)

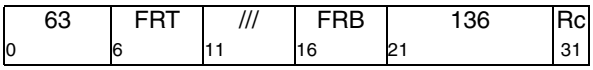


The contents of register FRB with bit 0 set to zero are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Negative Absolute Value X-form

fnabs FRT,FRB (Rc=0)
fnabs. FRT,FRB (Rc=1)

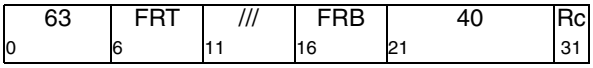


The contents of register FRB with bit 0 set to one are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Negate X-form

fneg FRT,FRB (Rc=0)
fneg. FRT,FRB (Rc=1)

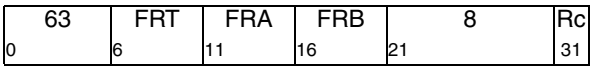


The contents of register FRB with bit 0 inverted are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Copy Sign X-form

fcpsgn FRT, FRA, FRB (Rc=0)
fcpsgn. FRT, FRA, FRB (Rc=1)



The contents of register FRB with bit 0 set to the value of bit 0 of register FRA are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Merge Even Word X-form
[Category: Vector-Scalar]

fmrgew FRT,FRA,FRB

63	FRT	FRA	FRB	966	/
0	6	11	16	21	31

```
if MSR.FP=0 then FP_Unavailable()
FPR[FRT].word[0] ← FPR[FRA].word[0]
FPR[FRT].word[1] ← FPR[FRB].word[0]
```

The contents of word element 0 of FPR[FRA] are placed into word element 0 of FPR[FRT].

The contents of word element 0 of FPR[FRB] are placed into word element 1 of FPR[FRT].

fmrgew is treated as a *Floating-Point* instruction in terms of resource availability.

Special Registers Altered

None

Floating Merge Odd Word X-form
[Category: Vector-Scalar]

fmrgow FRT,FRA,FRB

63	FRT	FRA	FRB	838	/
0	6	11	16	21	31

```
if MSR.FP=0 then FP_Unavailable()
FPR[FRT].word[0] ← FPR[FRA].word[1]
FPR[FRT].word[1] ← FPR[FRB].word[1]
```

The contents of word element 1 of FPR[FRA] are placed into word element 0 of FPR[FRT].

The contents of word element 1 of FPR[FRB] are placed into word element 1 of FPR[FRT].

fmrgow is treated as a *Floating-Point* instruction in terms of resource availability.

Special Registers Altered

None

4.6.6 Floating-Point Arithmetic Instructions

4.6.6.1 Floating-Point Elementary Arithmetic Instructions

Floating Add [Single] A-form

fadd FRT,FRA,FRB (Rc=0)
fadd. FRT,FRA,FRB (Rc=1)

63	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

fadds FRT,FRA,FRB (Rc=0)
fadds. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is added to the floating-point operand in register FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Subtract [Single] A-form

fsub FRT,FRA,FRB (Rc=0)
fsub. FRT,FRA,FRB (Rc=1)

63	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

fsubs FRT,FRA,FRB (Rc=0)
fsubs. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Multiply [Single] A-form

fmul FRT,FRA,FRC (Rc=0)
fmul. FRT,FRA,FRC (Rc=1)

63	FRT	FRA	///	FRC	25	Rc
0	6	11	16	21	26	31

fmuls FRT,FRA,FRC (Rc=0)
fmuls. FRT,FRA,FRC (Rc=1)

59	FRT	FRA	///	FRC	25	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXIMZ
CR1 (if Rc=1)

Floating Divide [Single] A-form

fdiv FRT,FRA,FRB (Rc=0)
fdiv. FRT,FRA,FRB (Rc=1)

63	FRT	FRA	FRB	///	18	Rc
0	6	11	16	21	26	31

fdivs FRT,FRA,FRB (Rc=0)
fdivs. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	///	18	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is divided by the floating-point operand in register FRB. The remainder is not supplied as a result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ
CR1 (if Rc=1)

Floating Square Root [Single] A-form

fsqrt FRT,FRB (Rc=0)
fsqrt. FRT,FRB (Rc=1)

63	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

fsqrts FRT,FRB (Rc=0)
fsqrts. FRT,FRB (Rc=1)

59	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

The square root of the floating-point operand in register FRB is placed into register FRT.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[1]	VXSQRT
< 0	QNaN[1]	VXSQRT
-0	-0	None
$+\infty$	$+\infty$	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1] No result if $FPSCR_{VE} = 1$

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI FX OX UX XX
VXSNAN VXSQRT
CR1 (if Rc=1)

Floating Reciprocal Estimate [Single] A-form

fre FRT,FRB (Rc=0)
fre. FRT,FRB (Rc=1)

63	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

fres FRT,FRB (Rc=0)
fres. FRT,FRB (Rc=1)

59	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

An estimate of the reciprocal of the floating-point operand in register FRB is placed into register FRT. Unless the reciprocal would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate is correct to a precision of one part in 256 of the reciprocal of (FRB), i.e.,

$$\text{ABS}(\frac{\text{estimate} - 1/x}{1/x}) \leq \frac{1}{256}$$

where x is the initial value in FRB.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	-0	None
-0	$-\infty$[1]	ZX
+0	$+\infty$[1]	ZX
$+\infty$	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if $FPSCR_{ZE} = 1$.
[2] No result if $FPSCR_{VE} = 1$.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

FPRF FR (undefined) FI (undefined)
FX OX UX ZX XX (undefined)
VXSNAN
CR1 (if Rc=1)

Programming Note

For the *Floating-Point Estimate* instructions, some implementations might implement a precision higher than the minimum architected precision. Thus, a program may take advantage of the higher precision instructions to increase performance by decreasing the iterations needed for software emulation of floating-point instructions. However, there is no guarantee given about the precision which may vary (up or down) between implementations. Only programs targeted at a specific implementation (i.e., the program will not be migrated to another implementation) should take advantage of the higher precision of the instructions. All other programs should rely on the minimum architected precision, which will guarantee the program to run properly across different implementations.

Floating Reciprocal Square Root Estimate [Single] A-form

frsqrte FRT,FRB (Rc=0)
frsqrte. FRT,FRB (Rc=1)

63	FRT	///	FRB	///	26	Rc
0	6	11	16	21	26	31

frsqrtes FRT,FRB (Rc=0)
frsqrtes. FRT,FRB (Rc=1)

59	FRT	///	FRB	///	26	Rc
0	6	11	16	21	26	31

A estimate of the reciprocal of the square root of the floating-point operand in register FRB is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 32 of the reciprocal of the square root of (FRB), i.e.,

$$\text{ABS}(\frac{\text{estimate} - 1/(\sqrt{x})}{1/(\sqrt{x})}) \leq \frac{1}{32}$$

where x is the initial value in FRB.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[2]	VXSQRT
< 0	QNaN[2]	VXSQRT
-0	$-\infty$[1]	ZX
+0	$+\infty$[1]	ZX
$+\infty$	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if $FPSCR_{ZE}$ = 1.
[2] No result if $FPSCR_{VE}$ = 1.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

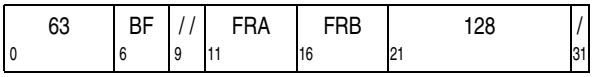
FPRF FR (undefined) FI (undefined)
FX ZX XX (undefined)
VXSNAN VXSQRT
CR1 (if Rc=1)

Note

See the Notes that appear with ***fre***[***s***].

Floating Test for software Divide X-form

[Category: Floating Point.Phased-In]
ftdiv BF,FRA,FRB



Let e_a be the unbiased exponent of the double-precision floating-point operand in register FRA.

Let e_b be the unbiased exponent of the double-precision floating-point operand in register FRB.

`fe_flag` is set to 1 if any of the following conditions occurs.

- The double-precision floating-point operand in register FRA is a NaN or an Infinity.
- The double-precision floating-point operand in register FRB is a Zero, a NaN, or an Infinity.
- e_b is less than or equal to -1022.
- e_b is greater than or equal to 1021.
- The double-precision floating-point operand in register FRA is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
- The double-precision floating-point operand in register FRA is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
- The double-precision floating-point operand in register FRA is not a zero and e_a is less than or equal to -970

Otherwise `fe_flag` is set to 0.

`fg_flag` is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register FRA is an Infinity.
- The double-precision floating-point operand in register FRB is a Zero, an Infinity, or a denormalized value.

Otherwise `fg_flag` is set to 0.

If the implementation guarantees a relative error of ***fre***[***s***][**.**] of less than or equal to 2^{-14}, then `fl_flag` is set to 1. Otherwise `fl_flag` is set to 0.

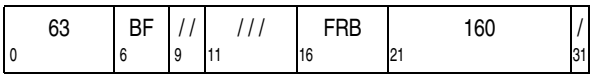
CR field BF is set to the value `fl_flag || fg_flag || fe_flag || 0b0`.

Special Registers Altered:

CR field BF

Floating Test for software Square Root X-form

[Category: Floating Point.Phased-In]
ftsqrt BF,FRB



Let e_b be the unbiased exponent of the double-precision floating-point operand in register FRB.

`fe_flag` is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register FRB is a zero, a NaN, or an infinity, or a negative value.
- e_b is less than or equal to -970.

Otherwise `fe_flag` is set to 0.

`fg_flag` is set to 1 if the following condition occurs.

- The double-precision floating-point operand in register FRB is a Zero, an Infinity, or a denormalized value.

Otherwise `fg_flag` is set to 0.

If the implementation guarantees a relative error of ***frsqrte***[***s***][**.**] of less than or equal to 2^{-14}, then `fl_flag` is set to 1. Otherwise `fl_flag` is set to 0.

CR field BF is set to the value `fl_flag || fg_flag || fe_flag || 0b0`.

Special Registers Altered:

CR field BF

Programming Note

ftdiv and ***ftsqrt*** are provided to accelerate software emulation of divide and square root operations, by performing the requisite special case checking. Software needs only a single branch, on FE=1 (in CR[BF]), to a special case handler. FG and FL may provide further acceleration opportunities.

4.6.6.2 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows.

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (***fmul***[***s***], followed by ***fadd***[***s***] or ***fsub***[***s***]). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

Floating Multiply-Add [Single] A-form

fmadd FRT,FRA,FRC,FRB (Rc=0)
fmadd. FRT,FRA,FRC,FRB (Rc=1)

63	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

fmadds FRT,FRA,FRC,FRB (Rc=0)
fmadds. FRT,FRA,FRC,FRB (Rc=1)

59	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

The operation

FRT ← [(FRA)×(FRC)] + (FRB)

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Multiply-Subtract [Single] A-form

fmsub FRT,FRA,FRC,FRB (Rc=0)
fmsub. FRT,FRA,FRC,FRB (Rc=1)

63	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

fmsubs FRT,FRA,FRC,FRB (Rc=0)
fmsubs. FRT,FRA,FRC,FRB (Rc=1)

59	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

The operation

FRT ← [(FRA)×(FRC)] - (FRB)

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Add [Single] A-form

fnmadd FRT,FRA,FRC,FRB (Rc=0)
fnmadd. FRT,FRA,FRC,FRB (Rc=1)

63	FRT	FRA	FRB	FRC	31	Rc
0	6	11	16	21	26	31

fnmadds FRT,FRA,FRC,FRB (Rc=0)
fnmadds. FRT,FRA,FRC,FRB (Rc=1)

59	FRT	FRA	FRB	FRC	31	Rc
0	6	11	16	21	26	31

The operation

FRT ← - ([(FRA)×(FRC)] + (FRB))

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Add* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Subtract [Single] A-form

fnmsub FRT,FRA,FRC,FRB (Rc=0)
fnmsub. FRT,FRA,FRC,FRB (Rc=1)

63	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

fnmsubs FRT,FRA,FRC,FRB (Rc=0)
fnmsubs. FRT,FRA,FRC,FRB (Rc=1)

59	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

The operation

FRT ← - ([(FRA)×(FRC)] - (FRB))

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Subtract* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

4.6.7 Floating-Point Rounding and Conversion Instructions

Programming Note

Examples of uses of these instructions to perform various conversions can be found in Section F.2, “Floating-Point Conversions [Category: Floating-Point]” on page 726.

4.6.7.1 Floating-Point Rounding Instruction

Floating Round to Single-Precision X-form

frsp FRT,FRB (Rc=0)
frsp. FRT,FRB (Rc=1)

63	FRT	///	FRB	12	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to single-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The rounding is described fully in Section A.1, “Floating-Point Round to Single-Precision Model” on page 685.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN
CR1 (if Rc=1)

4.6.7.2 Floating-Point Convert To/From Integer Instructions

Floating Convert To Integer Doubleword X-form

fctid FRT,FRB (Rc=0)
fctid. FRT,FRB (Rc=1)

63	FRT	///	FRB	814	Rc
0	6	11	16	21	31

Let `src` be the double-precision floating-point value in FRB.

If `src` is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and, if `src` is an SNaN, VXSNAN is set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode specified by $FPSCR_{RN}$.

If the rounded value is greater than 2^{63}-1, then the result is `0x7FFF_FFFF_FFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, then the result is `0x8000_0000_0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Doubleword with round toward Zero X-form

fctidz FRT,FRB (Rc=0)
fctidz. FRT,FRB (Rc=1)

63	FRT	///	FRB	815	Rc
0	6	11	16	21	31

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than 2^{63}-1, then the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, then the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Doubleword Unsigned X-form

[Category: Floating-Point.Phased-In]
fctidu FRT,FRB (Rc=0)
fctidu. FRT,FRB (Rc=1)

63	FRT	///	FRB	942	Rc
0	6	11	16	21	31

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by $FPSCR_{RN}$.

If the rounded value is greater than 2^{64}-1, then the result is 0xFFFF_FFFF_FFFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000_0000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Doubleword Unsigned with round toward Zero X-form

[Category: Floating-Point.Phased-In]

fctiduz FRT,FRB (Rc=0)

fctiduz. FRT,FRB (Rc=1)

63	FRT	///	FRB	943	Rc
0	6	11	16	21	31

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than 2^{64}-1, then the result is 0xFFFF_FFFF_FFFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000_0000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word X-form

fctiw FRT,FRB (Rc=0)

fctiw. FRT,FRB (Rc=1)

63	FRT	///	FRB	14	Rc
0	6	11	16	21	31

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by $FPSCR_{RN}$.

If the rounded value is greater than 2^{31}-1, then the result is 0x7FFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, then the result is 0x8000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word with round toward Zero X-form

fctiwz FRT,FRB (Rc=0)
fctiwz. FRT,FRB (Rc=1)

63	FRT	///	FRB	15	Rc
0	6	11	16	21	31

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than 2^{31}-1, then the result is 0x7FFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, then the result is 0x8000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word Unsigned X-form

[Category: Floating-Point.Phased-In]
fctiwu FRT,FRB (Rc=0)
fctiwu. FRT,FRB (Rc=1)

63	FRT	///	FRB	142	Rc
0	6	11	16	21	31

Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by $FPSCR_{RN}$.

If the rounded value is greater than 2^{32}-1, then the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0.0, then the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert To Integer Word Unsigned with round toward Zero X-form

[Category: Floating-Point.Phased-In]

fctiwuz	FRT,FRB	(Rc=0)
fctiwuz.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	143	Rc
0	6	11	16	21	31

Let `src` be the double-precision floating-point value in FRB.

If `src` is a NaN, then the result is 0x0000_0000, VXCVI is set to 1, and, if `src` is an SNaN, VXSNAN is set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than 2^{32}-1, then the result is `0xFFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0.0, then the result is `0x0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 689.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert From Integer Doubleword X-form

fcfid	FRT,FRB	(Rc=0)
fcfid.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	846	Rc
0	6	11	16	21	31

The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

$FPSCR_{FPRF}$ is set to the class and sign of the result. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

Programming Note

Converting a signed integer word to double-precision floating-point can be accomplished by loading the word from storage using *Load Float Word Algebraic Indexed* and then using ***fcfid***.

Floating Convert From Integer Doubleword Unsigned X-form

[Category: Floating-Point.Phased-In]

fcfidu FRT,FRB (Rc=0)

fcfidu. FRT,FRB (Rc=1)

63	FRT	///	FRB	974	Rc
0	6	11	16	21	31

The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

$FPSCR_{FPRF}$ is set to the class and sign of the result. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

Programming Note

Converting an unsigned integer word to double-precision floating-point can be accomplished by loading the word from storage using *Load Float Word and Zero Indexed* and then using ***fcfidu***.

Floating Convert From Integer Doubleword Single X-form

[Category: Floating-Point.Phased-In]

fcfids FRT,FRB (Rc=0)

fcfids. FRT,FRB (Rc=1)

59	FRT	///	FRB	846	Rc
0	6	11	16	21	31

The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

$FPSCR_{FPRF}$ is set to the class and sign of the result. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

Programming Note

Converting a signed integer word to single-precision floating-point can be accomplished by loading the word from storage using *Load Float Word Algebraic Indexed* and then using ***fcfids***.

Floating Convert From Integer Doubleword Unsigned Single X-form

[Category: Floating-Point.Phased-In]

fcfidus FRT,FRB (Rc=0)

fcfidus. FRT,FRB (Rc=1)

59	FRT	///	FRB	974	Rc
0	6	11	16	21	31

The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

$FPSCR_{FPRF}$ is set to the class and sign of the result. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

Programming Note

Converting a unsigned integer word to single-precision floating-point can be accomplished by loading the word from storage using *Load Float Word and Zero Indexed* and then using ***fcfidus***.

4.6.7.3 Floating Round to Integer Instructions

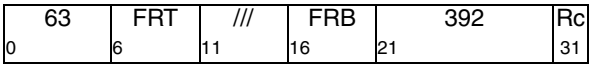
The *Floating Round to Integer* instructions provide direct support for rounding functions found in high level languages. For example, ***frin***, ***friz***, ***frip***, and ***frim*** implement C++ round(), trunc(), ceil(), and floor(), respectively. Note that ***frin*** does not implement the IEEE Round to Nearest function, which is often further described as “ties to even.” The rounding performed by these instructions is described fully in Section A.4, “Floating-Point Round to Integer Model” on page 694.

Programming Note

These instructions set $FPSCR_{FR\ FI}$ to 0b00 regardless of whether the result is inexact or rounded because there is a desire to preserve the value of $FPSCR_{XX}$. Furthermore, it is believed that most programs do not need to know whether these rounding operations produce inexact or rounded results. If it is necessary to determine whether the result is inexact or rounded, software must compare the result with the original source operand.

Floating Round to Integer Nearest X-form

frin FRT,FRB (Rc=0)
frin. FRT,FRB (Rc=1)



63	FRT	///	FRB	392	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to an integral value as follows, with the result placed into register FRT. If the sign of the operand is positive, (FRB) + 0.5 is truncated to an integral value, otherwise (FRB) - 0.5 is truncated to an integral value.

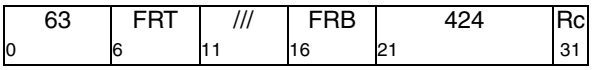
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Toward Zero X-form

friz FRT,FRB (Rc=0)
friz. FRT,FRB (Rc=1)



63	FRT	///	FRB	424	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward zero, and the result is placed into register FRT.

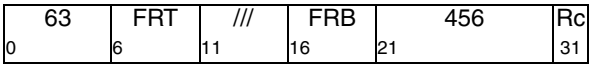
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Plus X-form

frip FRT,FRB (Rc=0)
frip. FRT,FRB (Rc=1)



63	FRT	///	FRB	456	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward +infinity, and the result is placed into register FRT.

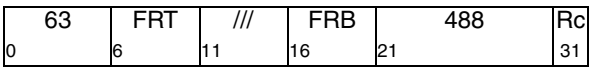
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Minus X-form

frim FRT,FRB (Rc=0)
frim. FRT,FRB (Rc=1)



63	FRT	///	FRB	488	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward -infinity, and the result is placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

4.6.8 Floating-Point Compare Instructions

The floating-point *Compare* instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to -0). The comparison can be ordered or unordered.

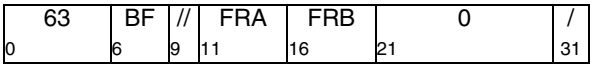
The comparison sets one bit in the designated CR field to 1 and the other three to 0. The FPCC is set in the same way.

The CR field and the FPCC are set as follows.

Bit	Name	Description
0	FL	(FRA) < (FRB)
1	FG	(FRA) > (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Floating Compare Unordered X-form

fcmpu BF,FRA,FRB




```
if (FRA) is a NaN or
   (FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                        c ← 0b0010
FPCC ← c
CR4×BF:4×BF+3 ← c
if (FRA) is an SNaN or
   (FRB) is an SNaN then
     VXSNAN ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

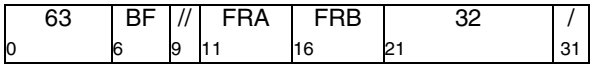
If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN

Floating Compare Ordered X-form

fcmpo BF,FRA,FRB




```
if (FRA) is a NaN or
   (FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                        c ← 0b0010
FPCC ← c
CR4×BF:4×BF+3 ← c
if (FRA) is an SNaN or
   (FRB) is an SNaN then
     VXSNAN ← 1
     if VE = 0 then VXVC ← 1
else if (FRA) is a QNaN or
   (FRB) is a QNaN then VXVC ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set and, if Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, then VXVC is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN VXVC

4.6.9 Floating-Point Select Instruction

Floating Select A-form

fsel FRT,FRA,FRC,FRB (Rc=0)
fsel. FRT,FRA,FRC,FRB (Rc=1)

63	FRT	FRA	FRB	FRC	23	Rc
0	6	11	16	21	26	31

```
if (FRA) ≥ 0.0 then FRT ← (FRC)
else FRT ← (FRB)
```

The floating-point operand in register FRA is compared to the value zero. If the operand is greater than or equal to zero, register FRT is set to the contents of register FRC. If the operand is less than zero or is a NaN, register FRT is set to the contents of register FRB. The comparison ignores the sign of zero (i.e., regards +0 as equal to -0).

Special Registers Altered:

CR1 (if Rc=1)

Programming Note

Examples of uses of this instruction can be found in Sections F.2, “Floating-Point Conversions [Category: Floating-Point]” on page 726 and F.3, “Floating-Point Selection [Category: Floating-Point]” on page 730.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section F.3.4, “Notes” on page 730.

4.6.10 Floating-Point Status and Control Register Instructions

Every *Floating-Point Status and Control Register* instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a *Floating-Point Status and Control Register* instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the *Floating-Point Status and Control Register* instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the *Floating-Point Status and Control Register* instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the Floating-Point Status and Control Register instruction is initiated.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the Floating-Point Status and Control Register instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the Floating-Point Status and Control Register instruction has completed.

(Floating-point Storage Access instructions are not affected.)

The instruction descriptions in this section refer to "FPSCR fields," where FPSCR field k is FPSCR bits 4xk:4xk+3.

Move From FPSCR X-form

mffs FRT (Rc=0)
mffs. FRT (Rc=1)

63	FRT	///	///	583	Rc
0	6	11	16	21	31

The contents of the FPSCR are placed into register FRT.

Special Registers Altered:

CR1 (if Rc=1)

Move to Condition Register from FPSCR X-form

mcrfs BF,BFA

63	BF	//	BFA	//	///	64	/
0	6	9	11	14	16	21	31

The contents of $FPSCR_{32:63}$ field BFA are copied to Condition Register field BF. All exception bits copied are set to 0 in the FPSCR. If the FX bit is copied, it is set to 0 in the FPSCR.

Special Registers Altered:

CR field BF
FX OX (if BFA=0)
UX ZX XX VXSNAN (if BFA=1)
VXISI VXIDI VXZDZ VXIMZ (if BFA=2)
VXVC (if BFA=3)
VXSOFT VXSQRT VXCVI (if BFA=5)

Move To FPSCR Field Immediate X-form

mtfsfi BF,U,W (Rc=0)
mtfsfi. BF,U,W (Rc=1)

63	BF	//	///	W	U	/	134	Rc
0	6	9	11	15	16	20	21	31

The value of the U field is placed into FPSCR field BF+8×(1-W).

$FPSCR_{FX}$ is altered only if BF = 0 and W = 0.

Special Registers Altered:

FPSCR field BF + 8×(1-W)
CR1 (if Rc=1)

Programming Note

mtfsfi serves as both a basic and an extended mnemonic. The Assembler will recognize a ***mtfsfi*** mnemonic with three operands as the basic form, and a ***mtfsfi*** mnemonic with two operands as the extended form. In the extended form the W operand is omitted and assumed to be 0.

Programming Note

When $FPSCR_{32:35}$ is specified, bits 32 (FX) and 35 (OX) are set to the values of U_0 and U_3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U_0 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 117, and not from $U_{1:2}$.

Move To FPSCR Fields XFL-form

mtfsf FLM,FRB,L,W (Rc=0)
mtfsf. FLM,FRB,L,W (Rc=1)

63	L	FLM	W	FRB	711	Rc
0	6	7	15	16	21	31

The FPSCR is modified as specified by the FLM, L, and W fields.

L = 0

The contents of register FRB are placed into the FPSCR under control of the W field and the field mask specified by FLM. W and the field mask identify the 4-bit fields affected. Let i be an integer in the range 0-7. If FLM_i=1 then FPSCR field k is set to the contents of the corresponding field of register FRB, where k = i+8×(1-W).

L = 1

The contents of register FRB are placed into the FPSCR.

$FPSCR_{FX}$ is not altered implicitly by this instruction.

Special Registers Altered:

FPSCR fields selected by mask, L, and W
CR1 (if Rc=1)

Programming Note

mtfsf serves as both a basic and an extended mnemonic. The Assembler will recognize a ***mtfsf*** mnemonic with four operands as the basic form, and a ***mtfsf*** mnemonic with two operands as the extended form. In the extended form the W and L operands are omitted and both are assumed to be 0.

Programming Note

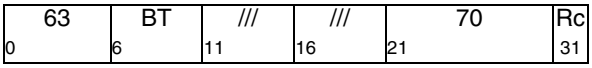
Updating fewer than eight fields of the FPSCR may have substantially poorer performance on some implementations than updating eight fields or all of the fields.

Programming Note

If L=1 or if L=0 and $FPSCR_{32:35}$ is specified, bits 32 (FX) and 35 (OX) are set to the values of $(FRB)_{32}$ and $(FRB)_{35}$ (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from $(FRB)_{32}$ and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 117, and not from $(FRB)_{33:34}$.

Move To FPSCR Bit 0 X-form

mtfsb0 BT (Rc=0)
mtfsb0. BT (Rc=1)



Bit BT+32 of the FPSCR is set to 0.

Special Registers Altered:

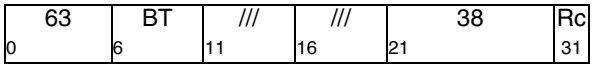
FPSCR bit BT+32
CR1 (if Rc=1)

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 X-form

mtfsb1 BT (Rc=0)
mtfsb1. BT (Rc=1)



Bit BT+32 of the FPSCR is set to 1.

Special Registers Altered:

FPSCR bits BT+32 and FX
CR1 (if Rc=1)

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly set.

Chapter 5. Vector Facility [Category: Vector]

5.1 Vector Facility Overview

This chapter describes the registers and instructions that make up the Vector Facility.

5.2 Chapter Conventions

5.2.1 Description of Instruction Operation

The following notation, in addition to that described in Section 1.3.2, is used in this chapter. Additional RTL functions are described in Appendix C.

x.bit[y]
Return the contents of bit y of x.

x.bit[y:z]
Return the contents of bits y:z of x.

x.byte[y]
Return the contents of byte element y of x.

x.byte[y:z]
Return the contents of byte elements y:z of x.

x.hword[y]
Return the contents of halfword element y of x.

x.hword[y:z]
Return the contents of halfword elements y:z of x.

x.word[y]
Return the contents of word element y of x.

x.word[y:z]
Return the contents of word element y:z of x.

x.dword[y]
Return the contents of doubleword element y of x.

x.dword[y:z]
Return the contents of doubleword elements y:z of x.

x ? y : z
if the value of x is true, then the value of y, otherwise the value z.

$+_{int}$
Integer addition.

$+_{fp}$
Floating-point addition.

$-_{fp}$
Floating-point subtraction.

$\times_{sui}$
Multiplication of a signed-integer (first operand) by an unsigned-integer (second operand).

$\times_{fp}$
Floating-point multiplication.

$=_{int}$
Integer equals relation.

$=_{fp}$
Floating-point equals relation.

$<_{ui}$, $\leq_{ui}$, $>_{ui}$, $\geq_{ui}$
Unsigned-integer comparison relations.

$<_{si}$, $\leq_{si}$, $>_{si}$, $\geq_{si}$
Signed-integer comparison relations.

$<_{fp}$, $\leq_{fp}$, $>_{fp}$, $\geq_{fp}$
Floating-point comparison relations.

LENGTH(x)
Length of x, in bits. If x is the word “element”, LENGTH(x) is the length, in bits, of the element implied by the instruction mnemonic.

x << y

Result of shifting x left by y bits, filling vacated bits with zeros.

```
b ← LENGTH(x)
result ← (y < b) ? (x_{y:b-1} ||^{y}0) : ^{b}0
```

x >>$_{ui}$ y

Result of shifting x right by y bits, filling vacated bits with zeros.

```
b ← LENGTH(x)
result ← (y < b) ? (^{y}0 || x_{0:(b-y)-1}) : ^{b}0
```

x >> y

Result of shifting x right by y bits, filling vacated bits with copies of bit 0 (sign bit) of x.

```
b ← LENGTH(x)
result ← (y<b) ? (^{y}x_0 ||x_{0:(b-y)-1}) : ^{b}x_0
```

x <<< y

Result of rotating x left by y bits.

```
b ← LENGTH(x)
result ← x_{y:b-1} || x_{0:y-1}
```

x >>> y

Returns the contents of x rotated right by y bits.

Chop(x, y)

Result of extending the right-most y bits of x on the left with zeros.

```
result ← x & ((1<<y)-1)
```

EXTZ(x)

Result of extending x on the left with zeros.

```
b ← LENGTH(x)
result ← x & ((1<<b)-1)
```

Clamp(x, y, z)

x is interpreted as a signed integer. If the value of x is less than y, then the value y is returned, else if the value of x is greater than z, the value z is returned, else the value x is returned.

```
if (x < y) then
   result ← y
   VSCR_SAT ← 1
else if (x > z) then
   result ← z
   VSCR_SAT ← 1
else result ← x
```

InvMixColumns(x)

```
do c = 0 to 3
   result.word[c].byte[0] = 0x0E•x.word[c].byte[0] ^ 0x0B•x.word[c].byte[1] ^ 0x0D•x.word[c].byte[2] ^ 0x09•x.word[c].byte[3]
   result.word[c].byte[1] = 0x09•x.word[c].byte[0] ^ 0x0E•x.word[c].byte[1] ^ 0x0B•x.word[c].byte[2] ^ 0x0D•x.word[c].byte[3]
   result.word[c].byte[2] = 0x0D•x.word[c].byte[0] ^ 0x09•x.word[c].byte[1] ^ 0x0E•x.word[c].byte[2] ^ 0x0B•x.word[c].byte[3]
   result.word[c].byte[3] = 0x0B•x.word[c].byte[0] ^ 0x0D•x.word[c].byte[1] ^ 0x09•x.word[c].byte[2] ^ 0x0E•x.word[c].byte[3]
end
return(result);
```

where "•" is a GF(2^8) multiply, a binary polynomial multiplication reduced by modulo `0x11B`.

The GF(2^8) multiply of `0x09`•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[3]
product.bit[1] = x.bit[1] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[2] ^ x.bit[5] ^ x.bit[0] ^ x.bit[1]
product.bit[3] = x.bit[3] ^ x.bit[6] ^ x.bit[1] ^ x.bit[2]
product.bit[4] = x.bit[4] ^ x.bit[7] ^ x.bit[0] ^ x.bit[2]
product.bit[5] = x.bit[5] ^ x.bit[0] ^ x.bit[1]
product.bit[6] = x.bit[6] ^ x.bit[1] ^ x.bit[2]
product.bit[7] = x.bit[7] ^ x.bit[2]
```

The GF(2^8) multiply of `0x0B`•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[1] ^ x.bit[3]
product.bit[1] = x.bit[1] ^ x.bit[2] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[2] ^ x.bit[3] ^ x.bit[5] ^ x.bit[0] ^ x.bit[1]
product.bit[3] = x.bit[3] ^ x.bit[4] ^ x.bit[6] ^ x.bit[0] ^ x.bit[1] ^ x.bit[2]
product.bit[4] = x.bit[4] ^ x.bit[5] ^ x.bit[7] ^ x.bit[2]
product.bit[5] = x.bit[5] ^ x.bit[6] ^ x.bit[0] ^ x.bit[1]
product.bit[6] = x.bit[6] ^ x.bit[7] ^ x.bit[0] ^ x.bit[1] ^ x.bit[2]
product.bit[7] = x.bit[7] ^ x.bit[0] ^ x.bit[2]
```

The GF(2^8) multiply of 0x0D•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[2] ^ x.bit[3]
product.bit[1] = x.bit[1] ^ x.bit[3] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[2] ^ x.bit[4] ^ x.bit[5] ^ x.bit[1]
product.bit[3] = x.bit[3] ^ x.bit[5] ^ x.bit[6] ^ x.bit[0] ^ x.bit[2]
product.bit[4] = x.bit[4] ^ x.bit[6] ^ x.bit[7] ^ x.bit[0] ^ x.bit[1] ^ x.bit[2]
product.bit[5] = x.bit[5] ^ x.bit[7] ^ x.bit[1]
product.bit[6] = x.bit[6] ^ x.bit[0] ^ x.bit[2]
product.bit[7] = x.bit[7] ^ x.bit[1] ^ x.bit[2]
```

The GF(2^8) multiply of 0x0E•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[1] ^ x.bit[2] ^ x.bit[3]
product.bit[1] = x.bit[2] ^ x.bit[3] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[3] ^ x.bit[4] ^ x.bit[5] ^ x.bit[1]
product.bit[3] = x.bit[4] ^ x.bit[5] ^ x.bit[6] ^ x.bit[2]
product.bit[4] = x.bit[5] ^ x.bit[6] ^ x.bit[7] ^ x.bit[1] ^ x.bit[2]
product.bit[5] = x.bit[6] ^ x.bit[7] ^ x.bit[1]
product.bit[6] = x.bit[7] ^ x.bit[2]
product.bit[7] = x.bit[0] ^ x.bit[1] ^ x.bit[2]
```

InvShiftRows(x)

```
result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[3].byte[1]
result.word[1].byte[1] = x.word[0].byte[1]
result.word[2].byte[1] = x.word[1].byte[1]
result.word[3].byte[1] = x.word[2].byte[1]

result.word[0].byte[2] = x.word[2].byte[2]
result.word[1].byte[2] = x.word[3].byte[2]
result.word[2].byte[2] = x.word[0].byte[2]
result.word[3].byte[2] = x.word[1].byte[2]

result.word[0].byte[3] = x.word[1].byte[3]
result.word[1].byte[3] = x.word[2].byte[3]
result.word[2].byte[3] = x.word[3].byte[3]
result.word[3].byte[3] = x.word[0].byte[3]

return(result)
```

InvSubBytes(x)

```
InvSBOX.byte[256] = { 0x52,0x09,0x6A,0xD5,0x30,0x36,0xA5,0x38,0xBF,0x40,0xA3,0x9E,0x81,0xF3,0xD7,0xFB,
                      0x7C,0xE3,0x39,0x82,0x9B,0x2F,0xFF,0x87,0x34,0x8E,0x43,0x44,0xC4,0xDE,0xE9,0xCB,
                      0x54,0x7B,0x94,0x32,0xA6,0xC2,0x23,0x3D,0xEE,0x4C,0x95,0x0B,0x42,0xFA,0xC3,0x4E,
                      0x08,0x2E,0xA1,0x66,0x28,0xD9,0x24,0xB2,0x76,0x5B,0xA2,0x49,0x6D,0x8B,0xD1,0x25,
                      0x72,0xF8,0xF6,0x64,0x86,0x68,0x98,0x16,0xD4,0xA4,0x5C,0xCC,0x5D,0x65,0xB6,0x92,
                      0x6C,0x70,0x48,0x50,0xFD,0xED,0xB9,0xDA,0x5E,0x15,0x46,0x57,0xA7,0x8D,0x9D,0x84,
                      0x90,0xD8,0xAB,0x00,0x8C,0xBC,0xD3,0x0A,0xF7,0xE4,0x58,0x05,0xB8,0xB3,0x45,0x06,
                      0xD0,0x2C,0x1E,0x8F,0xCA,0x3F,0x0F,0x02,0xC1,0xAF,0xBD,0x03,0x01,0x13,0x8A,0x6B,
                      0x3A,0x91,0x11,0x41,0x4F,0x67,0xDC,0xEA,0x97,0xF2,0xCF,0xCE,0xF0,0xB4,0xE6,0x73,
                      0x96,0xAC,0x74,0x22,0xE7,0xAD,0x35,0x85,0xE2,0xF9,0x37,0xE8,0x1C,0x75,0xDF,0x6E,
                      0x47,0xF1,0x1A,0x71,0x1D,0x29,0xC5,0x89,0x6F,0xB7,0x62,0x0E,0xAA,0x18,0xBE,0x1B,
                      0xFC,0x56,0x3E,0x4B,0xC6,0xD2,0x79,0x20,0x9A,0xDB,0xC0,0xFE,0x78,0xCD,0x5A,0xF4,
                      0x1F,0xDD,0xA8,0x33,0x88,0x07,0xC7,0x31,0xB1,0x12,0x10,0x59,0x27,0x80,0xEC,0x5F,
                      0x60,0x51,0x7F,0xA9,0x19,0xB5,0x4A,0x0D,0x2D,0xE5,0x7A,0x9F,0x93,0xC9,0x9C,0xEF,
                      0xA0,0xE0,0x3B,0x4D,0xAE,0x2A,0xF5,0xB0,0xC8,0xEB,0xBB,0x3C,0x83,0x53,0x99,0x61,
                      0x17,0x2B,0x04,0x7E,0xBA,0x77,0xD6,0x26,0xE1,0x69,0x14,0x63,0x55,0x21,0x0C,0x7D }

do i = 0 to 15
   result.byte[i] = InvSBOX.byte[x.byte[i]]
end
return(result)
```

MixColumns(x)

```
do c = 0 to 3
   result.word[c].byte[0] = 0x02•x.word[c].byte[0] ^ 0x03•x.word[c].byte[1] ^      x.word[c].byte[2] ^      x.word[c].byte[3]
   result.word[c].byte[1] =      x.word[c].byte[0] ^ 0x02•x.word[c].byte[1] ^ 0x03•x.word[c].byte[2] ^      x.word[c].byte[3]
   result.word[c].byte[2] =      x.word[c].byte[0] ^      x.word[c].byte[1] ^ 0x02•x.word[c].byte[2] ^ 0x03•x.word[c].byte[3]
   result.word[c].byte[3] = 0x03•x.word[c].byte[0] ^      x.word[c].byte[1] ^      x.word[c].byte[2] ^ 0x02•x.word[c].byte[3]
end
return(result)
```

The GF(2^8) multiply of `0x02•x` can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[1]
product.bit[1] = x.bit[2]
product.bit[2] = x.bit[3]
product.bit[3] = x.bit[4] ^ x.bit[0]
product.bit[4] = x.bit[5] ^ x.bit[0]
product.bit[5] = x.bit[6]
product.bit[6] = x.bit[7] ^ x.bit[0]
product.bit[7] = x.bit[0]
```

The GF(2^8) multiply of `0x03•x` can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[1]
product.bit[1] = x.bit[1] ^ x.bit[2]
product.bit[2] = x.bit[2] ^ x.bit[3]
product.bit[3] = x.bit[3] ^ x.bit[4] ^ x.bit[0]
product.bit[4] = x.bit[4] ^ x.bit[5] ^ x.bit[0]
product.bit[5] = x.bit[5] ^ x.bit[6]
product.bit[6] = x.bit[6] ^ x.bit[7] ^ x.bit[0]
product.bit[7] = x.bit[7] ^ x.bit[0]
```

ShiftRows(x)

```
result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[1].byte[1]
result.word[1].byte[1] = x.word[2].byte[1]
result.word[2].byte[1] = x.word[3].byte[1]
result.word[3].byte[1] = x.word[0].byte[1]

result.word[0].byte[2] = x.word[2].byte[2]
result.word[1].byte[2] = x.word[3].byte[2]
result.word[2].byte[2] = x.word[0].byte[2]
result.word[3].byte[2] = x.word[1].byte[2]

result.word[0].byte[3] = x.word[3].byte[3]
result.word[1].byte[3] = x.word[0].byte[3]
result.word[2].byte[3] = x.word[1].byte[3]
result.word[3].byte[3] = x.word[2].byte[3]

return(result)
```

Signed_BCD_Add(x,y,z)

Let x and y be 31-digit signed decimal values.

Performs a signed decimal addition of x and y.

If the unbounded result is equal to zero, `eq_flag` is set to `1`. Otherwise, `eq_flag` is set to `0`.
If the unbounded result is greater than zero, `gt_flag` is set to `1`. Otherwise, `gt_flag` is set to `0`.
If the unbounded result is less than zero, `lt_flag` is set to `1`. Otherwise, `lt_flag` is set to `0`.

If the magnitude of the unbounded result is greater than 10^{31}-1, `ox_flag` is set to `1`. Otherwise, `ox_flag` is set to `0`.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to `0b1100` if `z=0`.
If the unbounded result is greater than or equal to zero, the sign code of the result is set to `0b1111` if `z=1`.
If the unbounded result is less than zero, the sign code of the result is set to `0b1101`.

The low-order 31 digits of the unbounded result magnitude concatented with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and `inv_flag` is set to 1 and `lt_flag`, `gt_flag` and `eq_flag` are set to 0. Otherwise, `inv_flag` is set to 0.

Signed_BCD_Subtract(x,y,z)

Let x and y be 31-digit signed decimal values.

Performs a signed decimal subtract of y from x.

If the unbounded result is equal to zero, eq_flag is set to 1. Otherwise, eq_flag is set to 0.
If the unbounded result is greater than zero, gt_flag is set to 1. Otherwise, gt_flag is set to 0.
If the unbounded result is less than zero, lt_flag is set to 1. Otherwise, lt_flag is set to 0.

If the magnitude of the unbounded result is greater than 10^{31}-1, ox_flag is set to 1. Otherwise, ox_flag is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1100 if z=0.
If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1111 if z=1.
If the unbounded result is less than zero, the sign code of the result is set to 0b1101.

The low-order 31 digits of the unbounded result magnitude concatented with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and inv_flag is set to 1 and lt_flag, gt_flag and eq_flag are set to 0. Otherwise, inv_flag is set to 0.

SubBytes(x)

```
SBOX.byte[0:255] = { 0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
                     0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
                     0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
                     0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
                     0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
                     0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
                     0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
                     0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
                     0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
                     0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
                     0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
                     0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
                     0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
                     0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
                     0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
                     0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16 }

do i = 0 to 15
   result.byte[i] = SBOX.byte[x.byte[i]]
end
return(result)
```

RoundToSPIntCeil(x)

The value x if x is a single-precision floating-point integer; otherwise the smallest single-precision floating-point integer that is greater than x.

RoundToSPIntFloor(x)

The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x.

RoundToSPIntNear(x)

The value x if x is a single-precision floating-point integer; otherwise the single-precision floating-point integer that is nearest in value to x (in case of a tie, the even single-precision floating-point integer is used).

RoundToSPIntTrunc(x)

The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x if x>0, or the smallest single-precision floating-point integer that is greater than x if x<0.

RoundToNearSP(x)

The single-precision floating-point number that is nearest in value to the infinitely-precise floating-point intermediate result x (in case of a tie, the single-precision floating-point value with the least-significant bit equal to 0 is used).

ReciprocalEstimateSP(x)

A single-precision floating-point estimate of the reciprocal of the single-precision floating-point number x.

ReciprocalSquareRootEstimateSP(x)

A single-precision floating-point estimate of the reciprocal of the square root of the single-precision floating-point number x.

LogBase2EstimateSP(x)

A single-precision floating-point estimate of the base 2 logarithm of the single-precision floating-point number x.

Power2EstimateSP(x)

A single-precision floating-point estimate of the 2 raised to the power of the single-precision floating-point number x.

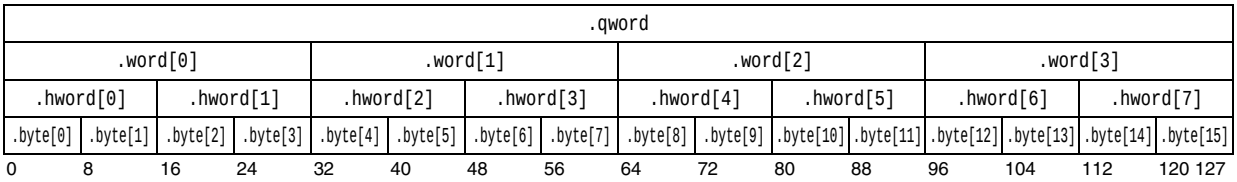


Figure 63. Vector Register elements

5.3 Vector Facility Registers

5.3.1 Vector Registers

There are 32 Vector Registers (VRs), each containing 128 bits. See Figure 64. All computations and other data manipulation are performed on data residing in Vector Registers, and results are placed into a VR.

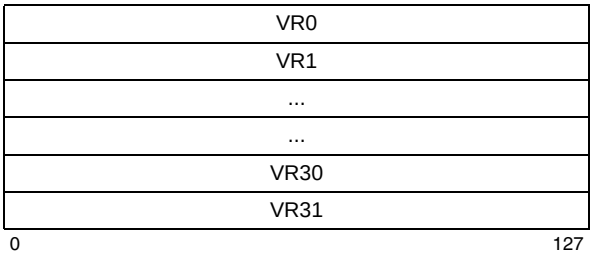


Figure 64. Vector Registers

Depending on the instruction, the contents of a Vector Register are interpreted as a sequence of equal-length elements (bytes, halfwords, or words) or as a quadword. Each of the elements is aligned at its natural boundary within the Vector Register, as shown in Figure 63. Many instructions perform a given operation in parallel on all elements in a Vector Register. Depending on the instruction, a byte, halfword, or word element can be interpreted as a signed-integer, an unsigned-integer, or a logical value; a word element can also be interpreted as a single-precision floating-point value. In the instruction descriptions, phrases like “signed-integer word element” are used as shorthand for “word element, interpreted as a signed-integer”.

Load and *Store* instructions are provided that transfer a byte, halfword, word, or quadword between storage and a Vector Register.

5.3.2 Vector Status and Control Register

The Vector Status and Control Register (VSCR) is a special 32-bit register (not an SPR) that is read and written in a manner similar to the FPSCR in the Power ISA scalar floating-point unit. Special instructions (***mfvscr*** and ***mtvscr***) are provided to move the VSCR from and to a vector register. When moved to or from a vector register, the 32-bit VSCR is right justified in the 128-bit vector register. When moved to a vector register, bits 0:95 of the vector register are cleared (set to 0).

Figure 65. Vector Status and Control Register

The bit definitions for the VSCR are as follows.

Bit(s)	Description
96:110	Reserved
111	***Vector Non-Java Mode*** (NJ) This bit controls how denormalized values are handled by *Vector Floating-Point* instructions. 0 Denormalized values are handled as specified by Java and the IEEE standard; see Section 5.6.1. 1 If an element in a source VR contains a denormalized value, the value 0 is used instead. If an instruction causes an Underflow Exception, the corresponding element in the target VR is set to 0. In both cases the 0 has the same sign as the denormalized or underflowing value.
112:126	Reserved
127	***Vector Saturation*** (SAT) Every vector instruction having “Saturate” in its name implicitly sets this bit to 1 if any result of that instruction “saturates”; see Section 5.8. ***mtvscr*** can alter this bit explicitly. This bit is sticky; that is, once set to 1 it remains set to 1 until it is set to 0 by an ***mtvscr*** instruction.

After the ***mfvscr*** instruction executes, the result in the target vector register will be architecturally precise. That is, it will reflect all updates to the SAT bit that could have been made by vector instructions logically preceding it in the program flow, and further, it will not reflect any SAT updates that may be made to it by vector instructions logically following it in the program flow. To implement this, processors may choose to make the ***mfvscr*** instruction executEXTZ(ion serializing within the vector unit, meaning that it will stall vector instruction execution until all preceding vector instructions are complete and have updated the architectural machine state. This is permitted in order to simplify implementation of the sticky status bit (SAT) which would otherwise be difficult to implement in an out-of-order execution machine. The implication of this is that reading the VSCR can be much slower than typical Vector instructions, and therefore care must be taken in reading it, as advised in Section 5.5.1, to avoid performance problems.

The ***mtvscr*** is context synchronizing. This implies that all Vector instructions logically preceding an ***mtvscr*** in the program flow will execute in the architectural context (NJ mode) that existed prior to completion of the ***mtvscr***, and that all instructions logically following the ***mtvscr*** will execute in the new context (NJ mode) established by the ***mtvscr***.

5.3.3 VR Save Register

The VR Save Register (VRSAVE) is a 32-bit register in the fixed-point processor provided for application and operating system use; see Section 3.2.3.

Programming Note

The VRSAVE register can be used to indicate which VRs are currently being used by a program. If this is done, the operating system could save only those VRs when an “interrupt” occurs (see Book III), and could restore only those VRs when resuming the interrupted program.

If this approach is taken it must be applied rigorously; if a program fails to indicate that a given VR is in use, software errors may occur that will be difficult to detect and correct because they are timing-dependent.

Some operating systems save and restore VRSAVE only for programs that also use other vector registers.

5.4 Vector Storage Access Operations

The *Vector Storage Access* instructions provide the means by which data can be copied from storage to a Vector Register or from a Vector Register to storage. Instructions are provided that access byte, halfword, word, and quadword storage operands. These instructions differ from the fixed-point and floating-point *Storage Access* instructions in that vector storage operands are assumed to be aligned, and vector storage accesses are performed as if the appropriate number of low-order bits of the specified effective address (EA) were zero. For example, the low-order bit of EA is ignored for halfword *Vector Storage Access* instructions, and the low-order four bits of EA are ignored for quadword *Vector Storage Access* instructions. The effect is to load or store the storage operand of the specified length that contains the byte addressed by EA.

If a storage operand is unaligned, additional instructions must be used to ensure that the operand is correctly placed in a Vector Register or in storage. Instructions are provided that shift and merge the contents of two Vector Registers, such that an unaligned quadword storage operand can be copied between storage and the Vector Registers in a relatively efficient manner.

As shown in Figure 63, the elements in Vector Registers are numbered; the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15. The numbering affects the values that must be placed into the permute control vector for the *Vector Permute* instruction in order for that instruction to achieve the desired effects, as illustrated by the examples in the following subsections.

A vector quadword *Load* instruction for which the effective address (EA) is quadword-aligned places the byte in storage addressed by EA into byte element 0 of the target Vector Register, the byte in storage addressed by EA+1 into byte element 1 of the target Vector Register, etc. Similarly, a vector quadword *Store* instruction for which the EA is quadword-aligned places the contents of byte element 0 of the source Vector Register into the byte in storage addressed by EA, the contents of byte element 1 of the source Vector Register into the byte in storage addressed by EA+1, etc.

Figure 66 shows an aligned quadword in storage. Figure 67 shows the result of loading that quadword into a Vector Register or, equivalently, shows the contents that must be in a Vector Register if storing that Vector Register is to produce the storage contents shown in Figure 66.

When an aligned byte, halfword, or word storage operand is loaded into a Vector Register, the element (byte, halfword, or word respectively) that receives the data is the element that would have received the data had the entire aligned quadword containing the storage operand addressed by EA been loaded. Similarly, when a byte, halfword, or word element in a Vector Register is stored into an aligned storage operand (byte, halfword, or word respectively), the element selected to be stored is the element that would have been stored into the storage operand addressed by EA had the entire Vector Register been stored to the aligned quadword containing the storage operand addressed by EA. (Byte storage operands are always aligned.)

For aligned byte, halfword, and word storage operands, if the corresponding element number is known when the program is written, the appropriate *Vector Splat* and *Vector Permute* instructions can be used to copy or replicate the data contained in the storage operand after loading the operand into a Vector Register. An example of this is given in the Programming Note for *Vector Splat*; see page 196. Another example is to replicate the element across an entire Vector Register before storing it into an arbitrary aligned storage operand of the same length; the replication ensures that the correct data are stored regardless of the offset of the storage operand in its aligned quadword in storage.

00	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 66. Aligned quadword storage operand

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 67. Vector Register contents for aligned quadword Load or Store

00												00	01	02	03	04
10	05	06	07	08	09	0A	0B	0C	0D	0E	0F					
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 68. Unaligned quadword storage operand

Vhi												00	01	02	03	04
Vlo	05	06	07	08	09	0A	0B	0C	0D	0E	0F					

Vt,Vs	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	0															15

Figure 69. Vector Register contents

5.4.1 Accessing Unaligned Storage Operands

Figure 68 shows an unaligned quadword storage operand that spans two aligned quadwords. In the remainder of this section, the aligned quadword that contains the most significant bytes of the unaligned quadword is called the most significant quadword (MSQ) and the aligned quadword that contains the least significant bytes of the unaligned quadword is called the least significant quadword (LSQ). Because the V*ector Storage Access* instructions ignore the low-order bits of the effective address, the unaligned quadword cannot be transferred between storage and a Vector Register using a single instruction. The remainder of this section gives examples of accessing unaligned quadword storage operands. Similar sequences can be used to access unaligned halfword and word storage operands.

Programming Note

The sequence of instructions given below is one approach that can be used to load the unaligned quadword shown in Figure 68 into a Vector Register. In Figure 69 Vhi and Vlo are the Vector Registers that will receive the most significant quadword and least significant quadword respectively. VRT is the target Vector Register.

After the two quadwords have been loaded into Vhi and Vlo, using *Load Vector Indexed* instructions, the alignment is performed by shifting the 32-byte quantity Vhi || Vlo left by an amount determined by the address of the first byte of the desired data. The shifting is done using a *Vector Permute* instruction for which the permute control vector is generated by a *Load Vector for Shift Left* instruction. The *Load Vector* for *Shift Left* instruction uses the same address specification as the *Load Vector Indexed* instruction that loads the Vhi register; this is the address of the desired unaligned quadword.

The following sequence of instructions copies the unaligned quadword storage operand into register Vt.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb       # load MSQ
lvsl   Vp,0,Rb        # set permute control vector
addi   Rb,Rb,16       # address of LSQ
lvx    Vlo,0,Rb       # load LSQ
vperm  Vt,Vhi,Vlo,Vp  # align the data
```

The procedure for storing an unaligned quadword is essentially the reverse of the procedure for loading one. However, a read-modify-write sequence is required that inserts the source quadword into two aligned quadwords in storage. The quadword to be stored is assumed to be in Vs; see Figure 69 The contents of Vs are shifted right and split into two parts, each of which is merged (using a Vector Select instruction) with the current contents of the two aligned quadwords (MSQ and LSQ) that will contain the most significant bytes and least significant bytes, respectively, of the unaligned quadword. The resulting two quadwords are stored using *Store Vector Indexed* instructions. A *Load Vector for Shift Right* instruction is used to generate the permute control vector that is used for the shifting. A single register is used for the "shifted" contents; this is possible because the "shifting" is done by means of a right rotation. The rotation is accomplished by specifying Vs for both components of the *Vector Permute* instruction. In addition, the same permute control vector is used on a sequence of 1s and 0s to generate the mask used by the *Vector Select* instructions that do the merging.

The following sequence of instructions copies the contents of Vs into an unaligned quadword in storage.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx       Vhi,0,Rb     # load current MSQ
lvsr      Vp,0,Rb      # set permute control vector
addi      Rb,Rb,16     # address of LSQ
lvx       Vlo,0,Rb     # load current LSQ
vspltisb  V1s,-1       # generate the select mask bits
vspltisb  V0s,0
vperm     Vmask,V0s,V1s,Vp # generate the select mask
vperm     Vs,Vs,Vs,Vp      # right rotate the data
vsel      Vlo,Vs,Vlo,Vmask # insert LSQ component
vsel      Vhi,Vhi,Vs,Vmask # insert MSQ component
stvx      Vlo,0,Rb         # store LSQ
addi      Rb,Rb,-16        # address of MSQ
stvx      Vhi,0,Rb         # store MSQ
```

5.5 Vector Integer Operations

Many of the instructions that produce fixed-point integer results have the potential to compute a result value that cannot be represented in the target format. When this occurs, this unrepresentable intermediate value is converted to a representable result value using one of the following methods.

1. The high-order bits of the intermediate result that do not fit in the target format are discarded. This method is used by instructions having names that include the word "Modulo".

2. The intermediate result is converted to the nearest value that is representable in the target format (i.e., to the minimum or maximum representable value, as appropriate). This method is used by instructions having names that include the word "Saturate". An intermediate result that is forced to the minimum or maximum representable value as just described is said to "saturate".

   An instruction for which an intermediate result saturates causes $VSCR_{SAT}$ to be set to 1; see Section 5.3.2.

3. If the intermediate result includes non-zero fraction bits it is rounded up to the nearest fixed-point integer value. This method is used by the six *Vector Average Integer* instructions and by the *Vector Multiply-High-Round-Add Signed Halfword Saturate* instruction. The latter instruction then uses method 2, if necessary.

Programming Note

Because $VSCR_{SAT}$ is sticky, it can be used to detect whether any instruction in a sequence of "Saturate"-type instructions produced an inexact result due to saturation. For example, the contents of the VSCR can be copied to a VR (***mfvscr***), bits other than the SAT bit can be cleared in the VR (***vand*** with a constant), the result can be compared to zero setting CR6 (***vcmpequb***.), and a branch can be taken according to whether $VSCR_{SAT}$ was set to 1 (*Branch Conditional* that tests CR field 6).

Testing $VSCR_{SAT}$ after each "Saturate"-type instruction would degrade performance considerably. Alternative techniques include the following:

– Retain sufficient information at "checkpoints" that the sequence of computations performed between one checkpoint and the next can be redone (more slowly) in a manner that detects exactly when saturation occurs. Test $VSCR_{SAT}$ only at checkpoints, or when redoing a sequence of computations that saturated.

– Perform intermediate computations using an element length sufficient to prevent saturation, and then use a *Vector Pack Integer Saturate* instruction to pack the final result to the desired length. (Vector Pack Integer Saturate causes results to saturate if necessary, and sets $VSCR_{SAT}$ to 1 if any result saturates.)

5.5.1 Integer Saturation

Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero (0) on underflow and to the maximum positive integer value (2^n-1, e.g. 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number (-2^{n-1}, e.g. -128 for byte fields) on underflow, and to the largest representable positive number (2^{n-1}-1, e.g. +127 for byte fields) on overflow.

In most cases, the simple maximum/minimum saturation performed by the vector instructions is adequate. However, sometimes, e.g. in the creation of very high quality images, more complex saturation functions must be applied. To support this, the Vector facility provides a mechanism for detecting that saturation has occurred. The VSCR has a bit, the SAT bit, which is set to a one (1) anytime any field in a saturating instruction saturates. The SAT bit can only be cleared by explicitly writing zero to it. Thus SAT accumulates a summary result of any integer overflow or underflow that occurs on a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned 0+0=0 and unsigned byte 1+254=255, are not considered saturation conditions and do not cause SAT to be set.

The SAT bit can be set by the following types of instructions:

- Move To VSCR
- Vector Add Integer with Saturation
- Vector Subtract Integer with Saturation
- Vector Multiply-Add Integer with Saturation
- Vector Multiply-Sum with Saturation
- Vector Sum-Across with Saturation
- Vector Pack with Saturation
- Vector Convert to Fixed-point with Saturation

Note that only instructions that explicitly call for “saturation” can set SAT. “Modulo” integer instructions and floating-point arithmetic instructions never set SAT.

Programming Note

The SAT state can be tested and used to alter program flow by moving the VSCR to a vector register (with ***mfvscr***), then masking out bits 0:126 (to clear undefined and reserved bits) and performing a vector compare equal-to unsigned byte w/record (***vcmpequb***.) with zero to get a testable value into the condition register for consumption by a subsequent branch.

Since ***mfvscr*** will be slow compared to other Vector instructions, reading and testing SAT after each instruction would be prohibitively expensive. Therefore, software is advised to employ strategies that minimize checking SAT. For example: checking SAT periodically and backtracking to the last checkpoint to identify exactly which field in which instruction saturated; or, working in an element size sufficient to prevent any overflow or underflow during intermediate calculations, then packing down to the desired element size as the final operation (the vector pack instruction saturates the results and updates SAT when a loss of significance is detected).

5.6 Vector Floating-Point Operations

5.6.1 Floating-Point Overview

Unless $VSCR_{NJ}$=1 (see Section 5.3.2), the floating-point model provided by the Vector Facility conforms to The Java Language Specification (hereafter referred to as "Java"), which is a subset of the default environment specified by the IEEE standard (i.e., by ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic"). For aspects of floating-point behavior that are not defined by Java but are defined by the IEEE standard, vector floating-point conforms to the IEEE standard. For aspects of floating-point behavior that are defined neither by Java nor by the IEEE standard but are defined by the "C9X Floating-Point Proposal" (hereafter referred to as "C9X"), vector floating-point conforms to C9X.

The single-precision floating-point data format, value representations, and computational models defined in Chapter 4. "Floating-Point Facility [Category: Floating-Point]" on page 115 apply to vector floating-point except as follows.

– In general, no status bits are set to reflect the results of floating-point operations. The only exception is that $VSCR_{SAT}$ may be set by the *Vector Convert To Fixed-Point Word* instructions.

– With the exception of the two *Vector Convert To Fixed-Point Word* instructions and three of the four *Vector Round to Floating-Point Integer* instructions, all vector floating-point instructions that round use the rounding mode Round to Nearest.

– Floating-point exceptions (see Section 5.6.2) cannot cause the system error handler to be invoked.

Programming Note

If a function is required that is specified by the IEEE standard, is not supported by the Vector Facility, and cannot be emulated satisfactorily using the functions that are supported by the Vector Facility, the functions provided by the Floating-Point Facility should be used; see Chapter 4.

5.6.2 Floating-Point Exceptions

The following floating-point exceptions may occur during execution of vector floating-point instructions.

– NaN Operand Exception
– Invalid Operation Exception
– Zero Divide Exception
– Log of Zero Exception
– Overflow Exception
– Underflow Exception

If an exception occurs, a result is placed into the corresponding target element as described in the following subsections. This result is the default result specified by Java, the IEEE standard, or C9X, as applicable.

Recall that denormalized source values are treated as if they were zero when $VSCR_{NJ}$=1. This has the following consequences regarding exceptions.

– Exceptions that can be caused by a zero source value can be caused by a denormalized source value when $VSCR_{NJ}$=1.

– Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when $VSCR_{NJ}$=1.

5.6.2.1 NaN Operand Exception

A NaN Operand Exception occurs when a source value for any of the following instructions is a NaN.

– A vector instruction that would normally produce floating-point results

– Either of the two *Vector Convert To Fixed-Point Word* instructions

– Any of the four *Vector Floating-Point Compare* instructions

The following actions are taken:

If the vector instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is a Signaling NaN it is converted to the corresponding Quiet NaN (by setting the high-order bit of the fraction field to 1) before being placed into the target element.

if the element in VRA is a NaN
  then the result is that NaN
  else if the element in VRB is a NaN
      then the result is that NaN
      else if the element in VRC is a NaN

then the result is that NaN
else if Invalid Operation exception
(Section 5.6.2.2)
then the result is the QNaN 0x7FC0_0000

If the instruction is either of the two *Vector Convert To Fixed-Point Word* instructions, the corresponding result is 0x0000_0000. $VSCR_{SAT}$ is not affected.

If the instruction is *Vector Compare Bounds Floating-Point*, the corresponding result is 0xC000_0000.

If the instruction is one of the other *Vector Floating-Point Compare* instructions, the corresponding result is 0x0000_0000.

5.6.2.2 Invalid Operation Exception

An Invalid Operation Exception occurs when a source value or set of source values is invalid for the specified operation. The invalid operations are:

– Magnitude subtraction of infinities
– Multiplication of infinity by zero
– Reciprocal square root estimate of a negative, nonzero number or -infinity.
– Log base 2 estimate of a negative, nonzero number or -infinity.

The corresponding result is the QNaN 0x7FC0_0000.

5.6.2.3 Zero Divide Exception

A Zero Divide Exception occurs when a V*ector Reciprocal Estimate Floating-Point* or *Vector Reciprocal Square Root Estimate Floating-Point* instruction is executed with a source value of zero.

The corresponding result is an infinity, where the sign is the sign of the source value.

5.6.2.4 Log of Zero Exception

A Log of Zero Exception occurs when a *Vector Log Base 2 Estimate Floating-Point* instruction is executed with a source value of zero.

The corresponding result is -Infinity.

5.6.2.5 Overflow Exception

An Overflow Exception occurs under either of the following conditions.

– For a vector instruction that would normally produce floating-point results, the magnitude of what would have been the result if the exponent range were unbounded exceeds that of the largest finite floating-point number for the target floating-point format.

– For either of the two *Vector Convert To Fixed-Point Word* instructions, either a source value is an infinity or the product of a source value and 2^{UIM} is a number too large in magnitude to be represented in the target fixed-point format.

The following actions are taken:

1. If the vector instruction would normally produce floating-point results, the corresponding result is an infinity, where the sign is the sign of the intermediate result.
2. If the instruction is *Vector Convert To Unsigned Fixed-Point Word Saturate*, the corresponding result is 0xFFFF_FFFF if the source value is a positive number or +infinity, and is 0x0000_0000 if the source value is a negative number or -infinity. $VSCR_{SAT}$ is set to 1.
3. If the instruction is *Vector Convert To Signed Fixed-Point Word Saturate*, the corresponding result is 0x7FFF_FFFF if the source value is a positive number or +infinity., and is 0x8000_0000 if the source value is a negative number or -infinity. $VSCR_{SAT}$ is set to 1.

5.6.2.6 Underflow Exception

An Underflow Exception can occur only for vector instructions that would normally produce floating-point results. It is detected before rounding. It occurs when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded is less in magnitude than the smallest normalized floating-point number for the target floating-point format.

The following actions are taken:

1. If $VSCR_{NJ}$=0, the corresponding result is the value produced by denormalizing and rounding the intermediate result.
2. If $VSCR_{NJ}$=1, the corresponding result is a zero, where the sign is the sign of the intermediate result.

5.7 Vector Storage Access Instructions

The Storage Access instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.10.3, “Effective Address Calculation” on page 26. The low-order bits of the EA that would correspond to an unaligned storage operand are ignored.

The *Load Vector Element Indexed* and *Store Vector Element Indexed* instructions transfer a byte, halfword, or word element between storage and a Vector Register. The *Load Vector Indexed* and *Store Vector Indexed* instructions transfer an aligned quadword between storage and a Vector Register.

5.7.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

5.7.2 Vector Load Instructions

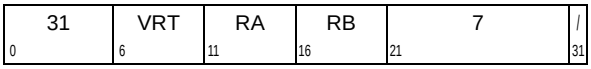
The aligned byte, halfword, word, or quadword in storage addressed by EA is loaded into register VRT.

Programming Note

The *Load Vector Element* instructions load the specified element into the same location in the target register as the location into which it would be loaded using the *Load Vector* instruction.

Load Vector Element Byte Indexed X-form

lvebx VRT,RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
eb  ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+7 ← MEM(EA,1)
else
   VRT120-(8×eb):127-(8×eb) ← MEM(EA,1)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte eb of register VRT. The remaining bytes in register VRT are set to undefined values.

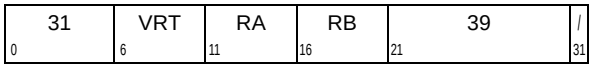
If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT. The remaining bytes in register VRT are set to undefined values.

Special Registers Altered:

None

Load Vector Element Halfword Indexed X-form

lvehx VRT,RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+15 ← MEM(EA,2)
else
   VRT112-(8×eb):127-(8×eb) ← MEM(EA,2)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

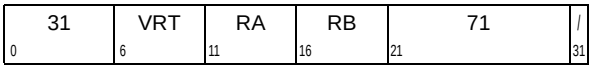
- the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

Special Registers Altered:

None

Load Vector Element Word Indexed X-form

lvewx VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFC

eb  ← EA60:63
VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+31 ← MEM(EA,4)
else
   VRT96-(8×eb):127-(8×eb) ← MEM(EA,4)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte eb+2 of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte eb+3 of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

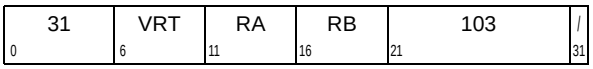
- the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte 13-eb of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte 12-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

Special Registers Altered:

None

Load Vector Indexed X-form

lvx VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
```

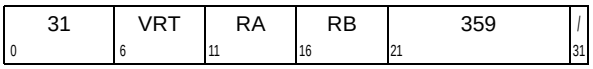
Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

Special Registers Altered:

None

Load Vector Indexed LRU X-form

lvxl VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon(EA)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

lvxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

Special Registers Altered:

None

Programming Note

On some implementations, the hint provided by the ***lvxl*** instruction and the corresponding hint provided by the ***stvxl***, ***lvepxl***, and ***stvepxl*** instructions are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for replacement when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.

5.7.3 Vector Store Instructions

Some portion or all of the contents of VRS are stored into the aligned byte, halfword, word, or quadword in storage addressed by EA.

Programming Note

The *Store Vector Element* instructions store the specified element into the same storage location as the location into which it would be stored using the *Store Vector* instruction.

Store Vector Element Byte Indexed X-form

stvebx VRS,RA,RB

31	VRS	RA	RB	135	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← b + (RB)
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,1) ← VRS8×eb:8×eb+7
else
   MEM(EA,1) ← VRS120-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of byte eb of register VRS are placed in the byte in storage at address EA.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access, the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA.

Special Registers Altered:

None

Programming Note

Unless bits 60:63 of the address are known to match the byte offset of the subject byte element in register VRS, software should use *Vector Splat* to splat the subject byte element before performing the store.

Store Vector Element Halfword Indexed X-form

stvehx VRS,RA,RB

31	VRS	RA	RB	167	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,2) ← VRS8×eb:8×eb+15
else
   MEM(EA,2) ← VRS112-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of byte eb of register VRS are placed in the byte in storage at address EA, and
- the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA, and
- the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1.

Special Registers Altered:

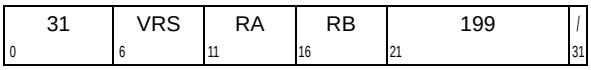
None

Programming Note

Unless bits 60:62 of the address are known to match the halfword offset of the subject halfword element in register VRS software should use *Vector Splat* to splat the subject halfword element before performing the store.

Store Vector Element Word Indexed X-form

stvewx VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFC
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,4) ← VRS8×eb:8×eb+31
else
   MEM(EA,4) ← VRS96-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of byte eb of register VRS are placed in the byte in storage at address EA,
- the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1,
- the contents of byte eb+2 of register VRS are placed in the byte in storage at address EA+2, and
- the contents of byte eb+3 of register VRS are placed in the byte in storage at address EA+3.

If Category: Vector.Little-Endian is supported, then if Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA,
- the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1,
- the contents of byte 13-eb of register VRS are placed in the byte in storage at address EA+2, and
- the contents of byte 12-eb of register VRS are placed in the byte in storage at address EA+3.

Special Registers Altered:

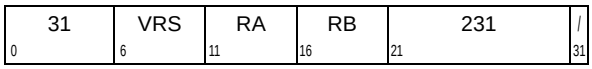
None

Programming Note

Unless bits 60:61 of the address are known to match the word offset of the subject word element in register VRS, software should use *Vector Splat* to splat the subject word element before performing the store.

Store Vector Indexed X-form

stvx VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
```

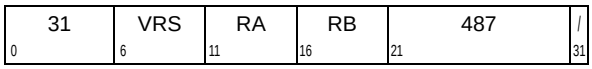
Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

Special Registers Altered:

None

Store Vector Indexed LRU X-form

stvxl VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
mark_as_not_likely_to_be_needed_again_anytime_soon(EA)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

stvxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

Special Registers Altered:

None

Programming Note

See the Programming Note for the ***lvxl*** instruction on page 182.

5.7.4 Vector Alignment Support Instructions

Programming Note

The ***lvsl*** and ***lvsr*** instructions can be used to create the permute control vector to be used by a subsequent ***vperm*** instruction (see page 198). Let X and Y be the contents of register VRA and VRB specified by the ***vperm***. The control vector created by ***lvsl*** causes the ***vperm*** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes. The control vector created by ***lvsr*** causes the ***vperm*** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

Programming Note

Examples of uses of ***lvsl***, ***lvsr***, and ***vperm*** to load and store unaligned data are given in Section 5.4.1.

These instructions can also be used to rotate or shift the contents of a Vector Register left (***lvsl***) or right (***lvsr***) by sh bytes. For rotating, the Vector Register to be rotated should be specified as both register VRA and VRB for ***vperm***. For shifting left, VRB for ***vperm*** should be a register containing all zeros and VRA should contain the value to be shifted, and vice versa for shifting right.

Load Vector for Shift Left Indexed X-form

lvsl VRT,RA,RB

31	VRT	RA	RB	6	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
sh  ← (b + (RB))60:63
switch(sh)
   case(0x0): VRT←0x000102030405060708090A0B0C0D0E0F
   case(0x1): VRT←0x0102030405060708090A0B0C0D0E0F10
   case(0x2): VRT←0x02030405060708090A0B0C0D0E0F1011
   case(0x3): VRT←0x030405060708090A0B0C0D0E0F101112
   case(0x4): VRT←0x0405060708090A0B0C0D0E0F10111213
   case(0x5): VRT←0x05060708090A0B0C0D0E0F1011121314
   case(0x6): VRT←0x060708090A0B0C0D0E0F101112131415
   case(0x7): VRT←0x0708090A0B0C0D0E0F10111213141516
   case(0x8): VRT←0x08090A0B0C0D0E0F1011121314151617
   case(0x9): VRT←0x090A0B0C0D0E0F101112131415161718
   case(0xA): VRT←0x0A0B0C0D0E0F10111213141516171819
   case(0xB): VRT←0x0B0C0D0E0F101112131415161718191A
   case(0xC): VRT←0x0C0D0E0F101112131415161718191A1B
   case(0xD): VRT←0x0D0E0F101112131415161718191A1B1C
   case(0xE): VRT←0x0E0F101112131415161718191A1B1C1D
   case(0xF): VRT←0x0F101112131415161718191A1B1C1D1E
```

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32 byte value `0x00` || `0x01` || `0x02` || ... || `0x1E` || `0x1F`.

Bytes sh to sh+15 of X are placed into VRT.

Special Registers Altered:

None

Load Vector for Shift Right Indexed X-form

lvsr VRT,RA,RB

31	VRT	RA	RB	38	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
sh  ← (b + (RB))60:63
switch(sh)
   case(0x0): VRT←0x101112131415161718191A1B1C1D1E1F
   case(0x1): VRT←0x0F101112131415161718191A1B1C1D1E
   case(0x2): VRT←0x0E0F101112131415161718191A1B1C1D
   case(0x3): VRT←0x0D0E0F101112131415161718191A1B1C
   case(0x4): VRT←0x0C0D0E0F101112131415161718191A1B
   case(0x5): VRT←0x0B0C0D0E0F10111213141516171819A
   case(0x6): VRT←0x0A0B0C0D0E0F1011121314151617181 9
   case(0x7): VRT←0x090A0B0C0D0E0F101112131415161718
   case(0x8): VRT←0x08090A0B0C0D0E0F1011121314151617
   case(0x9): VRT←0x0708090A0B0C0D0E0F10111213141516
   case(0xA): VRT←0x060708090A0B0C0D0E0F101112131415
   case(0xB): VRT←0x05060708090A0B0C0D0E0F1011121314
   case(0xC): VRT←0x0405060708090A0B0C0D0E0F10111213
   case(0xD): VRT←0x030405060708090A0B0C0D0E0F101112
   case(0xE): VRT←0x02030405060708090A0B0C0D0E0F1011
   case(0xF): VRT←0x0102030405060708090A0B0C0D0E0F10
```

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32-byte value `0x00` || `0x01` || `0x02` || ... || `0x1E` || `0x1F`.

Bytes 16-sh to 31-sh of X are placed into VRT.

Special Registers Altered:

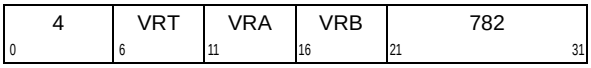
None

5.8 Vector Permute and Formatting Instructions

5.8.1 Vector Pack and Unpack Instructions

Vector Pack Pixel VX-form

vpkpx VRT,VRA,VRB




```
do i = 0 to 63 by 16
   VR[VRT]i          ← VR[VRA]i×2+7
   VR[VRT]i+1:i+5    ← VR[VRA]i×2+8:i×2+12
   VR[VRT]i+6:i+10   ← VR[VRA]i×2+16:i×2+20
   VR[VRT]i+11:i+15  ← VR[VRA]i×2+24:i×2+28
   VR[VRT]i+64       ← VR[VRB]i×2+7
   VR[VRT]i+65:i+69  ← VR[VRB]i×2+8:i×2+12
   VR[VRT]i+70:i+74  ← VR[VRB]i×2+16:i×2+20
   VR[VRT]i+75:i+79  ← VR[VRB]i×2+24:i×2+28
end
```

Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB].

For each integer value i from 0 to 7, do the following.

Word element i in the source vector is packed to produce a 16-bit value as described below.

- bit 7 of the first byte (bit 7 of the word)
- bits 0:4 of the second byte (bits 8:12 of the word)
- bits 0:4 of the third byte (bits 16:20 of the word)
- bits 0:4 of the fourth byte (bits 24:28 of the word)

The result is placed into halfword element i of VR[VRT].

Special Registers Altered:

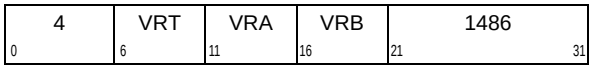
None

Programming Note

Each source word can be considered to be a 32-bit "pixel", consisting of four 8-bit "channels". Each target halfword can be considered to be a 16-bit pixel, consisting of one 1-bit channel and three 5-bit channels. A channel can be used to specify the intensity of a particular color, such as red, green, or blue, or to provide other information needed by the application.

Vector Pack Signed Doubleword Signed Saturate VX-form

vpksdss VRT,VRA,VRB




```
src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( Clamp( ExtendSign( src.dword[i]),
-2^31, 2^31-1 ), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The signed integer value in doubleword element i of src is placed into word element i of VR[VRT] in signed integer format.

- If the value is greater than $2^{31}-1$ the result saturates to $2^{31}-1$.
- If the value is less than -2^{31} the result saturates to -2^{31}.

Special Registers Altered:

SAT

Vector Pack Signed Doubleword Unsigned Saturate VX-form

vpksdus VRT,VRA,VRB

4	VRT	VRA	VRB	1358
0	6	11	16	21 31

```
src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( Clamp( ExtendSign(src.dword[i]), 0,
2^32-1 ), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The signed integer value in doubleword element i of src is placed into word element i of VR[VRT] in unsigned integer format.

– If the value is greater than 2^{32}-1 the result saturates to 2^{32}-1.

– If the value is less than 0 the result saturates to 0.

Special Registers Altered:

SAT

Vector Pack Signed Halfword Signed Saturate VX-form

vpkshss VRT,VRA,VRB

4	VRT	VRA	VRB	398
0	6	11	16	21 31

```
do i=0 to 63 by 8
   src1 ← EXTS((VRA)i×2:i×2+15)
   src2 ← EXTS((VRB)i×2:i×2+15)
   VRTi:i+7     ← Clamp(src1, -128, 127)24:31
   VRTi+64:i+71 ← Clamp(src2, -128, 127)24:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.

Signed-integer halfword element i in the source vector is converted to an signed-integer byte.

– If the value of the element is greater than 127 the result saturates to 127

– If the value of the element is less than -128 the result saturates to -128.

The low-order 8 bits of the result is placed into byte element *i* of VRT.

Special Registers Altered:

SAT

Vector Pack Signed Halfword Unsigned Saturate VX-form

vpkshus VRT,VRA,VRB

4	VRT	VRA	VRB	270
0	6	11	16	21 31

```
do i=0 to 63 by 8
   src1 ← EXTS((VRA)i×2:i×2+15)
   src2 ← EXTS((VRB)i×2:i×2+15)
   VRTi:i+7     ← Clamp(src1, 0, 255)24:31
   VRTi+64:i+71 ← Clamp(src2, 0, 255)24:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.
Signed-integer halfword element i in the source vector is converted to an unsigned-integer byte.

– If the value of the element is greater than 255 the result saturates to 255

– If the value of the element is less than 0 the result saturates to 0.

The low-order 8 bits of the result is placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Word Signed Saturate VX-form

vpkswss VRT,VRA,VRB

4	VRT	VRA	VRB	462
0	6	11	16	21 31

```
do i=0 to 63 by 16
   src1 ← EXTS((VRA)i×2:i×2+31)
   src2 ← EXTS((VRB)i×2:i×2+31)
   VRTi:i+15    ← Clamp(src1, -2^15, 2^15-1)16:31
   VRTi+64:i+79 ← Clamp(src2, -2^15, 2^15-1)16:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.
Signed-integer word element i in the source vector is converted to an signed-integer halfword.

– If the value of the element is greater than 2^{15}-1 the result saturates to 2^{15}-1

– If the value of the element is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result is placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Word Unsigned Saturate VX-form

vpkswus VRT,VRA,VRB

4	VRT	VRA	VRB	334
0	6	11	16	21 31

```
do i=0 to 63 by 16
   src1 ← EXTS((VRA)i×2:i×2+31)
   src2 ← EXTS((VRB)i×2:i×2+31)
   VRTi:i+15    ← Clamp(src1, 0, 2^16-1)16:31
   VRTi+64:i+79 ← Clamp(src2, 0, 2^16-1)16:31
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.

Signed-integer word element i in the source vector is converted to an unsigned-integer halfword.

– If the value of the element is greater than 2^{16}-1 the result saturates to 2^{16}-1

– If the value of the element is less than 0 the result saturates to 0.

The low-order 16 bits of the result is placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Pack Unsigned Doubleword Unsigned Modulo VX-form

vpkudum VRT,VRA,VRB

4	VRT	VRA	VRB	1102
0	6	11	16	21 31

```
if MSR.VEC then Vector_Unavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( ExtendZero(src.dword[i]), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The contents of bits 32:63 of doubleword element i of src is placed into word element i of VR[VRT].

Special Registers Altered:

None

Vector Pack Unsigned Doubleword Unsigned Saturate VX-form

vpkudus VRT,VRA,VRB

4	VRT	VRA	VRB	1230
0	6	11	16	21 31

```
if MSR.VEC then Vector_Unavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( Clamp( ExtendZero(src.dword[i]), 0,
2^32-1 ), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The unsigned integer value in doubleword element i of src is placed into word element i of VR[VRT] in unsigned integer format.

– If the value of the element is greater than 2^{32}-1 the result saturates to 2^{32}-1

Special Registers Altered:

SAT

Vector Pack Unsigned Halfword Unsigned Modulo VX-form

vpkuhum VRT,VRA,VRB

4	VRT	VRA	VRB	14
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRTi:i+7    ← (VRA)i×2+8:i×2+15
   VRTi+64:i+71 ← (VRB)i×2+8:i×2+15
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.

The contents of bits 8:15 of halfword element i in the source vector is placed into byte element i of VRT.

Special Registers Altered:

None

Vector Pack Unsigned Halfword Unsigned Saturate VX-form

vpkuhus VRT,VRA,VRB

4	VRT	VRA	VRB	142
0	6	11	16	21 31

```
do i=0 to 63 by 8
   src1 ← EXTZ((VRA)ix2:ix2+15)
   src2 ← EXTZ((VRB)ix2:ix2+15)
   VRTi:i+7     ← Clamp( src1, 0, 255 )24:31
   VRTi+64:i+71 ← Clamp( src2, 0, 255 )24:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.
Unsigned-integer halfword element i in the source vector is converted to an unsigned-integer byte.

– If the value of the element is greater than 255 the result saturates to 255.

The low-order 8 bits of the result is placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Pack Unsigned Word Unsigned Modulo VX-form

vpkuwum VRT,VRA,VRB

4	VRT	VRA	VRB	78
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRTi:i+15     ← (VRA)ix2+16:ix2+31
   VRTi+64:i+79 ← (VRB)ix2+16:ix2+31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.
The contents of bits 16:31 of word element i in the source vector is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Pack Unsigned Word Unsigned Saturate VX-form

vpkuwus VRT,VRA,VRB

4	VRT	VRA	VRB	206
0	6	11	16	21 31

```
do i=0 to 63 by 16
   src1 ← EXTZ((VRA)ix2:ix2+31 )
   src2 ← EXTZ((VRB)ix2:ix2+31 )
   VRTi:i+15 ← Clamp( src1, 0, 2^16-1 )16:31
   VRTi+64:i+79  ← Clamp( src2, 0, 2^16-1 )16:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.
Unsigned-integer word element i in the source vector is converted to an unsigned-integer halfword.

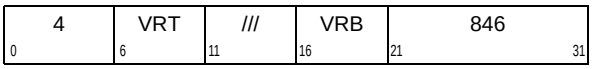
– If the value of the element is greater than 2^{16}-1 the result saturates to 2^{16}-1.

The low-order 16 bits of the result is placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Unpack High Pixel VX-form

vupkhpx VRT,VRB




```
do i=0 to 63 by 16
   VRT[ix2:ix2+7]     ← EXTS((VRB)[i     ])
   VRT[ix2+8:ix2+15]  ← EXTZ((VRB)[i+1:i+5 ])
   VRT[ix2+16:ix2+23] ← EXTZ((VRB)[i+6:i+10])
   VRT[ix2+24:ix2+31] ← EXTZ((VRB)[i+11:i+15])
end
```

For each vector element *i* from 0 to 3, do the following.
Halfword element *i* in VRB is unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element *i* of VRT.

Special Registers Altered:

None

Programming Note

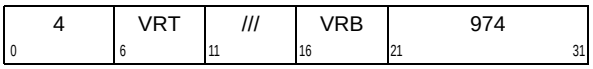
The source and target elements can be considered to be 16-bit and 32-bit “pixels” respectively, having the formats described in the Programming Note for the *Vector Pack Pixel* instruction on page 187.

Programming Note

Notice that the unpacking done by the *Vector Unpack Pixel* instructions does not reverse the packing done by the *Vector Pack Pixel* instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, *Vector Unpack Pixel* inserts high-order bits while *Vector Pack Pixel* discards low-order bits).

Vector Unpack Low Pixel VX-form

vupklpx VRT,VRB




```
do i=0 to 63 by 16
   VRT[ix2:ix2+7]     ← EXTS((VRB)[i+64   ])
   VRT[ix2+8:ix2+15]  ← EXTZ((VRB)[i+65:i+69])
   VRT[ix2+16:ix2+23] ← EXTZ((VRB)[i+70:i+74])
   VRT[ix2+24:ix2+31] ← EXTZ((VRB)[i+75:i+79])
end
```

For each vector element *i* from 0 to 3, do the following.
Halfword element *i*+4 in VRB is unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

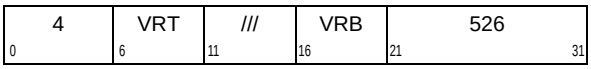
The result is placed in word element *i* of VRT.

Special Registers Altered:

None

Vector Unpack High Signed Byte VX-form

vupkhsb VRT,VRB



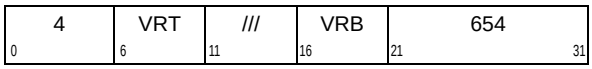

```
do i=0 to 63 by 8
   VRT[i*2:i*2+15] ← EXTS((VRB)[i:i+7])
end
```

For each vector element i from 0 to 7, do the following.
Signed-integer byte element i in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element *i* in VRT.

Special Registers Altered:
None

Vector Unpack Low Signed Byte VX-form

vupklsb VRT,VRB



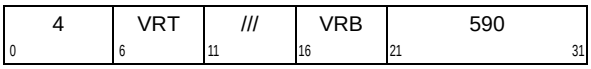

```
do i=0 to 63 by 8
   VRT[i*2:i*2+15] ← EXTS((VRB)[i+64:i+71])
end
```

For each vector element *i* from 0 to 7, do the following.
Signed-integer byte element i+8 in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element *i* in VRT.

Special Registers Altered:
None

Vector Unpack High Signed Halfword VX-form

vupkhsh VRT,VRB



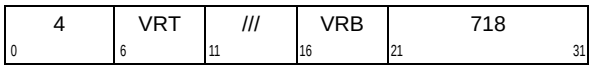

```
do i=0 to 63 by 16
   VRT[i*2:i*2+31] ← EXTS((VRB)[i:i+15])
end
```

For each vector element *i* from 0 to 3, do the following.
Signed-integer halfword element *i* in VRB is sign-extended to produce a signed-integer word and placed into word element *i* in VRT.

Special Registers Altered:
None

Vector Unpack Low Signed Halfword VX-form

vupklsh VRT,VRB



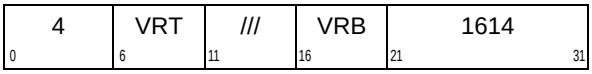

```
do i=0 to 63 by 16
   VRT[i*2:i*2+31] ← EXTS((VRB)[i+64:i+79])
end
```

For each vector element *i* from 0 to 3, do the following.
Signed-integer halfword element *i*+4 in VRB is sign-extended to produce a signed-integer word and placed into word element *i* in VRT.

Special Registers Altered:
None

Vector Unpack High Signed Word VX-form

vupkhsw VRT,VRB



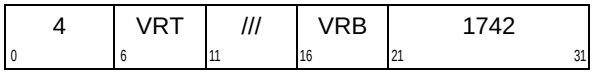

```
VR[VRT].dword[0] ← Chop( ExtendSign(VR[VRB].word[0]), 64 )
VR[VRT].dword[1] ← Chop( ExtendSign(VR[VRB].word[1]), 64 )
```

For each integer value i from 0 to 1, do the following.
The signed integer value in word element i of VR[VRB] is sign-extended and placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Unpack Low Signed Word VX-form

vupklsw VRT,VRB




```
VR[VRT].dword[0] ← Chop( ExtendSign(VR[VRB].word[2]), 64 )
VR[VRT].dword[1] ← Chop( ExtendSign(VR[VRB].word[3]), 64 )
```

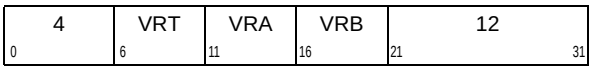
For each integer value i from 0 to 1, do the following.
The signed integer value in word element i+2 of VR[VRB] is sign-extended and placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

5.8.2 Vector Merge Instructions

Vector Merge High Byte VX-form

vmrghb VRT,VRA,VRB




```
do i=0 to 63 by 8
   VRT[ix2:ix2+7]    ← (VRA)[i:i+7]
   VRT[ix2+8:ix2+15] ← (VRB)[i:i+7]
end
```

For each vector element *i* from 0 to 7, do the following.

Byte element *i* in VRA is placed into byte element 2×i in VRT.

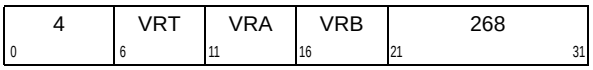
Byte element *i* in VRB is placed into byte element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge Low Byte VX-form

vmrglb VRT,VRA,VRB




```
do i=0 to 63 by 8
   VRT[ix2:ix2+7]    ← (VRA)[i+64:i+71]
   VRT[ix2+8:ix2+15] ← (VRB)[i+64:i+71]
end
```

For each vector element *i* from 0 to 7, do the following.

Byte element i+8 in VRA is placed into byte element 2×i in VRT.

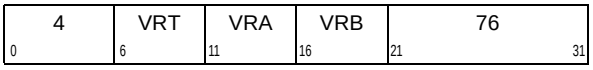
Byte element i+8 in VRB is placed into byte element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge High Halfword VX-form

vmrghh VRT,VRA,VRB




```
do i=0 to 63 by 16
   VRT[ix2:ix2+15]    ← (VRA)[i:i+15]
   VRT[ix2+16:ix2+31] ← (VRB)[i:i+15]
end
```

For each vector element *i* from 0 to 3, do the following.

Halfword element *i* in VRA is placed into halfword element 2×i in VRT.

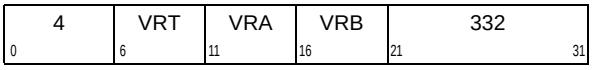
Halfword element *i* in VRB is placed into halfword element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge Low Halfword VX-form

vmrglh VRT,VRA,VRB




```
do i=0 to 63 by 16
   VRT[ix2:ix2+15]    ← (VRA)[i+64:i+79]
   VRT[ix2+16:ix2+31] ← (VRB)[i+64:i+79]
end
```

For each vector element *i* from 0 to 3, do the following.

Halfword element i+4 in VRA is placed into halfword element 2×i in VRT.

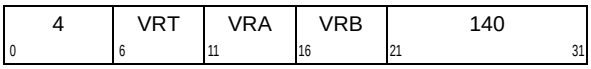
Halfword element i+4 in VRB is placed into halfword element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge High Word VX-form

vmrghw VRT,VRA,VRB




```
do i=0 to 63 by 32
   VRT[ix2:ix2+31]    ← (VRA)[i:i+31]
   VRT[ix2+32:ix2+63] ← (VRB)[i:i+31]
end
```

For each vector element *i* from 0 to 1, do the following.

Word element *i* in VRA is placed into word element 2×i in VRT.

Word element *i* in VRB is placed into word element 2×i+1 in VRT.

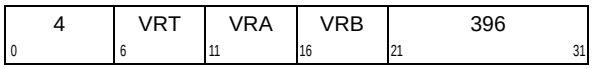
The word elements in the high-order half of VRA are placed, in the same order, into the even-numbered word elements of VRT. The word elements in the high-order half of VRB are placed, in the same order, into the odd-numbered word elements of VRT.

Special Registers Altered:

None

Vector Merge Low Word VX-form

vmrglw VRT,VRA,VRB




```
do i=0 to 63 by 32
   VRT[ix2:ix2+31]    ← (VRA)[i+64:i+95]
   VRT[ix2+32:ix2+63] ← (VRB)[i+64:i+95]
end
```

For each vector element *i* from 0 to 1, do the following.

Word element i+2 in VRA is placed into word element 2×i in VRT.

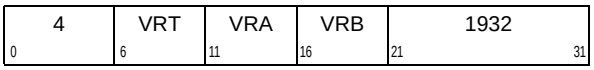
Word element i+2 in VRB is placed into word element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge Even Word VX-form
[Category: Vector-Scalar]

vmrgew VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
VR[VRT].word[0] ← VR[VRA].word[0]
VR[VRT].word[1] ← VR[VRB].word[0]
VR[VRT].word[2] ← VR[VRA].word[2]
VR[VRT].word[3] ← VR[VRB].word[2]
```

The contents of word element 0 of VR[VRA] are placed into word element 0 of VR[VRT].

The contents of word element 0 of VR[VRB] are placed into word element 1 of VR[VRT].

The contents of word element 2 of VR[VRA] are placed into word element 2 of VR[VRT].

The contents of word element 2 of VR[VRB] are placed into word element 3 of VR[VRT].

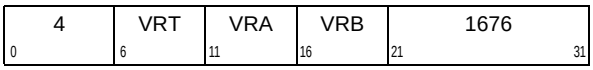
vmrgew is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered

None

Vector Merge Odd Word VX-form
[Category: Vector-Scalar]

vmrgow VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
VR[VRT].word[0] ← VR[VRA].word[1]
VR[VRT].word[1] ← VR[VRB].word[1]
VR[VRT].word[2] ← VR[VRA].word[3]
VR[VRT].word[3] ← VR[VRB].word[3]
```

The contents of word element 1 of VR[VRA] are placed into word element 0 of VR[VRT].

The contents of word element 1 of VR[VRB] are placed into word element 1 of VR[VRT].

The contents of word element 3 of VR[VRA] are placed into word element 2 of VR[VRT].

The contents of word element 3 of VR[VRB] are placed into word element 3 of VR[VRT].

vmrgow is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered

None

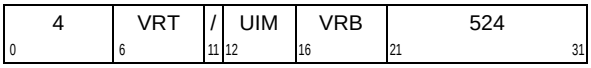
5.8.3 Vector Splat Instructions

Programming Note

The *Vector Splat* instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (e.g., multiplying all elements of a Vector Register by a constant).

Vector Splat Byte VX-form

vspltb VRT,VRB,UIM




```
b ← UIM || 0b000
do i=0 to 127 by 8
   VRT_{i:i+7} ← (VRB)_{b:b+7}
end
```

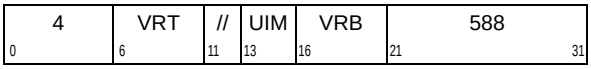
For each integer value i from 0 to 15, do the following. The contents of byte element UIM in VRB are placed into byte element i of VRT.

Special Registers Altered:

None

Vector Splat Halfword VX-form

vsplth VRT,VRB,UIM




```
b ← UIM || 0b0000
do i=0 to 127 by 16
   VRT_{i:i+15} ← (VRB)_{b:b+15}
end
```

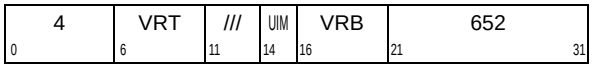
For each integer value i from 0 to 7, do the following. The contents of halfword element UIM in VRB are placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Splat Word VX-form

vspltw VRT,VRB,UIM




```
b ← UIM || 0b00000
do i=0 to 127 by 32
   VRT_{i:i+31} ← (VRB)_{b:b+31}
end
```

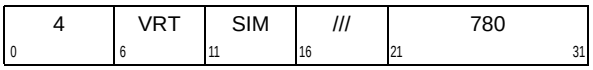
For each integer value i from 0 to 3, do the following. The contents of word element UIM in VRB are placed into word element i of VRT.

Special Registers Altered:

None

Vector Splat Immediate Signed Byte VX-form

vspltisb VRT,SIM



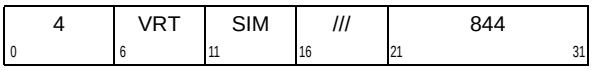

```
do i=0 to 127 by 8
   VRT_{i:i+7} ← EXTS(SIM, 8)
end
```

For each integer value i from 0 to 15, do the following.
The value of the SIM field, sign-extended to 8 bits, is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Halfword VX-form

vspltish VRT,SIM



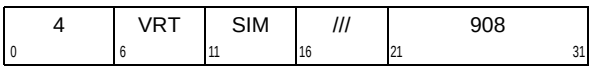

```
do i=0 to 127 by 16
   VRT_{i:i+15} ← EXTS(SIM, 16)
end
```

For each integer value i from 0 to 7, do the following.
The value of the SIM field, sign-extended to 16 bits, is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Word VX-form

vspltisw VRT,SIM




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← EXTS(SIM, 32)
end
```

For each vector element *i* from 0 to 3, do the following.
The value of the SIM field, sign-extended to 32 bits, is placed into word element *i* of VRT.

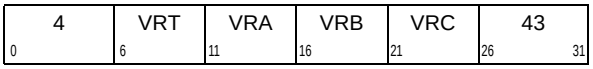
Special Registers Altered:
None

5.8.4 Vector Permute Instruction

The *Vector Permute* instruction allows any byte in two source Vector Registers to be copied to any byte in the target Vector Register. The bytes in a third source Vector Register specify from which byte in the first two source Vector Registers the corresponding target byte is to be copied. The contents of the third source Vector Register are sometimes referred to as the "permute control vector".

Vector Permute VA-form

vperm VRT,VRA,VRB,VRC




```
temp_{0:255} ← (VRA) || (VRB)
do i=0 to 127 by 8
   b ← (VRC)_{i+3:i+7} || 0b000
   VRT_{i:i+7} ← temp_{b:b+7}
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value *i* from 0 to 15, do the following.
The contents of the byte element in the source vector specified by bits 3:7 of byte element *i* of VRC are placed into byte element *i* of VRT.

Special Registers Altered:
None

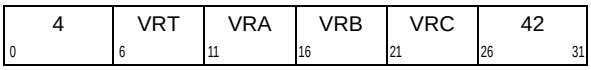
Programming Note

See the Programming Notes with the *Load Vector for Shift Left* and *Load Vector for Shift Right* instructions on page 186 for examples of uses of ***vperm***.

5.8.5 Vector Select Instruction

Vector Select VA-form

vsel VRT,VRA,VRB,VRC




```
do i=0 to 127
   VRT_i ← ((VRC)_i=0) ? (VRA)_i : (VRB)_i
end
```

For each bit in VRC that contains the value 0, the corresponding bit in VRA is placed into the corresponding bit of VRT. Otherwise, the corresponding bit in VRB is placed into the corresponding bit of VRT.

Special Registers Altered:

None

5.8.6 Vector Shift Instructions

The *Vector Shift* instructions rotate or shift the contents of a Vector Register or a pair of Vector Registers left or right by a specified number of bytes (***vslo***, ***vsro***, ***vsldoi***) or bits (***vsl***, ***vsr***). Depending on the instruction, this "shift count" is specified either by the contents of a Vector Register or by an immediate field in the instruction. In the former case, 7 bits of the shift count register give the shift count in bits ($0 \le \text{count} \le 127$). Of these 7 bits, the high-order 4 bits give the number of complete bytes by which to shift and are used by ***vslo*** and ***vsro***; the low-order 3 bits give the number of remaining bits by which to shift and are used by ***vsl*** and ***vsr***.

Programming Note

A pair of these instructions, specifying the same shift count register, can be used to shift the contents of a Vector Register left or right by the number of bits (0-127) specified in the shift count register. The following example shifts the contents of register Vx left by the number of bits specified in register Vy and places the result into register Vz.

```
vslo    Vz,Vx,Vy
vsl     Vz,Vz,Vy
```

Vector Shift Left VX-form

vsl VRT,VRA,VRB

4	VRT	VRA	VRB	452
0	6	11	16	21 31

```
sh ← (VRB)125:127
t ← 1
do i=0 to 127 by 8
   t ← t & ((VRB)i+5:i+7=sh)
end
if t=1 then VRT ← (VRA) << sh
else        VRT ← undefined
```

The contents of VRA are shifted left by the number of bits specified in $(VRB)_{125:127}$.

– Bits shifted out of bit 0 are lost.

– Zeros are supplied to the vacated bits on the right.

The result is place into VRT, except if, for any byte element in register VRB, the low-order 3 bits are not equal to the shift amount, then VRT is undefined.

Special Registers Altered:

None

Vector Shift Left Double by Octet Immediate VA-form

vsldoi VRT,VRA,VRB,SHB

4	VRT	VRA	VRB	/	SHB	44
0	6	11	16	21	22	26 31

```
VRT ← ( (VRA) || (VRB) )8×SHB:8×SHB+127
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB. Bytes SHB:SHB+15 of the source vector are placed into VRT.

Special Registers Altered:

None

Vector Shift Left by Octet VX-form

vslo VRT,VRA,VRB

4	VRT	VRA	VRB	1036
0	6	11	16	21 31

```
shb ← (VRB)121:124
VRT ← (VRA) << ( shb || 0b000 )
```

The contents of VRA are shifted left by the number of bytes specified in $(VRB)_{121:124}$.

– Bytes shifted out of byte 0 are lost.

– Zeros are supplied to the vacated bytes on the right.

The result is placed into VRT.

Special Registers Altered:

None

Vector Shift Right VX-form

vsr VRT,VRA,VRB

4	VRT	VRA	VRB	708
0	6	11	16	21 31

```
sh ← (VRB)125:127
t ← 1
do i=0 to 127 by 8
   t ← t & ((VRB)i+5:i+7=sh)
end
if t=1 then VRT ← (VRA) >>ui sh
else        VRT ← undefined
```

The contents of VRA are shifted right by the number of bits specified in $(VRB)_{125:127}$.

– Bits shifted out of bit 127 are lost.

– Zeros are supplied to the vacated bits on the left.

The result is place into VRT, except if, for any byte element in register VRB, the low-order 3 bits are not equal to the shift amount, then VRT is undefined.

Special Registers Altered:

None

Programming Note

A double-register shift by a dynamically specified number of bits (0-127) can be performed in six instructions. The following example shifts Vw || Vx left by the number of bits specified in Vy and places the high-order 128 bits of the result into Vz.

```
vslo     Vt1,Vw,Vy    #shift high-order reg left
vsl      Vt1,Vt1,Vy
vsububm  Vt3,V0,Vy    #adjust shift count ((V0)=0)
vsro     Vt2,Vx,Vt3   #shift low-order reg right
vsr      Vt2,Vt2,Vt3
vor      Vz,Vt1,Vt2   #merge to get final result
```

Vector Shift Right by Octet VX-form

vsro VRT,VRA,VRB

4	VRT	VRA	VRB	1100
0	6	11	16	21 31

```
shb ← (VRB)121:124
VRT ← (VRA) >>ui ( shb || 0b000 )
```

The contents of VRA are shifted right by the number of bytes specified in $(VRB)_{121:124}$.

– Bytes shifted out of byte 15 are lost.

– Zeros are supplied to the vacated bytes on the left.

The result is placed into VRT.

Special Registers Altered:

None

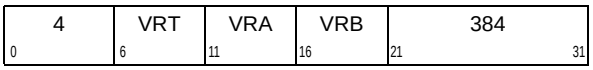
5.9 Vector Integer Instructions

5.9.1 Vector Integer Arithmetic Instructions

5.9.1.1 Vector Integer Add Instructions

Vector Add and Write Carry-Out Unsigned Word VX-form

vaddcuw VRT,VRA,VRB



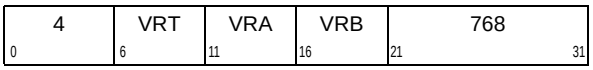

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop( ( aop +int bop ) >>ui 32,1)
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB. The carry out of the 32-bit sum is zero-extended to 32 bits and placed into word element i of VRT.

Special Registers Altered:
None

Vector Add Signed Byte Saturate VX-form

vaddsbs VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTS(VRAi:i+7)
   bop ← EXTS(VRBi:i+7)
   VRTi:i+7 ← Clamp( aop +int bop, -128, 127 )24:31
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is added to signed-integer byte element i in VRB.

– If the sum is greater than 127 the result saturates to 127.

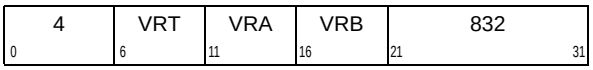
– If the sum is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Add Signed Halfword Saturate VX-form

vaddshs VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int bop, -2^15, 2^15-1)16:31
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is added to signed-integer halfword element i in VRB.

– If the sum is greater than 2^{15}-1 the result saturates to 2^{15}-1

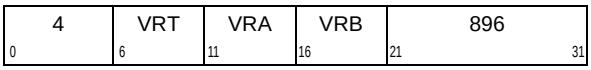
– If the sum is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Add Signed Word Saturate VX-form

vaddsws VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int bop, -2^31, 2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

Signed-integer word element i in VRA is added to signed-integer word element i in VRB.

– If the sum is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the sum is less than -2^{31} the result saturates to -2^{31}.

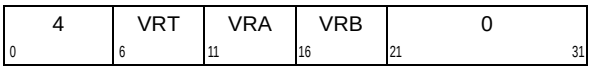
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Add Unsigned Byte Modulo VX-form

vaddubm VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop( aop +int bop, 8 )
end
```

For each integer value i from 0 to 15, do the following.

Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

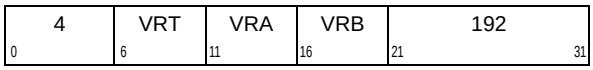
None

Programming Note

vaddubm can be used for unsigned or signed-integers.

Vector Add Unsigned Doubleword Modulo VX-form

vaddudm VRT,VRA,VRB




```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← Chop( aop +int bop, 64 )
end
```

For each integer value i from 0 to 1, do the following.

The integer value in doubleword element i of VR[VRB] is added to the integer value in doubleword element i of VR[VRA].

The low-order 64 bits of the result are placed into doubleword element i of VR[VRT].

Special Registers Altered:

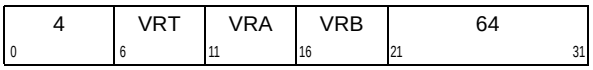
None

Programming Note

vaddudm can be used for signed or unsigned integers.

Vector Add Unsigned Halfword Modulo VX-form

vadduhm VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop  ← EXTZ((VRA)i:i+15)
   bop  ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Chop( aop +int bop, 16 )
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

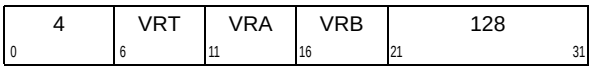
None

Programming Note

vadduhm can be used for unsigned or signed-integers.

Vector Add Unsigned Word Modulo VX-form

vadduwm VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   temp ← aop +int bop
   VRTi:i+31 ← Chop( aop +int bop, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

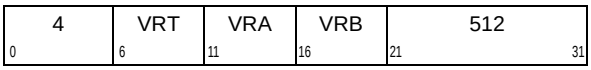
None

Programming Note

vadduwm can be used for unsigned or signed-integers.

Vector Add Unsigned Byte Saturate VX-form

vaddubs VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Clamp( aop +int bop, 0, 255 )24:31
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB.

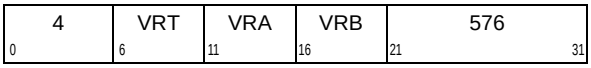
– If the sum is greater than 255 the result saturates to 255.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Halfword Saturate VX-form

vadduhs VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int bop, 0, 2^16-1)16:31
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB.

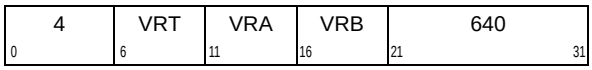
– If the sum is greater than 2^{16}-1 the result saturates to 2^{16}-1.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Word Saturate VX-form

vadduws VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int bop, 0, 2^32-1)
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB.

– If the sum is greater than 2^{32}-1 the result saturates to 2^{32}-1.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Quadword Modulo VX-form

vadduqm VRT,VRA,VRB

4	VRT	VRA	VRB	256
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(src2)

VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1 and src2 are placed into VR[VRT].

Special Registers Altered:

None

Vector Add Extended Unsigned Quadword Modulo VA-form

vaddeuqm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	60
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(src2) + EXTZ(cin)

VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Vector Add & write Carry Unsigned Quadword VX-form

vaddcuq VRT,VRA,VRB

4	VRT	VRA	VRB	320
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(src2)

VR[VRT] ← Chop( EXTZ( Chop(sum>>128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1 and src2 is placed into VR[VRT].

Special Registers Altered:

None

Vector Add Extended & write Carry Unsigned Quadword VA-form

vaddecuq VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	61
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(src2) + EXTZ(cin)

VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Programming Note

The *Vector Add Unsigned Quadword* instructions support efficient wide-integer addition. The following code sequence can be used to implement a 512-bit signed or unsigned add operation.

```
vadduqm         vS3,vA3,vB3         # bits 384:511 of sum
vaddcuq         vC3,vA3,vB3         # carry out of bit 384 of sum
vaddeuqm        vS2,vA2,vB2,vC3     # bits 256:383 of sum
vaddecuq        vC2,vA2,vB2,vC3     # carry out of bit 256 of sum
vaddeuqm        vS1,vA1,vB1,vC2     # bits 128:255 of sum
vaddecuq        vC1,vA1,vB1,vC2     # carry out of bit 128 of sum
vaddeuqm        vS0,vA0,vB0,vC1     # bits 0:127 of sum
```

5.9.1.2 Vector Integer Subtract Instructions

Vector Subtract and Write Carry-Out Unsigned Word VX-form

vsubcuw VRT,VRA,VRB

4	VRT	VRA	VRB	1408
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   temp ← (aop +int ¬bop +int 1) >> 32
   VRTi:i+31 ← temp & 0x0000_0001
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into word element i of VRT.

Special Registers Altered:
None

Vector Subtract Signed Byte Saturate VX-form

vsubsbs VRT,VRA,VRB

4	VRT	VRA	VRB	1792
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← Clamp(aop +int ¬bop +int 1, -128, 127)24:31
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRB is subtracted from signed-integer byte element i in VRA.

– If the intermediate result is greater than 127 the result saturates to 127.

– If the intermediate result is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Subtract Signed Halfword Saturate VX-form

vsubshs VRT,VRA,VRB

4	VRT	VRA	VRB	1856
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   temp ← aop +int ¬bop +int 1
   VRTi:i+15 ← Clamp(temp, -2^15, 2^15-1)16:31
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRB is subtracted from signed-integer halfword element i in VRA.

– If the intermediate result is greater than 2^{15}-1 the result saturates to 2^{15}-1.

– If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Subtract Signed Word Saturate VX-form

vsubsws VRT,VRA,VRB

4	VRT	VRA	VRB	1920
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int ¬bop +int 1,-2^31,2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

Signed-integer word element i in VRB is subtracted from signed-integer word element i in VRA.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

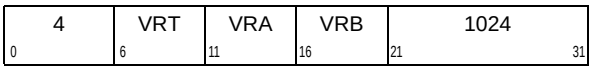
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Byte Modulo VX-form

vsububm VRT,VRA,VRB



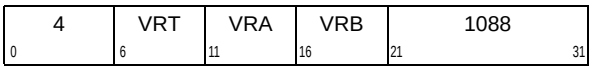

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop( aop +int ¬bop +int 1, 8 )
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRB is subtracted from unsigned-integer byte element i in VRA. The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Halfword Modulo VX-form

vsubuhm VRT,VRA,VRB



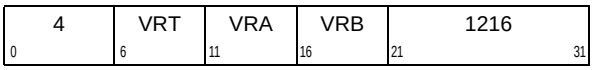

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+16 ← Chop( aop +int ¬bop +int 1, 16 )
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRB is subtracted from unsigned-integer halfword element i in VRA. The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Doubleword Modulo VX-form

vsubudm VRT,VRA,VRB




```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← Chop( aop +int ~bop +int 1, 64 )
end
```

For each integer value i from 0 to 1, do the following.
The integer value in doubleword element i of VR[VRB] is subtracted from the integer value in doubleword element i of VR[VRA].

The low-order 64 bits of the result are placed into doubleword element i of VR[VRT].

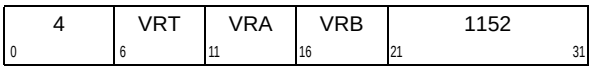
Special Registers Altered:
None

Programming Note

vsubudm can be used for signed or unsigned integers.

Vector Subtract Unsigned Word Modulo VX-form

vsubuwm VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop( aop +int ¬bop +int 1, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA. The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
None

Vector Subtract Unsigned Byte Saturate VX-form

vsububs VRT,VRA,VRB

4	VRT	VRA	VRB	1536
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Clamp(aop +int ¬bop +int 1, 0, 255)24:31
end
```

For each integer value i from 0 to 15, do the following.

Unsigned-integer byte element i in VRB is subtracted from unsigned-integer byte element i in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Halfword Saturate VX-form

vsubuhs VRT,VRA,VRB

4	VRT	VRA	VRB	1600
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int ¬bop +int 1,0,2^16-1)16:31
end
```

For each integer value i from 0 to 7, do the following.

Unsigned-integer halfword element i in VRB is subtracted from unsigned-integer halfword element i in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Word Saturate VX-form

vsubuws VRT,VRA,VRB

4	VRT	VRA	VRB	1664
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int ¬bop +int 1, 0, 2^32-1)
end
```

For each integer value i from 0 to 7, do the following.

Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA.

– If the intermediate result is less than 0 the result saturates to 0.

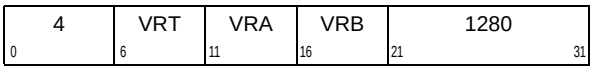
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Quadword Modulo VX-form

vsubuqm VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ← VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(1)
VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

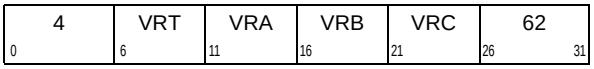
The rightmost 128 bits of the sum of src1, the one's complement of src2, and the value 1 are placed into VR[VRT].

Special Registers Altered:

None

Vector Subtract Extended Unsigned Quadword Modulo VA-form

vsubeuqm VRT,VRA,VRB,VRC




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(cin)
VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

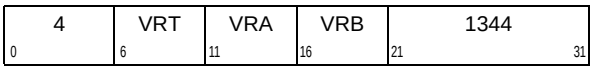
The rightmost 128 bits of the sum of src1, the one's complement of src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Vector Subtract & write Carry Unsigned Quadword VX-form

vsubcuq VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ←  VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(1)
VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

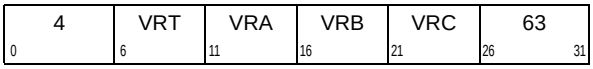
The carry out of the sum of src1, the one's complement of src2, and the value 1 is placed into VR[VRT].

Special Registers Altered:

None

Vector Subtract Extended & write Carry Unsigned Quadword VA-form

vsubecuq VRT,VRA,VRB,VRC




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(cin)
VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, the one's complement of src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Programming Note

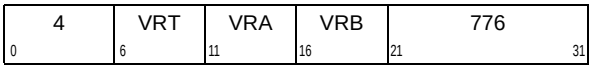
The *Vector Subtract Unsigned Quadword* instructions support efficient wide-integer subtraction. The following code sequence can be used to implement a 512-bit signed or unsigned subtract operation.

```
vsubuqm     vS3,vA3,vB3          # bits 384:511 of difference
vsubcuq     vC3,vA3,vB3          # carry out of bit 384 of difference
vsubeuqm    vS2,vA2,vB2,vC3      # bits 256:383 of difference
vsubecuq    vC2,vA2,vB2,vC3      # carry out of bit 256 of difference
vsubeuqm    vS1,vA1,vB1,vC2      # bits 128:255 of difference
vsubecuq    vC1,vA1,vB1,vC2      # carry out of bit 128 of difference
vsubeuqm    vS0,vA0,vB0,vC1      # bits 0:127 of difference
```

5.9.1.3 Vector Integer Multiply Instructions

Vector Multiply Even Signed Byte VX-form

vmulesb VRT,VRA,VRB




```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i:i+7) xsi EXTS((VRB)i:i+7)
   VRTi:i+15 ← Chop( prod, 16 )
end
```

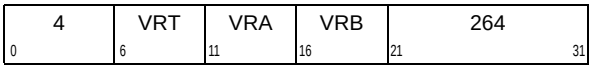
For each integer value i from 0 to 7, do the following. Signed-integer byte element i×2 in VRA is multiplied by signed-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Odd Signed Byte VX-form

vmulosb VRT,VRA,VRB




```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i+8:i+15) xsi EXTS((VRB)i+8:i+15)
   VRTi:i+15 ← Chop( prod, 16 )
end
```

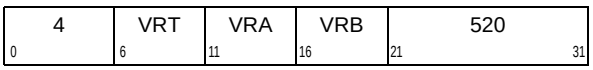
For each integer value i from 0 to 7, do the following. Signed-integer byte element i×2+1 in VRA is multiplied by signed-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Even Unsigned Byte VX-form

vmuleub VRT,VRA,VRB




```
do i=0 to 127 by 16
   prod ← EXTZ((VRA)i:i+7) xui EXTZ((VRB)i:i+7)
   VRTi:i+15 ← Chop(prod, 16)
end
```

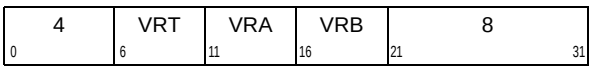
For each integer value i from 0 to 7, do the following. Unsigned-integer byte element i×2 in VRA is multiplied by unsigned-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Odd Unsigned Byte VX-form

vmuloub VRT,VRA,VRB




```
do i=0 to 127 by 16
   prod ← EXTZ((VRA)i+8:i+15) xui EXTZ((VRB)i+8:i+15)
   VRTi:i+15 ← Chop( prod, 16 )
end
```

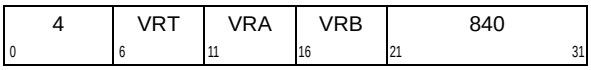
For each integer value i from 0 to 7, do the following. Unsigned-integer byte element i×2+1 in VRA is multiplied by unsigned-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Even Signed Halfword VX-form

vmulesh VRT,VRA,VRB




```
do i=0 to 127 by 32
   prod ← EXTS((VRA)i:i+15) xsi EXTS((VRB)i:i+15)
   VRTi:i+31 ← Chop( prod, 32 )
end
```

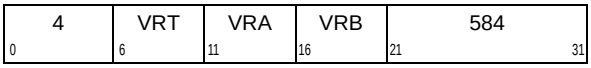
For each integer value i from 0 to 3, do the following. Signed-integer halfword element i×2 in VRA is multiplied by signed-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Even Unsigned Halfword VX-form

vmuleuh VRT,VRA,VRB




```
do i=0 to 127 by 32
   prod ← EXTZ((VRA)i:i+15) xui EXTZ((VRB)i:i+15)
   VRTi:i+31 ← Chop(prod, 32)
end
```

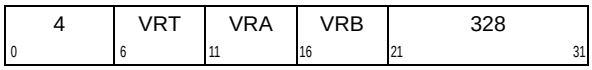
For each integer value i from 0 to 3, do the following. Unsigned-integer halfword element i×2 in VRA is multiplied by unsigned-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Odd Signed Halfword VX-form

vmulosh VRT,VRA,VRB




```
do i=0 to 127 by 32
   prod ← EXTS((VRA)i+16:i+31) xsi EXTS((VRB)i+16:i+31)
   VRTi:i+31 ← Chop( prod, 32 )
end
```

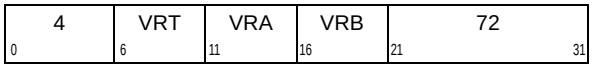
For each integer value i from 0 to 3, do the following. Signed-integer halfword element i×2+1 in VRA is multiplied by signed-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Odd Unsigned Halfword VX-form

vmulouh VRT,VRA,VRB




```
do i=0 to 127 by 32
   prod ← EXTZ((VRA)i+16:i+31)xui EXTZ((VRB)i+16:i+31)
   VRTi:i+31 ← Chop( prod, 32 )
end
```

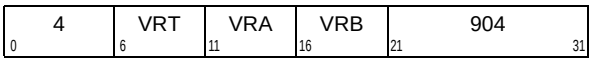
For each integer value i from 0 to 3, do the following. Unsigned-integer halfword element i×2+1 in VRA is multiplied by unsigned-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:

None

Vector Multiply Even Signed Word VX-form

vmulesw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i]
   src2 ← VR[VRB].word[2×i]
   VR[VRT].dword[i] ← src1 ×si src2
end
```

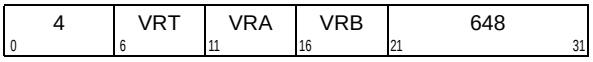
For each integer value i from 0 to 1, do the following.
The signed integer in word element 2×i of VR[VRA] is multiplied by the signed integer in word element 2×i of VR[VRB].

The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Multiply Even Unsigned Word VX-form

vmuleuw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i]
   src2 ← VR[VRB].word[2×i]
   VR[VRT].dword[i] ← src1 ×ui src2
end
```

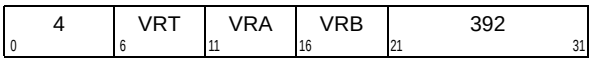
For each integer value i from 0 to 1, do the following.
The unsigned integer in word element 2×i of VR[VRA] is multiplied by the unsigned integer in word element 2×i of VR[VRB].

The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Multiply Odd Signed Word VX-form

vmulosw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i+1]
   src2 ← VR[VRB].word[2×i+1]
   VR[VRT].dword[i] ← src1 ×si src2
end
```

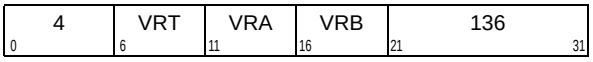
For each integer value i from 0 to 1, do the following.
The signed integer in word element 2×i+1 of VR[VRA] is multiplied by the signed integer in word element 2×i+1 of VR[VRB].

The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Multiply Odd Unsigned Word VX-form

vmulouw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i+1]
   src2 ← VR[VRB].word[2×i+1]
   VR[VRT].dword[i] ← src1 ×ui src2
end
```

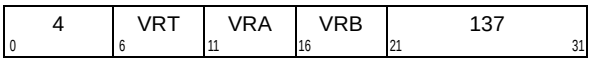
For each integer value i from 0 to 1, do the following.
The unsigned integer in word element 2×i+1 of VR[VRA] is multiplied by the unsigned integer in word element 2×i+1 of VR[VRB].

The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Multiply Unsigned Word Modulo
VX-form

vmuluwm VRT,VRA,VRB




```
do i = 0 to 3
   src1 ← VR[VRA].word[i]
   src2 ← VR[VRB].word[i]
   VR[VRT].word[i] ← Chop( src1 x_ui src2, 32 )
end
```

The integer in word element i of VR[VRA] is multiplied by the integer in word element i of VR[VRB].

The least-significant 32 bits of the product are placed into word element i of VR[VRT].

Special Registers Altered:

None

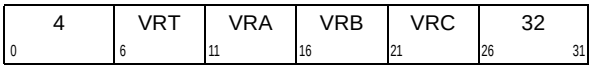
Programming Note

vmuluwm can be used for unsigned or signed integers.

5.9.1.4 Vector Integer Multiply-Add/Sum Instructions

Vector Multiply-High-Add Signed Halfword Saturate VA-form

vmhaddshs VRT,VRA,VRB,VRC




```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i:i+15) xsi EXTS((VRB)i:i+15)
   sum  ← (prod >>si 15) +int EXTS((VRC)i:i+15)
   VRTi:i+15 ← Clamp(sum, -2^15, 2^15-1)16:31
end
```

For each vector element i from 0 to 7, do the following.

Signed-integer halfword element i in VRA is multiplied by signed-integer halfword element i in VRB, producing a 32-bit signed-integer product. Bits 0:16 of the product are added to signed-integer halfword element i in VRC.

- If the intermediate result is greater than 2^{15}-1 the result saturates to 2^{15}-1.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

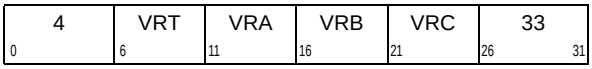
The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form

vmhraddshs VRT,VRA,VRB,VRC




```
do i=0 to 127 by 16
   temp ← EXTS((VRC)i:i+15)
   prod ← EXTS((VRA)i:i+15) xsi EXTS((VRB)i:i+15)
   sum  ← ((prod +int 0x0000_4000) >>si 15) +int temp
   VRTi:i+15 ← Clamp(sum, -2^15, 2^15-1)16:31
end
```

For each vector element i from 0 to 7, do the following.

Signed-integer halfword element i in VRA is multiplied by signed-integer halfword element i in VRB, producing a 32-bit signed-integer product. The value 0x0000_4000 is added to the product, producing a 32-bit signed-integer sum. Bits 0:16 of the sum are added to signed-integer halfword element i in VRC.

- If the intermediate result is greater than 2^{15}-1 the result saturates to 2^{15}-1.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form

vmladduhm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	34
0	6	11	16	21	26 31

```
do i=0 to 127 by 16
   prod ← EXTZ((VRA)i:i+15) ×ui EXTZ((VRB)i:i+15)
   sum  ← Chop( prod, 16 ) +int (VRC)i:i+15
   VRTi:i+15 ← Chop( sum, 16 )
end
```

For each integer value i from 0 to 3, do the following.

Unsigned-integer halfword element i in VRA is multiplied by unsigned-integer halfword element i in VRB, producing a 32-bit unsigned-integer product. The low-order 16 bits of the product are added to unsigned-integer halfword element i in VRC.

The low-order 16 bits of the sum are placed into halfword element i of VRT.

Special Registers Altered:

None

Programming Note

vmladduhm can be used for unsigned or signed-integers.

Vector Multiply-Sum Unsigned Byte Modulo VA-form

vmsumubm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	36
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 8
      prod ← EXTZ((VRA)i+j:i+j+7) ×ui EXTZ((VRB)i+j:i+j+7)
      temp ← temp +int prod
   end
   VRTi:i+31 ← Chop( temp, 32 )
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the four unsigned-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing an unsigned-integer halfword product.

– The sum of these four unsigned-integer halfword products is added to the unsigned-integer word element in VRC.

– The unsigned-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Mixed Byte Modulo VA-form

vmsummbm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	37
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← (VRC)i:i+31
   do j=0 to 31 by 8
      prod0:15 ← (VRA)i+j:i+j+7 xsui (VRB)i+j:i+j+7
      temp     ← temp +int EXTS(prod)
   end
   VRTi:i+31 ← temp
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the four signed-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing a signed-integer product.

– The sum of these four signed-integer halfword products is added to the signed-integer word element in VRC.

– The signed-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Signed Halfword Modulo VA-form

vmsumshm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	40
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← (VRC)i:i+31
   do j=0 to 31 by 16
      prod0:31 ← (VRA)i+j:i+j+15 xsi (VRB)i+j:i+j+15
      temp     ← temp +int prod
   end
   VRTi:i+31 ← temp
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

– The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

– The signed-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Signed Halfword Saturate VA-form

vmsumshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	41
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTS((VRC)i:i+31)
   do j=0 to 31 by 16
      srcA ← EXTS((VRA)i+j:i+j+15)
      srcB ← EXTS((VRB)i+j:i+j+15)
      prod ← srcA xsi srcB
      temp ← temp +int prod
   end
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

– The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1 and if it is less than -2^{31} it saturates to -2^{31}.

– The result is placed into the corresponding word element of VRT.

Special Registers Altered:

SAT

Vector Multiply-Sum Unsigned Halfword Modulo VA-form

vmsumuhm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	38
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 16
      srcA ← EXTZ((VRA)i+j:i+j+15)
      srcB ← EXTZ((VRB)i+j:i+j+15)
      prod ← srcA xui srcB
      temp ← temp +int prod
   end
   VRTi:i+31 ← Chop( temp, 32 )
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer word product.

– The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

– The unsigned-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Unsigned Halfword Saturate VA-form

vmsumuhs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	39
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 16
      src1 ← EXTZ((VRA)i+j:i+j+15)
      src2 ← EXTZ((VRB)i+j:i+j+15)
      prod ← src1 xui src2
   end
   temp ← temp +int prod
   VRTi:i+31 ← Clamp(temp, 0, 2^32-1)
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer product.

– The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

– If the intermediate result is greater than 2^{32}-1 the result saturates to 2^{32}-1.

– The result is placed into the corresponding word element of VRT.

Special Registers Altered:

SAT

5.9.1.5 Vector Integer Sum-Across Instructions

Vector Sum across Signed Word Saturate VX-form

vsumsws VRT,VRA,VRB

4	VRT	VRA	VRB	1928
0	6	11	16	21 31

```
temp ← EXTS((VRB)96:127)
do i=0 to 127 by 32
   temp ← temp +int EXTS((VRA)i:i+31)
end
VRT0:31   ← 0x0000_0000
VRT32:63  ← 0x0000_0000
VRT64:95  ← 0x0000_0000
VRT96:127 ← Clamp(temp, -2^31, 2^31-1)
```

The sum of the four signed-integer word elements in VRA is added to signed-integer word element 3 of VRB.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-end 32 bits of the result are placed into word element 3 of VRT.

Word elements 0 to 2 of VRT are set to 0.

Special Registers Altered:

SAT

Vector Sum across Half Signed Word Saturate VX-form

vsum2sws VRT,VRA,VRB

4	VRT	VRA	VRB	1672
0	6	11	16	21 31

```
do i=0 to 127 by 64
   temp ← EXTS((VRB)i+32:i+63)
   do j=0 to 63 by 32
      temp ← temp +int EXTS((VRA)i+j:i+j+31)
   end
   VRTi:i+63 ← 0x0000_0000 || Clamp(temp, -2^31, 2^31-1)
end
```

Word elements 0 and 2 of VRT are set to 0.

The sum of the signed-integer word elements 0 and 1 in VRA is added to the signed-integer word element in bits 32:63 of VRB.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into word element 1 of VRT.

The sum of signed-integer word elements 2 and 3 in VRA is added to the signed-integer word element in bits 96:127 of VRB.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

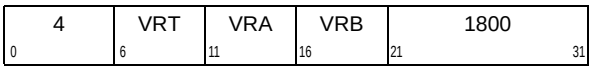
The low-order 32 bits of the result are placed into word element 3 of VRT.

Special Registers Altered:

SAT

Vector Sum across Quarter Signed Byte Saturate VX-form

vsum4sbs VRT,VRA,VRB




```
do i=0 to 127 by 32
   temp ← EXTS((VRB)i:i+31)
   do j=0 to 31 by 8
      temp ← temp +int EXTS((VRA)i+j:i+j+7)
   end
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

The sum of the four signed-integer byte elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

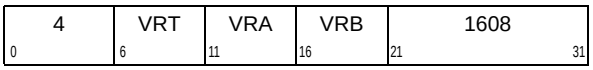
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Sum across Quarter Signed Halfword Saturate VX-form

vsum4shs VRT,VRA,VRB




```
do i=0 to 127 by 32
   temp ← EXTS((VRB)i:i+31)
   do j=0 to 31 by 16
      temp ← temp +int EXTS((VRA)i+j:i+j+15)
   end
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

The sum of the two signed-integer halfword elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into the corresponding word element of VRT.

Special Registers Altered:

SAT

Vector Sum across Quarter Unsigned Byte Saturate VX-form

vsum4ubs VRT,VRA,VRB

4	VRT	VRA	VRB	1544
0	6	11	16	21 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRB)i:i+31)
   do j=0 to 31 by 8
      temp ← temp +int EXTZ((VRA)i+j:i+j+7)
   end
   VRTi:i+31 ← Clamp( temp, 0, 2^32-1 )
end
```

For each integer value i from 0 to 3, do the following.

The sum of the four unsigned-integer byte elements contained in word element i of VRA is added to unsigned-integer word element i in VRB.

- If the intermediate result is greater than 2^{32}-1 it saturates to 2^{32}-1.

The low-order 32 bits of the result are placed into word element i of VRT.

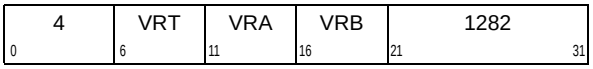
Special Registers Altered:

SAT

5.9.1.6 Vector Integer Average Instructions

Vector Average Signed Byte VX-form

vavgsb VRT,VRA,VRB




```
do i=0 to 127 by 8
        aop ← EXTS((VRA)i:i+7)
        bop ← EXTS((VRB)i:i+7)
        VRTi:i+7 ← Chop(( aop +int bop +int 1 ) >> 1, 8)
end
```

For each integer value i from 0 to 15, do the following. Signed-integer byte element i in VRA is added to signed-integer byte element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

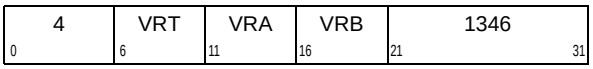
The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

None

Vector Average Signed Halfword VX-form

vavgsh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← Chop(( aop +int bop +int 1 ) >> 1, 16)
end
```

For each integer value i from 0 to 7, do the following. Signed-integer halfword element i in VRA is added to signed-integer halfword element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

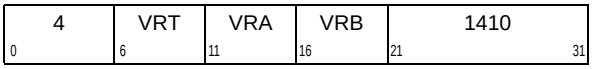
The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Average Signed Word VX-form

vavgsw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Chop(( aop +int bop +int 1 ) >> 1, 32)
end
```

For each integer value i from 0 to 3, do the following. Signed-integer word element i in VRA is added to signed-integer word element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

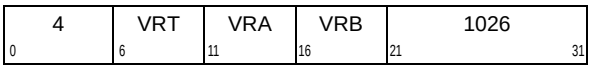
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

None

Vector Average Unsigned Byte VX-form

vavgub VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7
   VRTi:i+7 ← Chop((aop +int bop +int 1) >>ui 1, 8)
end
```

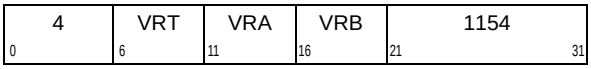
For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
None

Vector Average Unsigned Word VX-form

vavguw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop((aop +int bop +int 1) >>ui 1, 32)
end
```

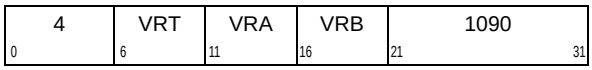
For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
None

Vector Average Unsigned Halfword VX-form

vavguh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Chop((aop +int bop +int 1) >>ui 1, 16)
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

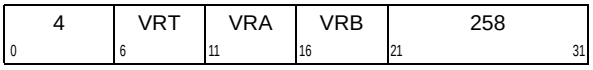
The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
None

5.9.1.7 Vector Integer Maximum and Minimum Instructions

Vector Maximum Signed Byte VX-form

vmaxsb VRT,VRA,VRB



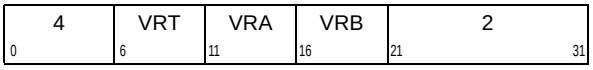

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← ( aop >si bop ) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is compared to signed-integer byte element i in VRB. The larger of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Byte VX-form

vmaxub VRT,VRA,VRB



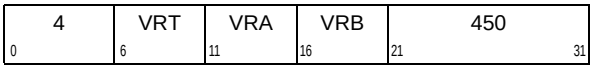

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← (aop >ui bop) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. The larger of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Maximum Signed Doubleword VX-form

vmaxsd VRT,VRA,VRB



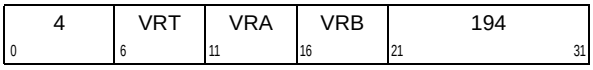

```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (aop >si bop) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following.
The signed integer value in doubleword element i of VR[VRA] is compared to the signed integer value in doubleword element i of VR[VRB]. The larger of the two values is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Maximum Unsigned Doubleword VX-form

vmaxud VRT,VRA,VRB



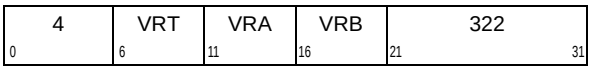

```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (aop >ui bop) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following.
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. The larger of the two values is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Maximum Signed Halfword VX-form

vmaxsh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15
   VRTi:i+15 ← ( aop >si bop ) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

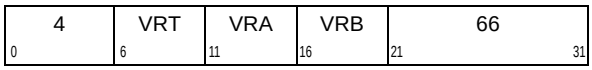
For each integer value i from 0 to 7, do the following. Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. The larger of the two values is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Maximum Unsigned Halfword VX-form

vmaxuh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← (aop >ui bop) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

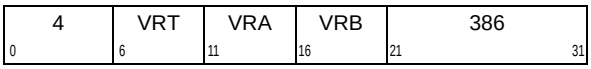
For each integer value i from 0 to 7, do the following. Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. The larger of the two values is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Maximum Signed Word VX-form

vmaxsw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← ( aop >si bop ) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

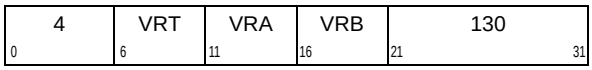
For each integer value i from 0 to 3, do the following. Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. The larger of the two values is placed into word element i of VRT.

Special Registers Altered:

None

Vector Maximum Unsigned Word VX-form

vmaxuw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← (aop >ui bop) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

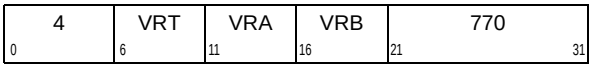
For each integer value i from 0 to 3, do the following. Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. The larger of the two values is placed into word element i of VRT.

Special Registers Altered:

None

Vector Minimum Signed Byte VX-form

vminsb VRT,VRA,VRB



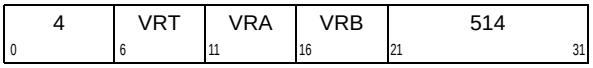

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← (aop <si bop) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is compared to signed-integer byte element i in VRB. The smaller of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Byte VX-form

vminub VRT,VRA,VRB



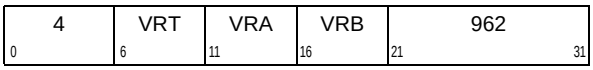

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7
   VRTi:i+7 ← ( aop <ui bop ) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. The smaller of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Minimum Signed Doubleword VX-form

vminsd VRT,VRA,VRB



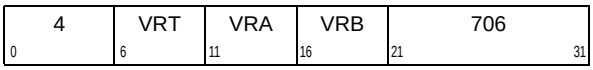

```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (ExtendSign(aop) <si ExtendSign(bop)) ?
aop : bop
end
```

For each integer value `i` from 0 to 1, do the following.
The signed integer value in doubleword element `i` of `VR[VRA]` is compared to the signed integer value in doubleword element `i` of `VR[VRB]`. The smaller of the two values is placed into doubleword element `i` of `VR[VRT]`.

Special Registers Altered:
None

Vector Minimum Unsigned Doubleword VX-form

vminud VRT,VRA,VRB




```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (aop <ui bop) ? aop : bop
end
```

For each integer value `i` from 0 to 1, do the following.
The unsigned integer value in doubleword element `i` of `VR[VRA]` is compared to the unsigned integer value in doubleword element `i` of `VR[VRB]`. The smaller of the two values is placed into doubleword element `i` of `VR[VRT]`.

Special Registers Altered:
None

Vector Minimum Signed Halfword VX-form

vminsh VRT,VRA,VRB

4	VRT	VRA	VRB	834
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← ( aop <si bop ) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

For each integer value i from 0 to 7, do the following. Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. The smaller of the two values is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Minimum Signed Word VX-form

vminsw VRT,VRA,VRB

4	VRT	VRA	VRB	898
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← ( aop <si bop ) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following. Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. The smaller of the two values is placed into word element i of VRT.

Special Registers Altered:

None

Vector Minimum Unsigned Halfword VX-form

vminuh VRT,VRA,VRB

4	VRT	VRA	VRB	578
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← ( aop <ui bop ) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

For each integer value i from 0 to 7, do the following. Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. The smaller of the two values is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Minimum Unsigned Word VX-form

vminuw VRT,VRA,VRB

4	VRT	VRA	VRB	642
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← ( aop <ui bop ) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following. Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. The smaller of the two values is placed into word element i of VRT.

Special Registers Altered:

None

5.9.2 Vector Integer Compare Instructions

The *Vector Integer Compare* instructions compare two Vector Registers element by element, interpreting the elements as unsigned or signed-integers depending on the instruction, and set the corresponding element of the target Vector Register to all 1s if the relation being tested is true and to all 0s if the relation being tested is false.

If Rc=1 CR Field 6 is set to reflect the result of the comparison, as follows.

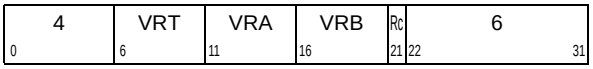
Bit	Description
0	The relation is true for all element pairs (i.e., VRT is set to all 1s)
1	0
2	The relation is false for all element pairs (i.e., VRT is set to all 0s)
3	0

Programming Note

***vcmpequb*[.], *vcmpequh*[.], *vcmpequw*[.], and *vcmpequd*[.]** can be used for unsigned or signed-integers.

Vector Compare Equal To Unsigned Byte VC-form

vcmpequb VRT,VRA,VRB (Rc=0)
vcmpequb. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 8
   VRT_{i:i+7} ← ((VRA)_{i:i+7} =int (VRB)_{i:i+7}) ? 8^1 : 8^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

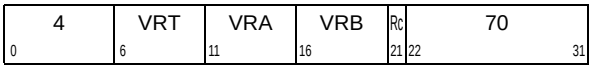
For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. Byte element i in VRT is set to all 1s if unsigned-integer byte element i in VRA is equal to unsigned-integer byte element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . .(if Rc=1)

Vector Compare Equal To Unsigned Halfword VC-form

vcmpequh VRT,VRA,VRB (Rc=0)
vcmpequh. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 16
   VRT_{i:i+15} ←((VRA)_{i:i+15} =int (VRB)_{i:i+15}) ? 16^1 : 16^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

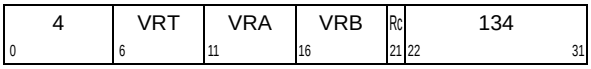
For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element element i in VRB. Halfword element i in VRT is set to all 1s if unsigned-integer halfword element i in VRA is equal to unsigned-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Equal To Unsigned Word VC-form

vcmpequw VRT,VRA,VRB (Rc=0)
vcmpequw. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_{i:i+31} ←((VRA)_{i:i+31} =int (VRB)_{i:i+31}) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

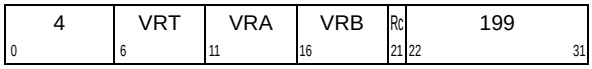
The unsigned integer value in word element i in VR[VRA] is compared to the unsigned integer value in word element i in VR[VRB]. Word element i in VR[VRT] is set to all 1s if unsigned-integer word element i in VR[VRA] is equal to unsigned-integer word element i in VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Equal To Unsigned Doubleword VX-form

vcmpequd VRT,VRA,VRB (Rc=0)
vcmpequd. VRT,VRA,VRB (Rc=1)




```
do i = 0 to 1
   aop ← EXTZ(VR[VRA].dword[i])
   bop ← EXTZ(VR[VRB].dword[i])
   if (aop = bop) then do
      VR[VRT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      flag.bit[i] ← 0b1
   end
   else do
      VR[VRT].dword[i] ← 0x0000_0000_0000_0000
      flag.bit[i] ← 0b0
   end
end
if Rc=1 then do
   CR.bit[24] ← (flag=0b11)
   CR.bit[25] ← 0b0
   CR.bit[26] ← (flag=0b00)
   CR.bit[27] ← 0b0
end
```

For each integer value i from 0 to 1, do the following.

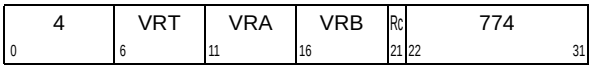
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. Doubleword element i of VR[VRT] is set to all 1s if the unsigned integer value in doubleword element i of VR[VRA] is equal to the unsigned integer value in doubleword element i of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Signed Byte VC-form

vcmpgtsb VRT,VRA,VRB (Rc=0)
vcmpgtsb. VRT,VRA,VRB (Rc=1)



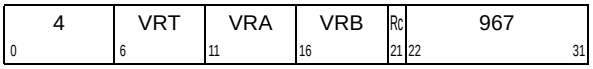

```
do i=0 to 127 by 8
   VRT_{i:i+7} ← ((VRA)_{i:i+7} >_si (VRB)_{i:i+7}) ? 8^1 : 8^0
end
if Rc=1 then do
   t ← (VRT=^128 1)
   f ← (VRT=^128 0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 15, do the following.
The signed integer value in byte element i in VR[VRA] is compared to the signed integer value in byte element i in VR[VRB]. Byte element i in VR[VRT] is set to all 1s if signed-integer byte element i in VR[VRA] is greater than to signed-integer byte element i in VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Signed Doubleword VX-form

vcmpgtsd VRT,VRA,VRB (Rc=0)
vcmpgtsd. VRT,VRA,VRB (Rc=1)



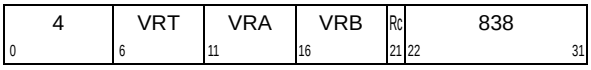

```
do i = 0 to 1
   aop ← EXTS(VR[VRA].dword[i])
   bop ← EXTS(VR[VRB].dword[i])
   if (aop >_si bop) then do
      VR[VRT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      flag.bit[i] ← 0b1
   end
   else do
      VR[VRT].dword[i] ← 0x0000_0000_0000_0000
      flag.bit[i] ← 0b0
   end
end
if "vcmpgtsd." then do
   CR.bit[24] ← (flag=0b11)
   CR.bit[25] ← 0b0
   CR.bit[26] ← (flag=0b00)
   CR.bit[27] ← 0b0
end
```

For each integer value i from 0 to 1, do the following.
The signed integer value in doubleword element i of VR[VRA] is compared to the signed integer value in doubleword element i of VR[VRB]. Doubleword element i of VR[VRT] is set to all 1s if the signed integer value in doubleword element i of VR[VRA] is greater than the signed integer value in doubleword element i of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Signed Halfword VC-form

vcmpgtsh VRT,VRA,VRB (Rc=0)
vcmpgtsh. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 16
   VRT_{i:i+15} ← ((VRA)_{i:i+15} >_si (VRB)_{i:i+15}) ? 16^1 : 16^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

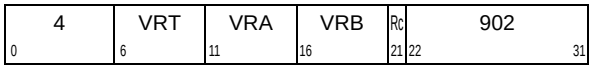
For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. Halfword element i in VRT is set to all 1s if signed-integer halfword element i in VRA is greater than signed-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . .(if Rc=1)

Vector Compare Greater Than Signed Word VC-form

vcmpgtsw VRT,VRA,VRB (Rc=0)
vcmpgtsw. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ((VRA)_{i:i+31} >_si (VRB)_{i:i+31}) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

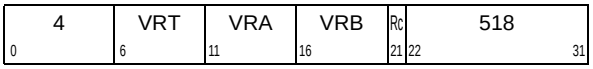
For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. Word element i in VRT is set to all 1s if signed-integer word element i in VRA is greater than signed-integer word element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Unsigned Byte VC-form

vcmpgtub VRT,VRA,VRB (Rc=0)
vcmpgtub. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 8
   VRT[i:i+7] ← ((VRA)[i:i+7] >ui (VRB)[i:i+7]) ? 8^1 : 8^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 15, do the following.

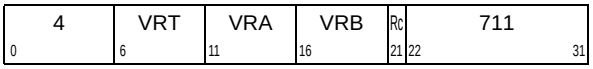
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. Byte element i in VRT is set to all 1s if unsigned-integer byte element i in VRA is greater than to unsigned-integer byte element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Unsigned Doubleword VX-form

vcmpgtud VRT,VRA,VRB (Rc=0)
vcmpgtud. VRT,VRA,VRB (Rc=1)




```
do i = 0 to 1
   aop ← EXTZ(VR[VRA].dword[i])
   bop ← EXTZ(VR[VRB].dword[i])
   if (ExtendZero(aop) >ui ExtendZero(bop)) then do
      VR[VRT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      flag.bit[i] ← 0b1
   end
   else do
      VR[VRT].dword[i] ← 0x0000_0000_0000_0000
      flag.bit[i] ← 0b1
   end
end
if "vcmpgtud." then do
   CR.bit[24] ← (flag=0b11)
   CR.bit[25] ← 0b0
   CR.bit[26] ← (flag=0b00)
   CR.bit[27] ← 0b0
end
```

For each integer value i from 0 to 1, do the following.

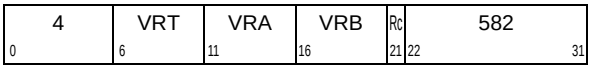
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. Doubleword element i of VR[VRT] is set to all 1s if the unsigned integer value in doubleword element i of VR[VRA] is greater than the unsigned integer value in doubleword element i of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 (if Rc=1)

Vector Compare Greater Than Unsigned Halfword VC-form

vcmpgtuh VRT,VRA,VRB (Rc=0)
vcmpgtuh. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 16
   VRT_{i:i+15} ← ((VRA)_{i:i+15} >_ui (VRB)_{i:i+15}) ? 16^1 : 16^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

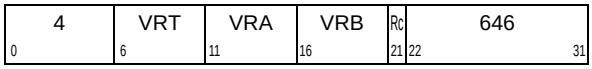
For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. Halfword element i in VRT is set to all 1s if unsigned-integer halfword element i in VRA is greater than to unsigned-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . .(if Rc=1)

Vector Compare Greater Than Unsigned Word VC-form

vcmpgtuw VRT,VRA,VRB (Rc=0)
vcmpgtuw. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ((VRA)_{i:i+31} >_ui (VRB)_{i:i+31}) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. Word element i in VRT is set to all 1s if unsigned-integer word element i in VRA is greater than to unsigned-integer word element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

5.9.3 Vector Logical Instructions

Extended mnemonics for vector logical operations

Extended mnemonics are provided that use the Vector OR and Vector NOR instructions to copy the contents of one Vector Register to another, with and without complementing. These are shown as examples with the two instructions.

Vector Move Register

Several vector instructions can be coded in a way such that they simply copy the contents of one Vector Register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Vy to register Vx.

vmr Vx,Vy (equivalent to: vor Vx,Vy,Vy)

Vector Complement Register

The *Vector NOR* instruction can be coded in a way such that it complements the contents of one Vector Register and places the result into another Vector Register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Vy and places the result into register Vx.

vnot Vx,Vy (equivalent to: vnor Vx,Vy,Vy)

Vector Logical AND VX-form

vand VRT,VRA,VRB

4	VRT	VRA	VRB	1028
0	6	11	16	21 31

```
VR[VRT] ← VR[VRA] & VR[VRB]
```

The contents of VR[VRA] are ANDed with the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical AND with Complement VX-form

vandc VRT,VRA,VRB

4	VRT	VRA	VRB	1092
0	6	11	16	21 31

```
VR[VRT] ← VR[VRA] & ¬VR[VRB]
```

The contents of VR[VRA] are ANDed with the complement of the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical Equivalent VX-form

veqv VRT,VRA,VRB

4	VRT	VRA	VRB	1668
0	6	11	16	21 31

```
VR[VRT] ← VR[VRA] ≡ VR[VRB]
```

The contents of VR[VRA] are XORed with the contents of VR[VRB] and the complemented result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical NAND VX-form

vnand VRT,VRA,VRB

4	VRT	VRA	VRB	1412
0	6	11	16	21 31

```
if MSR.VEC=0 then VECTOR_UNAVAILABLE()

VR[VRT] ← ¬( VR[VRA] & VR[VRB] )
```

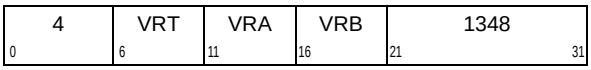
The contents of VR[VRA] are ANDed with the contents of VR[VRB] and the complemented result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical OR with Complement VX-form

vorc VRT,VRA,VRB



VR[VRT] ← VR[VRA] | ¬VR[VRB]

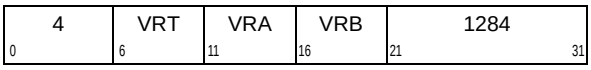
The contents of VR[VRA] are ORed with the complement of the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical NOR VX-form

vnor VRT,VRA,VRB



VR[VRT] ← ¬(VR[VRA] | VR[VRB])

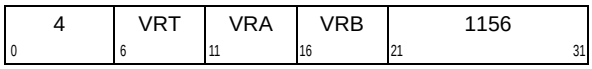
The contents of VR[VRA] are ORed with the contents of VR[VRB] and the complemented result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical OR VX-form

vor VRT,VRA,VRB



VR[VRT] ← VR[VRA] | VR[VRB]

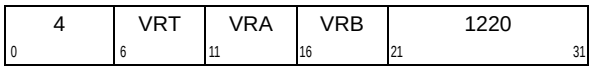
The contents of VR[VRA] are ORed with the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical XOR VX-form

vxor VRT,VRA,VRB



VR[VRT] ← VR[VRA] ⊕ VR[VRB]

The contents of VR[VRA] are XORed with the contents of VR[VRB] and the result is placed into VR[VRT].

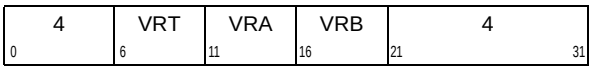
Special Registers Altered:

None

5.9.4 Vector Integer Rotate and Shift Instructions

Vector Rotate Left Byte VX-form

vrlb VRT,VRA,VRB




```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 <<< sh
end
```

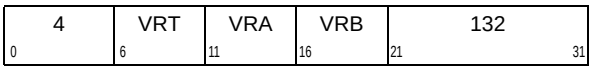
For each integer value i from 0 to 15, do the following.
Byte element i in VRA is rotated left by the number of bits specified in the low-order 3 bits of the corresponding byte element i in VRB.

The result is placed into byte element i in VRT.

Special Registers Altered:
None

Vector Rotate Left Word VX-form

vrlw VRT,VRA,VRB




```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 <<< sh
end
```

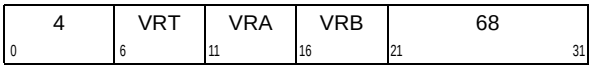
For each integer value i from 0 to 3, do the following.
Word element i in VRA is rotated left by the number of bits specified in the low-order 5 bits of the corresponding word element i in VRB.

The result is placed into word element i in VRT.

Special Registers Altered:
None

Vector Rotate Left Halfword VX-form

vrlh VRT,VRA,VRB




```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 <<< sh
end
```

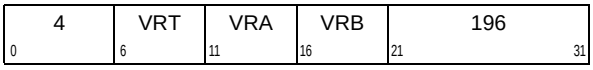
For each integer value i from 0 to 7, do the following.
Halfword element i in VRA is rotated left by the number of bits specified in the low-order 4 bits of the corresponding halfword element i in VRB.

The result is placed into halfword element i in VRT.

Special Registers Altered:
None

Vector Rotate Left Doubleword VX-form

vrld VRT,VRA,VRB




```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] <<< sh
end
```

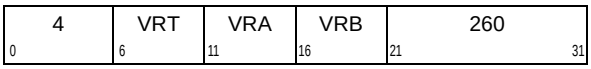
For each integer value i from 0 to 1, do the following.
The contents of doubleword element i of VR[VRA] are rotated left by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB].

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Shift Left Byte VX-form

vslb VRT,VRA,VRB




```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 << sh
end
```

For each integer value i from 0 to 15, do the following.
Byte element i in VRA is shifted left by the number of bits specified in the low-order 3 bits of byte element i in VRB.

– Bits shifted out of bit 0 are lost.

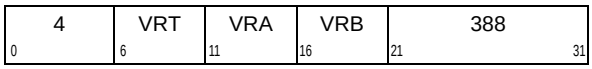
– Zeros are supplied to the vacated bits on the right.

The result is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Shift Left Word VX-form

vslw VRT,VRA,VRB




```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 << sh
end
```

For each integer value i from 0 to 3, do the following.
Word element i in VRA is shifted left by the number of bits specified in the low-order 5 bits of word element i in VRB.

– Bits shifted out of bit 0 are lost.

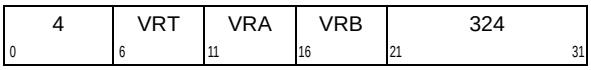
– Zeros are supplied to the vacated bits on the right.

The result is placed into word element i of VRT.

Special Registers Altered:
None

Vector Shift Left Halfword VX-form

vslh VRT,VRA,VRB




```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 << sh
end
```

For each integer value i from 0 to 7, do the following.
Halfword element i in VRA is shifted left by the number of bits specified in the low-order 4 bits of halfword element i in VRB.

– Bits shifted out of bit 0 are lost.

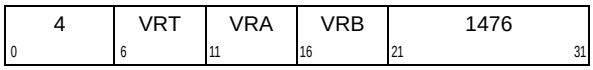
– Zeros are supplied to the vacated bits on the right.

The result is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Shift Left Doubleword VX-form

vsld VRT,VRA,VRB




```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] << sh
end
```

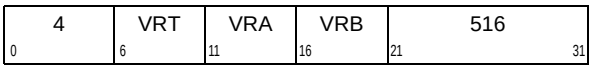
For each integer value i from 0 to 1, do the following.
The contents of doubleword element i of VR[VRA] are shifted left by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB].
– Bits shifted out of bit 0 are lost.
– Zeros are supplied to the vacated bits on the right.

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Shift Right Byte VX-form

vsrb VRT,VRA,VRB



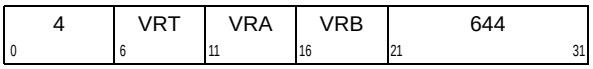

```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 >>ui sh
end
```

For each integer value i from 0 to 15, do the following.
Byte element i in VRA is shifted right by the number of bits specified in the low-order 3 bits of byte element i in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Shift Right Word VX-form

vsrw VRT,VRA,VRB



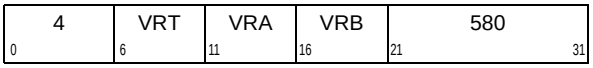

```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 >>ui sh
end
```

For each integer value i from 0 to 3, do the following.
Word element i in VRA is shifted right by the number of bits specified in the low-order 5 bits of word element i in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into word element i of VRT.

Special Registers Altered:
None

Vector Shift Right Halfword VX-form

vsrh VRT,VRA,VRB



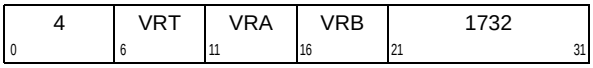

```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 >>ui sh
end
```

For each integer value i from 0 to 7, do the following.
Halfword element i in VRA is shifted right by the number of bits specified in the low-order 4 bits of halfword element i in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Shift Right Doubleword VX-form

vsrd VRT,VRA,VRB




```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] >>ui sh
end
```

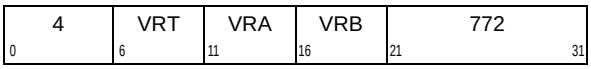
For each integer value i from 0 to 1, do the following.
The contents of doubleword element i of VR[VRA] are shifted right by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB]. Zeros are supplied to the vacated bits on the left.

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Shift Right Algebraic Byte VX-form

vsrab VRT,VRA,VRB




```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 >>si sh
end
```

For each integer value i from 0 to 15, do the following.

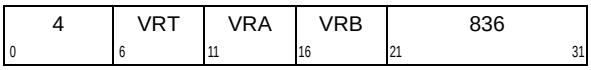
Byte element i in VRA is shifted right by the number of bits specified in the low-order 3 bits of the corresponding byte element i in VRB. Bits shifted out of bit 7 of the byte element are lost. Bit 0 of the byte element is replicated to fill the vacated bits on the left. The result is placed into byte element i of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Halfword VX-form

vsrah VRT,VRA,VRB




```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 >>si sh
end
```

For each integer value i from 0 to 7, do the following.

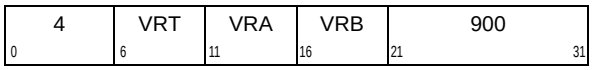
Halfword element i in VRA is shifted right by the number of bits specified in the low-order 4 bits of the corresponding halfword element i in VRB. Bits shifted out of bit 15 of the halfword are lost. Bit 0 of the halfword is replicated to fill the vacated bits on the left. The result is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Word VX-form

vsraw VRT,VRA,VRB




```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 >>si sh
end
```

For each integer value i from 0 to 3, do the following.

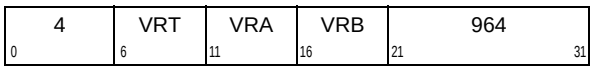
Word element i in VRA is shifted right by the number of bits specified in the low-order 5 bits of the corresponding word element i in VRB. Bits shifted out of bit 31 of the word are lost. Bit 0 of the word is replicated to fill the vacated bits on the left. The result is placed into word element i of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Doubleword VX-form

vsrad VRT,VRA,VRB




```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] >>si sh
end
```

For each integer value i from 0 to 1, do the following.

The contents of doubleword element i of VR[VRA] are shifted right by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB]. Bit 0 of doubleword element i of VR[VRA] is replicated to fill the vacated bits on the left.

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:

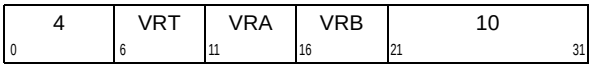
None

5.10 Vector Floating-Point Instruction Set

5.10.1 Vector Floating-Point Arithmetic Instructions

Vector Add Single-Precision VX-form

vaddfp VRT,VRA,VRB




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← RoundToNearSP((VRA)_{i:i+31} +fp (VRB)_{i:i+31})
end
```

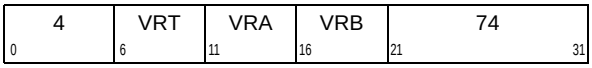
For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is added to single-precision floating-point element i in VRB. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

Special Registers Altered:

None

Vector Subtract Single-Precision VX-form

vsubfp VRT,VRA,VRB




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← RoundToNearSP((VRA)_{i:i+31} -fp (VRB)_{i:i+31})
end
```

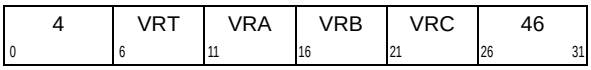
For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is subtracted from single-precision floating-point element i in VRA. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

Special Registers Altered:

None

Vector Multiply-Add Single-Precision VA-form

vmaddfp VRT,VRA,VRC,VRB




```
do i=0 to 127 by 32
   prod ← (VRA)i:i+31 xfp (VRC)i:i+31
   VRTi:i+31 ← RoundToNearSP( prod +fp (VRB)i:i+31 )
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is multiplied by single-precision floating-point element i in VRC. Single-precision floating-point element i in VRB is added to the infinitely-precise product. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

Special Registers Altered:

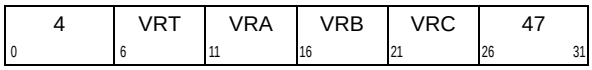
None

Programming Note

To use a multiply-add to perform an IEEE or Java compliant multiply, the addend must be -0.0. This is necessary to insure that the sign of a zero result will be correct when the product is -0.0 (+0.0 + -0.0 ≥ +0.0, and -0.0 + -0.0≥ -0.0). When the sign of a resulting 0.0 is not important, then +0.0 can be used as an addend which may, in some cases, avoid the need for a second register to hold a -0.0 in addition to the integer 0/floating-point +0.0 that may already be available.

Vector Negative Multiply-Subtract Single-Precision VA-form

vnmsubfp VRT,VRA,VRC,VRB




```
do i=0 to 127 by 32
   prod0:inf ← (VRA)i:i+31 xfp (VRC)i:i+31
   VRTi:i+31 ← -RoundToNearSP(prod0:inf -fp (VRB)i:i+31)
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is multiplied by single-precision floating-point element i in VRC. Single-precision floating-point element i in VRB is subtracted from the infinitely-precise product. The intermediate result is rounded to the nearest single-precision floating-point number, then negated and placed into word element i of VRT.

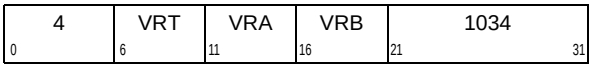
Special Registers Altered:

None

5.10.2 Vector Floating-Point Maximum and Minimum Instructions

Vector Maximum Single-Precision VX-form

vmaxfp VRT,VRA,VRB




```
do i=0 to 127 by 32
   gt_flag ← ( (VRA)i:i+31 >fp (VRB)i:i+31 )
   VRTi:i+31 ← gt_flag ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. The larger of the two values is placed into word element i of VRT.

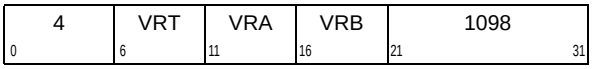
The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

Special Registers Altered:

None

Vector Minimum Single-Precision VX-form

vminfp VRT,VRA,VRB




```
do i=0 to 127 by 32
   lt_flag ← ( (VRA)i:i+31 <fp (VRB)i:i+31 )
   VRTi:i+31 ← lt_flag ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. The smaller of the two values is placed into word element i of VRT.

The minimum of +0 and -0 is -0. The minimum of any value and a NaN is a QNaN.

Special Registers Altered:

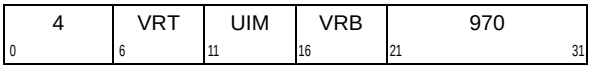
None

5.10.3 Vector Floating-Point Rounding and Conversion Instructions

See Appendix C, “Vector RTL Functions [Category: Vector]” on page 701, for RTL function descriptions.

Vector Convert To Signed Fixed-Point Word Saturate VX-form

vctsxs VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT[i:i+31] ← ConvertSPtoSXWsaturate((VRB)[i:i+31], UIM)
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRB is multiplied by 2^{UIM}. The product is converted to a 32-bit signed fixed-point integer using the rounding mode Round toward Zero.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The result is placed into word element i of VRT.

Special Registers Altered:

SAT

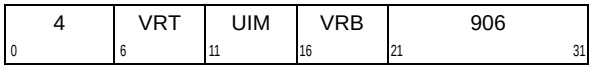
Extended Mnemonics:

Example of an extended mnemonics for *Vector Convert to Signed Fixed-Point Word Saturate*:

Extended:		Equivalent to:	
vcfpsxws	VRT,VRB,UIM	vctsxs	VRT,VRB,UIM

Vector Convert To Unsigned Fixed-Point Word Saturate VX-form

vctuxs VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT[i:i+31] ← ConvertSPtoUXWsaturate((VRB)[i:i+31], UIM)
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRB is multiplied by 2^{UIM}. The product is converted to a 32-bit unsigned fixed-point integer using the rounding mode Round toward Zero.

– If the intermediate result is greater than 2^{32}-1 the result saturates to 2^{32}-1.

The result is placed into word element *i* of VRT.

Special Registers Altered:

SAT

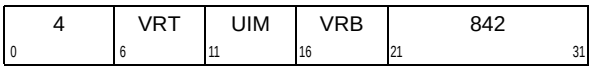
Extended Mnemonics:

Example of an extended mnemonics for *Vector Convert to Unsigned Fixed-Point Word Saturate*:

Extended:		Equivalent to:	
vcfpuxws	VRT,VRB,UIM	vctuxs	VRT,VRB,UIM

Vector Convert From Signed Fixed-Point Word VX-form

vcfsx VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ConvertSXWtoSP( (VRB)_{i:i+31} ) ÷_fp 2^UIM
end
```

For each integer value i from 0 to 3, do the following.
Signed fixed-point word element i in VRB is converted to the nearest single-precision floating-point value. Each result is divided by 2^{UIM} and placed into word element *i* of VRT.

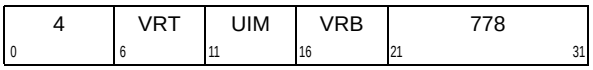
Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for *Vector Convert from Signed Fixed-Point Word*

Extended:		Equivalent to:	
vcsxwfp	VRT,VRB,UIM	vcfsx	VRT,VRB,UIM

Vector Convert From Unsigned Fixed-Point Word VX-form

vcfux VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ConvertUXWtoSP( (VRB)_{i:i+31} ) ÷_fp 2^UIM
end
```

For each integer value i from 0 to 3, do the following.
Unsigned fixed-point word element i in VRB is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIM} and placed into word element *i* of VRT.

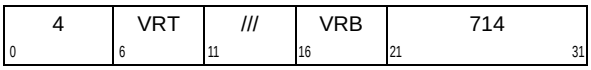
Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for *Vector Convert from Unsigned Fixed-Point Word*

Extended:		Equivalent to:	
vcuxwfp	VRT,VRB,UIM	vcfux	VRT,VRB,UIM

Vector Round to Single-Precision Integer toward -Infinity VX-form

vrfim VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntFloor( (VRB)0:31 )
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward -Infinity.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Programming Note

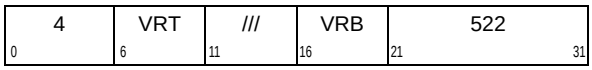
The *Vector Convert To Fixed-Point Word* instructions support only the rounding mode Round toward Zero. A floating-point number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate *Vector Round to Floating-Point Integer* instruction before the *Vector Convert To Fixed-Point Word* instruction.

Programming Note

The fixed-point integers used by the *Vector Convert* instructions can be interpreted as consisting of 32-UIM integer bits followed by UIM fraction bits.

Vector Round to Single-Precision Integer Nearest VX-form

vrfin VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntNear( (VRB)0:31 )
end
```

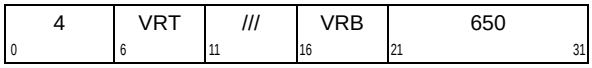
For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round to Nearest.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Vector Round to Single-Precision Integer toward +Infinity VX-form

vrfip VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntCeil( (VRB)0:31 )
end
```

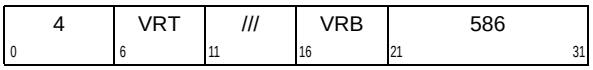
For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward +Infinity.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Vector Round to Single-Precision Integer toward Zero VX-form

vrfiz VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntTrunc( (VRB)0:31 )
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward Zero.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:

None

5.10.4 Vector Floating-Point Compare Instructions

The *Vector Floating-Point Compare* instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the *Vector Compare Bounds Floating-Point* instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the *Vector Integer Compare* instructions; see Section 5.9.2.

The *Vector Compare Bounds Floating-Point* instruction sets the target Vector Register, and CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be "within the bounds" specified by a single-precision floating-point value y if $-y \le x \le y$.

Vector Compare Bounds Single-Precision VC-form

vcmpbfp VRT,VRA,VRB (Rc=0)
vcmpbfp. VRT,VRA,VRB (Rc=1)

4	VRT	VRA	VRB	Rc	966
0	6	11	16	21	22 31

```
do i=0 to 127 by 32
   le ← ( (VRA)i:i+31 ≤fp (VRB)i:i+31 )
   ge ← ( (VRA)i:i+31 ≥fp -(VRB)i:i+31 )
   VRTi:i+31 ← ¬le || ¬ge || 30 0
end
if Rc=1 then do
   ib ← (VRT=128 0)
   CR6 ← 0b00 || ib || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRA is compared to single-precision floating-point word element i in VRB. A 2-bit value is formed that indicates whether the element in VRA is within the bounds specified by the element in VRB, as follows.

– Bit 0 of the 2-bit value is set to 0 if the element in VRA is less than or equal to the element in VRB, and is set to 1 otherwise.

– Bit 1 of the 2-bit value is set to 0 if the element in VRA is greater than or equal to the negation of the element in VRB, and is set to 1 otherwise.

The 2-bit value is placed into the high-order two bits of word element *i* of VRT and the remaining bits of element *i* are set to 0.

If Rc=1, CR field 6 is set as follows.

Bit	Description
0	Set to 0
1	Set to 0
2	Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.
3	Set to 0

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

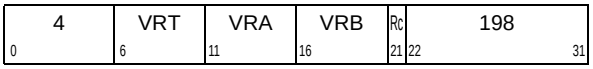
Programming Note

Each single-precision floating-point word element in VRB should be non-negative; if it is negative, the corresponding element in VRA will necessarily be out of bounds.

One exception to this is when the value of an element in VRB is -0.0 and the value of the corresponding element in VRA is either +0.0 or -0.0. +0.0 and -0.0 compare equal to -0.0.

Vector Compare Equal To Single-Precision VC-form

vcmpeqfp VRT,VRA,VRB (Rc=0)
vcmpeqfp. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ((VRA)_{i:i+31} =_fp (VRB)_{i:i+31}) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← ( VRT=128^1 )
   f ← ( VRT=128^0 )
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. Word element i in VRT is set to all 1s if single-precision floating-point element i in VRA is equal to single-precision floating-point element i in VRB, and is set to all 0s otherwise.

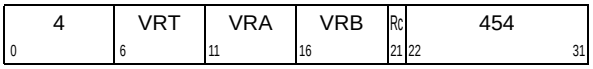
If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not equal to”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 1s, indicating “equal to”.

Special Registers Altered:

CR field 6 . . . . . .(if Rc=1)

Vector Compare Greater Than or Equal To Single-Precision VC-form

vcmpgefp VRT,VRA,VRB (Rc=0)
vcmpgefp. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_{i:i+31} ←((VRA)_{i:i+31} ≥_fp (VRB)_{i:i+31}) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← ( VRT=128^1 )
   f ← ( VRT=128^0 )
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. Word element i in VRT is set to all 1s if single-precision floating-point element i in VRA is greater than or equal to single-precision floating-point element i in VRB, and is set to all 0s otherwise.

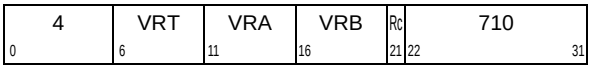
If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not greater than or equal to”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 1s, indicating “greater than or equal to”.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Single-Precision VC-form

vcmpgtfp VRT,VRA,VRB (Rc=0)
vcmpgtfp. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT[i:i+31] ← ((VRA)[i:i+31] >fp (VRB)[i:i+31]) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← ( VRT=128^1 )
   f ← ( VRT=128^0 )
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. Word element i in VRT is set to all 1s if single-precision floating-point element i in VRA is greater than single-precision floating-point element i in VRB, and is set to all 0s otherwise.

If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not greater than”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 0s, indicating “not greater than”.

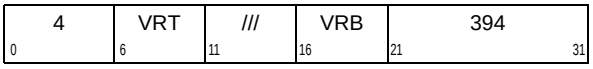
Special Registers Altered:

CR field 6 . . . . . .(if Rc=1)

5.10.5 Vector Floating-Point Estimate Instructions

Vector 2 Raised to the Exponent Estimate Floating-Point VX-form

vexptefp VRT,VRB




```
do i=0 to 127 by 32
   VRT_i:i+31 ← Power2EstimateSP( (VRB)_i:i+31 )
end
```

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of 2 raised to the power of single-precision floating-point element *i* in VRB is placed into word element *i* of VRT.

Let x be any single-precision floating-point input value. Unless x< -146 or the single-precision floating-point result of computing 2 raised to the power x would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16. The most significant 12 bits of the estimate's significand are monotonic. An integral input value returns an integral value when the result is representable.

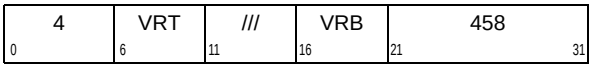
The result for various special cases of the source value is given below.

Value	Result
- Infinity	+0
-0	+1
+0	+1
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:
None

Vector Log Base 2 Estimate Floating-Point VX-form

vlogefp VRT,VRB




```
do i=0 to 127 by 32
   VRT_i:i+31 ← LogBase2EstimateSP((VRB)_i:i+31)
end
```

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of the base 2 logarithm of single-precision floating-point element *i* in VRB is placed into the corresponding word element of VRT.

Let x be any single-precision floating-point input value. Unless | x-1 | is less than or equal to 0.125 or the single-precision floating-point result of computing the base 2 logarithm of x would be an infinity or a QNaN, the estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than 2^{-5}. Under the same conditions, the estimate has a relative error in precision no greater than one part in 8.

The most significant 12 bits of the estimate's significand are monotonic. The estimate is exact if $x=2^y$, where y is an integer between -149 and +127 inclusive. Otherwise the value placed into the element of register VRT may vary between implementations, and between different executions on the same implementation.

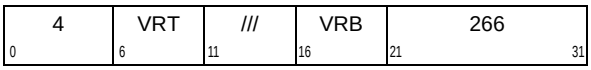
The result for various special cases of the source value is given below.

Value	Result
- Infinity	QNaN
< 0	QNaN
- 0	- Infinity
+0	- Infinity
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:
None

Vector Reciprocal Estimate Single-Precision VX-form

vrefp VRT,VRB




```
do i=0 to 127 by 32
   VRT[i:i+31] ← ReciprocalEstimateSP( (VRB)[i:i+31] )
end
```

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of the reciprocal of single-precision floating-point element i in VRB is placed into word element i of VRT.

Unless the single-precision floating-point result of computing the reciprocal of a value would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

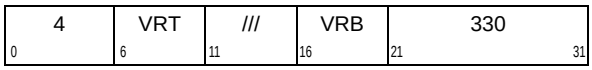
Value	Result
- Infinity	-0
- 0	- Infinity
+0	+ Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:

None

Vector Reciprocal Square Root Estimate Single-Precision VX-form

vrsqrtefp VRT,VRB




```
do i=0 to 127 by 32
   VRT[i:i+31] ← RecipSquareRootEstimateSP((VRB)[i:i+31])
end
```

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of the reciprocal of the square root of single-precision floating-point element i in VRB is placed into word element i of VRT.

Let x be any single-precision floating-point value. Unless the single-precision floating-point result of computing the reciprocal of the square root of x would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
- Infinity	QNaN
< 0	QNaN
- 0	- Infinity
+0	+ Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:

None

5.11 Vector Exclusive-OR-based Instructions

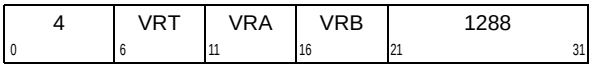
5.11.1 Vector AES Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 197 Advanced Encryption Standard for encryption and decryption.

Vector AES Cipher VX-form

[Category:Vector.Crypto]

vcipher VRT,VRA,VRB




```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← SubBytes(State)
vtemp2   ← ShiftRows(vtemp1)
vtemp3   ← MixColumns(vtemp2)
VR[VRT]  ← vtemp3 ^ RoundKey
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

One round of an AES cipher operation is performed on the intermediate State array, sequentially applying the transforms, SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the new intermediate state of the cipher operation.

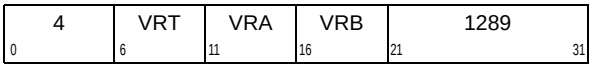
Special Registers Altered:

None

Vector AES Cipher Last VX-form

[Category:Vector.Crypto]

vcipherlast VRT,VRA,VRB




```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← SubBytes(State)
vtemp2   ← ShiftRows(vtemp1)
VR[VRT]  ← vtemp2 ^ RoundKey
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

The final round in an AES cipher operation is performed on the intermediate State array, sequentially applying the transforms, SubBytes(), ShiftRows(), AddRoundKey(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the final state of the cipher operation.

Special Registers Altered:

None

Vector AES Inverse Cipher VX-form
[Category:Vector.Crypto]

vncipher VRT,VRA,VRB

4	VRT	VRA	VRB	1352
0	6	11	16	21 31

```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← InvShiftRows(State)
vtemp2   ← InvSubBytes(vtemp1)
vtemp3   ← InvMixColumns(vtemp2)
VR[VRT]  ← vtemp3 ^ RoundKey
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

One round of an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), AddRoundKey(), and InvMixColumns(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the new intermediate state of the inverse cipher operation.

Special Registers Altered:
None

Vector AES Inverse Cipher Last VX-form
[Category:Vector.Crypto]

vncipherlast VRT,VRA,VRB

4	VRT	VRA	VRB	1353
0	6	11	16	21 31

```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← InvShiftRows(State)
vtemp2   ← InvSubBytes(vtemp1)
VR[VRT]  ← vtemp2 ^ RoundKey
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

The final round in an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), and AddRoundKey(), as defined in FIPS-197.

The result is placed into VR[VRT], representing the final state of the inverse cipher operation.

Special Registers Altered:
None

Vector AES SubBytes VX-form
[Category:Vector.Crypto]

vsbox VRT,VRA

4	VRT	VRA	///	1480
0	6	11	16	21 31

```
State   ← VR[VRA]
VR[VRT] ← SubBytes(State)
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

The result of applying the transform, SubBytes() on State, as defined in FIPS-197, is placed into VR[VRT].

Special Registers Altered:
None

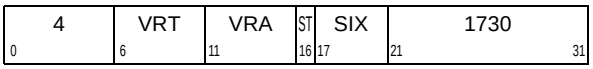
5.11.2 Vector SHA-256 and SHA-512 Sigma Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 180-3 Secure Hash Standard.

Vector SHA-512 Sigma Doubleword VX-form

[Category:Vector.Crypto]

vshasigmad VRT,VRA,ST,SIX




```
do i = 0 to 1
   src ← VR[VRA].doubleword[i]
   if ST=0 & SIX.bit[2×i]=0 then  // SHA-512 σ0 function
      VR[VRT].dword[i] ← (src >>>  1) ^
                         (src >>>  8) ^
                         (src >> 7)
   if ST=0 & SIX.bit[2×i]=1 then  // SHA-512 σ1 function
      VR[VRT].dword[i] ← (src >>> 19) ^
                         (src >>> 61) ^
                         (src >> 6)
   if ST=1 & SIX.bit[2×i]=0 then  // SHA-512 Σ0 function
      VR[VRT].dword[i] ← (src >>> 28) ^
                         (src >>> 34) ^
                         (src >>> 39)
   if ST=1 & SIX.bit[2×i]=1  then  // SHA-512 Σ1 function
      VR[VRT].dword[i] ← (src >>> 14) ^
                         (src >>> 18) ^
                         (src >>> 41)
end
```

For each integer value i from 0 to 1, do the following.

When ST=0 and bit 2×i of SIX is 0, a SHA-512 σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=0 and bit 2×i of SIX is 1, a SHA-512 σ1 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 0, a SHA-512 Σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 1, a SHA-512 Σ1 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

Bits 1 and 3 of SIX are reserved.

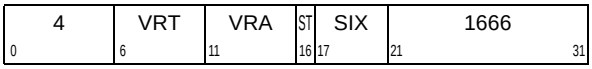
Special Registers Altered:

None

Vector SHA-256 Sigma Word VX-form

[Category:Vector.Crypto]

vshasigmaw VRT,VRA,ST,SIX




```
do i = 0 to 3
   src ← VR[VRA].word[i]
   if ST=0 & SIX.bit[i]=0 then  // SHA-256 σ0 function
      VR[VRT].word[i] ← (src >>>  7) ^
                        (src >>> 18) ^
                        (src >>  3)
   if ST=0 & SIX.bit[i]=1 then  // SHA-256 σ1 function
      VR[VRT].word[i] ← (src >>> 17) ^
                        (src >>> 19) ^
                        (src >> 10)
   if ST=1 & SIX.bit[i]=0 then  // SHA-256 Σ0 function
      VR[VRT].word[i] ← (src >>>  2) ^
                        (src >>> 13) ^
                        (src >>> 22)
   if ST=1 & SIX.bit[i]=1 then  // SHA-256 Σ1 function
      VR[VRT].word[i] ← (src >>>  6) ^
                        (src >>> 11) ^
                        (src >>> 25)
end
```

For each integer value i from 0 to 3, do the following.

When ST=0 and bit i of SIX is 0, a SHA-256 σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=0 and bit i of SIX is 1, a SHA-256 σ1 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=1 and bit i of SIX is 0, a SHA-256 Σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=1 and bit i of SIX is 1, a SHA-256 Σ1 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

Special Registers Altered:

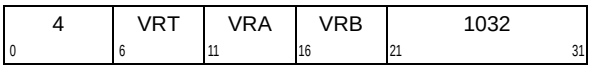
None

5.11.3 Vector Binary Polynomial Multiplication Instructions

This section describes a set of binary polynomial multiply-sum instructions. Corresponding elements are multiplied and the exclusive-OR of each even-odd pair of products summ, useful for a variety of finite field arithmetic operations.

Vector Polynomial Multiply-Sum Byte VX-form

vpmsumb VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 15
   prod[i].bit[0:14] ← 0
   srcA ← VR[VRA].byte[i]
   srcB ← VR[VRB].byte[i]
   do j = 0 to 7
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 8 to 14
      do k = j-7 to 7
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
do i = 0 to 7
   VR[VRT].hword[i] ← 0b0 || (prod[2×i] ^ prod[2×i+1])
end
```

For each integer value i from 0 to 15, do the following.

Let prod[i] be the 15-bit result of a binary polynomial multiplication of the contents of byte element i of VR[VRA] and the contents of byte element i of VR[VRB].

For each integer value i from 0 to 7, do the following.

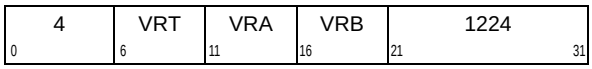
The exclusive-OR of prod[2×i] and prod[2×i+1] is placed in bits 1:15 of halfword element i of VR[VRT]. Bit 0 of halfword element i of VR[VRT] is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Doubleword VX-form

vpmsumd VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 1
   prod[i].bit[0:126] ← 0
   srcA ← VR[VRA].doubleword[i]
   srcB ← VR[VRB].doubleword[i]
   do j = 0 to 63
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 64 to 126
      do k = j-63 to 63
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
VR[VRT] ← 0b0 || (prod[0] ^ prod[1])
```

Let prod[0] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 0 of VR[VRA] and the contents of doubleword element 0 of VR[VRB].

Let prod[1] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 1 of VR[VRA] and the contents of doubleword element 1 of VR[VRB].

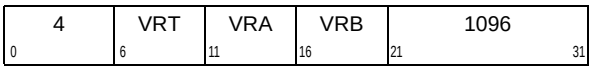
The exclusive-OR of prod[0] and prod[1] is placed in bits 1:127 of VR[VRT]. Bit 0 of VR[VRT] is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Halfword VX-form

vpmsumh VRT,VRA,VRB




```
do i = 0 to 7
   prod.bit[0:30] ← 0
   srcA ← VR[VRA].halfword[i]
   srcB ← VR[VRB].halfword[i]
   do j = 0 to 15
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 16 to 30
      do k = j-15 to 15
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
VR[VRT].word[0] ← 0b0 || (prod[0] ^ prod[1])
VR[VRT].word[1] ← 0b0 || (prod[2] ^ prod[3])
VR[VRT].word[2] ← 0b0 || (prod[4] ^ prod[5])
VR[VRT].word[3] ← 0b0 || (prod[6] ^ prod[7])
```

For each integer value i from 0 to 7, do the following.
Let prod[i] be the 31-bit result of a binary polynomial multiplication of the contents of halfword element i of VR[VRA] and the contents of halfword element i of VR[VRB].

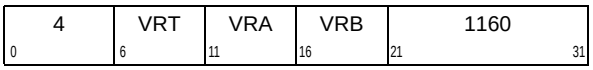
For each integer value i from 0 to 3, do the following.
The exclusive-OR of prod[2×i] and prod[2×i+1] is placed in bits 1:31 of word element i of VR[VRT]. Bit 0 of word element i of VR[VRT] is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Word VX-form

vpmsumw VRT,VRA,VRB




```
do i = 0 to 3
   prod[i].bit[0:62] ← 0
   srcA ← VR[VRA].word[i]
   srcB ← VR[VRB].word[i]
   do j = 0 to 31
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 32 to 62
      do k = j-31 to 31
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
VR[VRT].dword[0] ← 0b0 || (prod[0] ^ prod[1])
VR[VRT].dword[1] ← 0b0 || (prod[2] ^ prod[3])
```

For each integer value i from 0 to 3, do the following.
Let prod[i] be the 63-bit result of a binary polynomial multiplication of the contents of word element i of VR[VRA] and the contents of word element i of VR[VRB].

For each integer value i from 0 to 1, do the following.
The exclusive-OR of prod[2×i] and prod[2×i+1] is placed in bits 1:63 of doubleword element i of VR[VRT]. Bit 0 of doubleword element i of VR[VRT] is set to 0.

Special Registers Altered:

None

5.11.4 Vector Permute and Exclusive-OR Instruction

Vector Permute and Exclusive-OR
VA-form
[Category:Vector.RAID]

vpermxor VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	45
0	6	11	16	21	26 31

```
do i = 0 to 15
   indexA ← VR[VRC].byte[i].bit[0:3]
   indexB ← VR[VRC].byte[i].bit[4:7]
   src1   ← VR[VRA].byte[indexA]
   src2   ← VR[VRB].byte[indexB]
   VSR[VRT].byte[i] ← src1 ^ src2
end
```

For each integer value i from 0 to 15, do the following.

Let indexA be the contents of bits 0:3 of byte element i of VR[VRC].

Let indexB be the contents of bits 4:7 of byte element i of VR[VRC].

The exclusive OR of the contents of byte element indexA of VR[VRA] and the contents of byte element indexB of VR[VRB] is placed into byte element i of VR[VRT].

Special Registers Altered:

None

5.12 Vector Gather Instruction

Vector Gather Bits by Bytes by Doubleword VX-form

vgbbd VRT,VRB

4	VRT	///	VRB	1292
0	6	11	16	21 31

```
do i = 0 to 1
    do j = 0 to 7
        do k = 0 to 7
            b ← VSR[VRB].dword[i].byte[k].bit[j]
            VSR[VRT].dword[i].byte[j].bit[k] ← b
        end
    end
end
```

Let src be the contents of VR[VRB], composed of two doubleword elements numbered 0 and 1.

Let each doubleword element be composed of eight bytes numbered 0 through 7.

An 8-bit × 8-bit bit-matrix transpose is performed on the contents of each doubleword element of VR[VRB] (see Figure 70).

For each integer value i from 0 to 1, do the following,

The contents of bit 0 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 0 of doubleword element i of VR[VRT].

The contents of bit 1 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 1 of doubleword element i of VR[VRT].

The contents of bit 2 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 2 of doubleword element i of VR[VRT].

The contents of bit 3 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 3 of doubleword element i of VR[VRT].

The contents of bit 4 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 4 of doubleword element i of VR[VRT].

The contents of bit 5 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 5 of doubleword element i of VR[VRT].

The contents of bit 6 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 6 of doubleword element i of VR[VRT].

The contents of bit 7 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 7 of doubleword element i of VR[VRT].

Special Registers Altered:

None

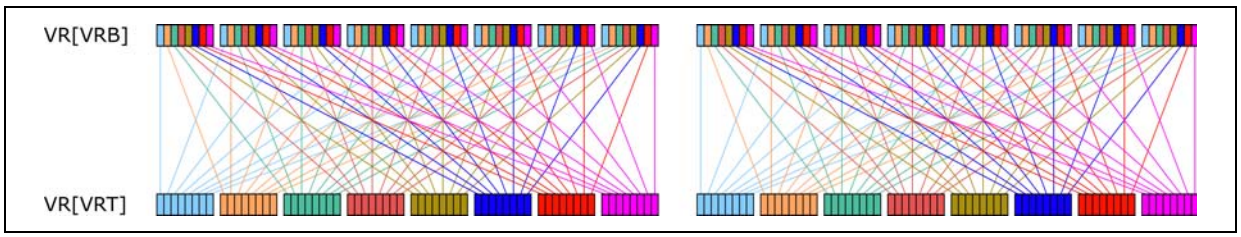


Figure 70. Vector Gather Bits by Bytes by Doubleword

5.13 Vector Count Leading Zeros Instructions

Vector Count Leading Zeros Byte VX-form

vclzb VRT,VRB

4	VRT	///	VRB	1794
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do while n < 8
      if VR[VRB].byte[i].bit[n] = 0b1 then leave
      n ← n + 1
   end
   VSR[VRT].byte[i] ← n
end
```

For each integer value i from 0 to 15, do the following.
A count of the number of consecutive zero bits starting at bit 0 of byte element i of VR[VRB] is placed into byte element i of VR[VRT]. This number ranges from 0 to 8, inclusive.

Special Registers Altered:
None

Vector Count Leading Zeros Halfword VX-form

vclzh VRT,VRB

4	VRT	///	VRB	1858
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n ← 0
   do while n < 16
      if VR[VRB].hword[i].bit[n] = 0b1 then leave
      n ← n + 1
   end
   VSR[VRT].hword[i] ← n
end
```

For each integer value i from 0 to 7, do the following.
A count of the number of consecutive zero bits starting at bit 0 of halfword element i of VR[VRB] is placed into halfword element i of VR[VRT]. This number ranges from 0 to 16, inclusive.

Special Registers Altered:
None

Vector Count Leading Zeros Word VX-form

vclzw VRT,VRB

4	VRT	///	VRB	1922
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n ← 0
   do while n < 32
      if VR[VRB].word[i].bit[n] = 0b1 then leave
      n ← n + 1
   end
   VSR[VRT].word[i] ← n
end
```

For each integer value i from 0 to 3, do the following.
A count of the number of consecutive zero bits starting at bit 0 of word element i of VR[VRB] is placed into word element i of VR[VRT]. This number ranges from 0 to 32, inclusive.

Special Registers Altered:
None

Vector Count Leading Zeros Doubleword

vclzd VRT,VRB

4	VRT	///	VRB	1986
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do while (n<64) & (VR[VRB].dword[i].bit[n]=0b0)
      n ← n + 1
   end
   VSR[VRT].dword[i] ← n
end
```

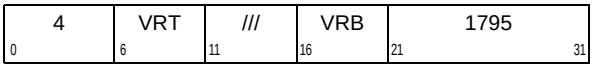
For each integer value i from 0 to 1, do the following.
A count of the number of consecutive zero bits starting at bit 0 of doubleword element i of VR[VRB] is placed into doubleword element i of VR[VRT]. This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None

5.14 Vector Population Count Instructions

Vector Population Count Byte

vpopcntb VRT,VRB



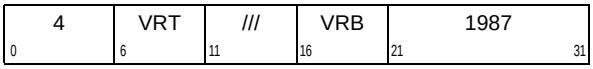

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do j = 0 to 7
      n ← n + VR[VRB].byte[i].bit[j]
   end
   VSR[VRT].byte[i] ← n
end
```

For each integer value i from 0 to 15, do the following.
A count of the number of bits set to 1 in byte element i of VR[VRB] is placed into byte element i of VR[VRT]. This number ranges from 0 to 8, inclusive.

Special Registers Altered:
None

Vector Population Count Doubleword

vpopcntd VRT,VRB



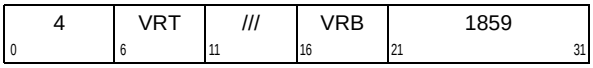

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do j = 0 to 63
      n ← n + VR[VRB].dword[i].bit[j]
   end
   VSR[VRT].dword[i] ← n
end
```

For each integer value i from 0 to 1, do the following.
A count of the number of bits set to 1 in doubleword element i of VR[VRB] is placed into doubleword element i of VR[VRT]. This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None

Vector Population Count Halfword

vpopcnth VRT,VRB



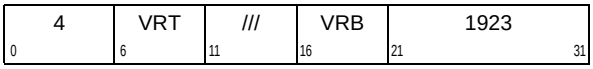

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n ← 0
   do j = 0 to 15
      n ← n + VR[VRB].hword[i].bit[j]
   end
   VSR[VRT].hword[i] ← n
end
```

For each integer value i from 0 to 7, do the following.
A count of the number of bits set to 1 in halfword element i of VR[VRB] is placed into halfword element i of VR[VRT]. This number ranges from 0 to 16, inclusive.

Special Registers Altered:
None

Vector Population Count Word

vpopcntw VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n ← 0
   do j = 0 to 31
      n ← n + VR[VRB].word[i].bit[j]
   end
   VSR[VRT].word[i] ← n
end
```

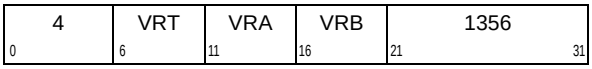
For each integer value i from 0 to 3, do the following.
A count of the number of bits set to 1 in word element i of VR[VRB] is placed into word element i of VR[VRT]. This number ranges from 0 to 32, inclusive.

Special Registers Altered:
None

5.15 Vector Bit Permute Instruction

Vector Bit Permute Quadword VX-form

vbpermq VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   index ← VR[VRB].byte[j]
   if index < 128 then
      perm.bit[i] ← VR[VRA].bit[index]
   else
      perm.bit[i] ← 0
end
VR[VRT].dword[0] ← Chop(EXTZ(perm),64)
VR[VRT].dword[1] ← 0x0000_0000_0000_0000
```

For each integer value i from 0 to 15, do the following.

Let index be the contents of byte element i of VR[VRB].

If index is less than 128, then the contents of bit index of VR[VRA] are placed into bit 48+i of doubleword element i of VR[VRT]. Otherwise, bit 48+i of doubleword element i of VR[VRT] is set to 0.

The contents of bits 0:47 of VR[VRT] are set to 0.
The contents of bits 63:127 of VR[VRT] are set to 0.

Special Registers Altered:

None

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 127 permits the permuted bits to be selected from a 256-bit quantity, using a single index register. For example, assume that the 256-bit quantity Q, from which the permuted bits are to be selected, is in registers v2 (high-order 128 bits of Q) and v3 (low-order 128 bits of Q), that the index values are in register v1, with each byte of v1 containing a value in the range 0:255, and that each byte of register v4 contains the value 128. The following code sequence selects eight permuted bits from Q and places them into the low-order byte of v6.

```
vbpermq   v6,v1,v2    # select from high-order half
                          of Q
vxor      v0,v1,v4    # adjust index values
vbpermq   v5,v0,v3    # select from low-order half
                          of Q
vor       v6,v6,v5    # merge the two selections
```

5.16 Decimal Integer Arithmetic Instructions

The *Decimal Integer Arithmetic* instructions operate on decimal integer values only in signed packed decimal format. Signed packed decimal format consists of 31 4-bit base-10 digits of magnitude and a trailing 4-bit sign code. Operations are performed as sign-magnitude, and produce a decimal result placed in a Vector Register (i.e., ***bcdadd***, ***bcdsub***).

A *valid encoding* of a decimal integer value requires the following properties.

– Each of the 31 4-bit digits of the operand's magnitude (bits 0:123) must be in the range 0-9.
– The sign code (bits 124:127) must be in the range 10-15.

Source operands with sign codes of `0b1010`, `0b1100`, `0b1110`, and `0b1111` are interpreted as positive values.

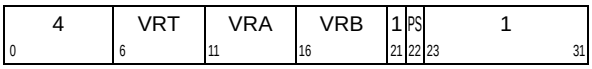
Source operands with sign codes of `0b1011` and `0b1101` are interpreted as negative values.

Positive and zero results are encoded with a either sign code of `0b1100` or `0b1111`, depending on the preferred sign (indicated as an immediate operand).

Negative results are encoded with a sign code of `0b1101`.

Decimal Add Modulo VX-form

bcdadd. VRT,VRA,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Add(VR[VRA],VR[VRB],PS)

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag
```

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

src1 is added to src2.

If the unbounded result is equal to zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101. Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.

The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 0b1001. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VR[VRT].

The sign code is placed in bits 124:127 of VR[VRT].

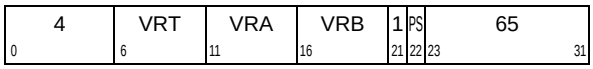
If either src1 or src2 is an *invalid encoding* of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Subtract Modulo VX-form

bcdsub. VRT,VRA,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Subtract(VR[VRA],VR[VRB],PS)

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag
```

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

src1 is subtracted by src2.

If the unbounded result is equal to zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101. Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.

The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 0b1001. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VR[VRT].

The sign code is placed in bits 124:127 of VR[VRT].

If either src1 or src2 is an *invalid encoding* of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

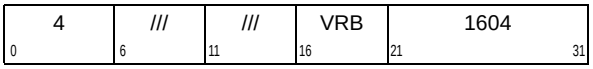
Special Registers Altered:

CR field 6

5.17 Vector Status and Control Register Instructions

Move To Vector Status and Control Register VX-form

mtvscr VRB



VSCR ← (VRB)$_{96:127}$

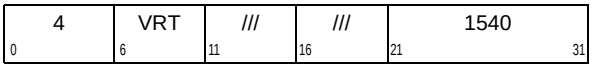
The contents of word element 3 of VRB are placed into the VSCR.

Special Registers Altered:

None

Move From Vector Status and Control Register VX-form

mfvscr VRT



VRT ← $^{96}0$ || (VSCR)

The contents of the VSCR are placed into word element 3 of VRT.

The remaining word elements in VRT are set to 0.

Special Registers Altered:

None

Chapter 6. Decimal Floating-Point [Category: Decimal Floating-Point]

6.1 Decimal Floating-Point (DFP) Facility Overview

This chapter describes the behavior of the decimal floating-point facility, the supported data types, formats, and classes, and the usage of registers. Also included are the execution model, exceptions, and instructions supported by the decimal floating-point facility.

The decimal floating-point (DFP) facility shares the 32 floating-point registers (FPRs) and the Floating-Point Status and Control Register (FPSCR) with the floating-point (BFP) facility. However, the interpretation of data formats in the FPRs, and the meaning of some control and status bits in the FPSCR are different between the BFP and DFP facilities.

The DFP facility also shares the Condition Register (CR) with the fixed-Point facility, the BFP faciltiy, and the vector facility.

The DFP facility supports three DFP data formats: DFP Short (single precision), DFP Long (double precision), and DFP Extended (quad precision). Most operations are performed on DFP Long or DFP Extended format directly. Support for DFP Short is limited to conversion to and from DFP Long. Some DFP instructions operate on other data types, including signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

DFP instructions are provided to perform arithmetic, compare, test, quantum-adjustment, conversion, and format operations on operands held in FPRs or FPR pairs.

- Arithmetic instructions

  These instructions perform addition, subtraction, multiplication, and division operations.

- Compare instructions

  These instructions perform a comparison operation on the numerical value of two DFP operands.

- Test instructions

  These instructions test the data class, the data group, the exponent, or the number of significant digits of a DFP operand.

- Quantum-adjustment instructions

  These instructions convert a DFP number to a result in the form that has the designated exponent, which may be explicitly or implicitly specified.

- Conversion instructions

  These instructions perform conversion between different data formats or data types.

- Format instructions

  These instructions facilitate composing or decomposing a DFP operand.

These instructions are described in Section 6.6 “DFP Instruction Descriptions” on page 288.

The three DFP data formats allow finite numbers to be represented with different precision and ranges. Special codes are also provided to represent +Infinity, -Infinity, Quiet NaN (Not-a-Number), and Signaling NaN. Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. The encoding of NaNs provides a diagnostic information field. This diagnostic field may be used to indicate such things as the source of an uninitialized variable or the reason an invalid result was produced.

The DFP processor recognizes a set of DFP exceptions which are indicated via bits set in the FPSCR. Additionally, the DFP exception actions depend on the setting of the various exception enable bits in the FPSCR.

The following DFP exceptions are detected by the DFP processor. The exception status bits in the FPSCR are indicated in parentheses.

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - $\infty - \infty$ (VXISI)
  - $\infty \div \infty$ (VXIDI)
  - $0 \div 0$ (VXZDZ)

- $\infty \times 0$ (VXIMZ)
- Invalid Compare (VXVC)
- Invalid conversion (VXCVI)

- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XX)

Each DFP exception and each category of Invalid Operation Exception has an exception status bit in the FPSCR. In addition, each of the five DFP exceptions has a corresponding enable bit in the FPSCR. These enable bits enable or disable the invocation of the system floating-point enabled exception error handler, and may affect the setting of some exception status bits in the FPSCR.

The usage of these bits by the DFP facility differs from the usage by the BFP facility. Section 6.5.10 "DFP Exceptions" on page 280 provides a detailed discussion of DFP exceptions, including the effects of the enable bits.

6.2 DFP Register Handling

The following sections describe first how the floating-point registers are utilized by the DFP facility. The subsequent section covers the DFP usage of CR and FPSCR.

6.2.1 DFP Usage of Floating-Point Registers

The DFP facility shares the same 32 64-bit FPRs with the BFP facility. Like the FP instructions, DFP instructions also use 5-bit fields for designating the FPRs to hold the source or target operands.

When data in DFP Short format is held in a FPR, it occupies the rightmost 32 bits of the FPR. The *Load Floating-Point as Integer Word Algebraic* instruction is provided to load the rightmost 32 bits of a FPR with a single-word data from storage. The *Store Floating-Point as Integer Word* instruction is available to store the rightmost 32 bits of a FPR to a storage location.

Data in DFP Long format, 64-bit binary fixed-point values, or 64-bit BCD values is held in a FPR using all 64 bits. Data of 64 bits may be loaded from storage via any of the *Load Floating-Point Double* instructions and stored via any of the *Store Floating-Point Double* instructions.

Data in DFP Extended format or 128-bit BCD values is held in an even-odd FPR pair using all 128 bits. Data of 128 bits must be loaded into the desired even-odd pair of floating-point registers using an appropriate sequence of the *Load Floating-Point Double* instructions and stored using an appropriate sequence of the *Store Floating-Point Double* instructions.

Data used as a source operand by any *Decimal Floating-Point* instruction that was produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a *Floating Round to Single-Precision* instruction, or a binary floating-point single-precision arithmetic instruction is boundedly undefined.

When an even-odd FPR pair is used to hold a 128-bit operand, the even-numbered FPR is used to hold the leftmost doubleword of the operand and the next higher-numbered FPR is used to hold the rightmost doubleword. A DFP instruction designating an odd-numbered FPR for a 128-bit operand is an invalid instruction form.

Programming Note

The *Floating-Point Move* instructions can be used to move operands between FPRs.

The bit definitions for the FPSCR are as follows.

Bit(s)	Description
0:28	Reserved
29:31	***DFP Rounding Control*** (DRN) See Section 6.5.2, "Rounding Mode Specification" on page 277. 000 Round to Nearest, Ties to Even 001 Round toward Zero 010 Round toward +Infinity 011 Round toward -Infinity 100 Round to Nearest, Ties away from 0 101 Round to Nearest, Ties toward 0 110 Round to away from Zero 111 Round to Prepare for Shorter Precision

Programming Note

$FPSCR_{28}$ is reserved for extension of the DRN field, therefore DRN may be set using the ***mtfsfi*** instruction to set the rounding mode.

32	***Floating-Point Exception Summary*** (FX) Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets $FPSCR_{FX}$ to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter $FPSCR_{FX}$ explicitly.
33	***Floating-Point Enabled Exception Summary*** (FEX) This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{FEX}$ explicitly.
34	***Floating-Point Invalid Operation Exception Summary*** (VX) This bit is the OR of all the Invalid Operation

exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{VX}$ explicitly.

35 ***Floating-Point Overflow Exception*** (OX)See Section 6.5.10.3, “Overflow Exception” on page 283.

36 ***Floating-Point Underflow Exception*** (UX) See Section 6.5.10.4, “Underflow Exception” on page 284.

37 ***Floating-Point Zero Divide Exception*** (ZX) See Section 6.5.10.2, “Zero Divide Exception” on page 283.

38 ***Floating-Point Inexact Exception*** (XX) See Section 6.5.10.5, “Inexact Exception” on page 285.

$FPSCR_{XX}$ is a sticky version of $FPSCR_{FI}$ (see below). Thus the following rules completely describe how $FPSCR_{XX}$ is set by a given instruction.

- If the instruction affects $FPSCR_{FI}$, the new value of $FPSCR_{XX}$ is obtained by ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
- If the instruction does not affect $FPSCR_{FI}$, the value of $FPSCR_{XX}$ is unchanged.

39 ***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN) See Section 6.5.10.1, “Invalid Operation Exception” on page 282.

40 ***Floating-Point Invalid Operation Exception (∞ - ∞)*** (VXISI) See Section 6.5.10.1.

41 ***Floating-Point Invalid Operation Exception (∞ ÷ ∞)*** (VXIDI) See Section 6.5.10.1.

~~1~~42 ***Floating-Point Invalid Operation Exception (0÷ 0)*** (VXZDZ) See Section 6.5.10.1.

43 ***Floating-Point Invalid Operation Exception (∞ × 0)*** (VXIMZ) See Section 6.5.10.1.

44 ***Floating-Point Invalid Operation Exception (Invalid Compare)*** (VXVC) See Section 6.5.10.1.

45 ***Floating-Point Fraction Rounded*** (FR) The last *Arithmetic* or *Rounding and Conversion* instruction incremented the fraction during rounding. See Section 6.5.1, “Rounding” on page 276. This bit is not sticky.

46 ***Floating-Point Fraction Inexact*** (FI) The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 6.5.1. This bit is not sticky.

See the definition of $FPSCR_{XX}$, above, regarding the relationship between $FPSCR_{FI}$ and $FPSCR_{XX}$.

47:51 ***Floating-Point Result Flags*** (FPRF) This field is set as described below. For arithmetic, rounding, and conversion instructions, the field is set based on the result placed into the target register, except that if any portion of the result is undefined then the value placed into FPRF is undefined.

47 ***Floating-Point Result Class Descriptor*** (C) Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 71 on page 272.

48:51 ***Floating-Point Condition Code*** (FPCC) Floating-point *Compare* and *DFP Test* instructions set one of the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 71 on page 272. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 ***Floating-Point Less Than or Negative*** (FL or <)

49 ***Floating-Point Greater Than or Positive*** (FG or >)

50 ***Floating-Point Equal or Zero*** (FE or =)

51 ***Floating-Point Unordered or NaN*** (FU or ?)

52 Reserved

53 ***Floating-Point Invalid Operation Exception (Software Request)*** (VXSOFT) This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 6.5.10.1, “Invalid Operation Exception” on page 282.

54 Neither used nor changed by DFP.

Programming Note

Although the architecture does not provide a DFP square root instruction, if software simulates such an instruction, it should set bit 54 whenever the source operand of the square root function is invalid.

55 ***Floating-Point Invalid Operation Exception (Invalid Conversion)*** (VXCVI) See Section 6.5.10.1.

56 ***Floating-Point Invalid Operation Exception Enable*** (VE)
See Section 6.5.10.1.

57 ***Floating-Point Overflow Exception Enable*** (OE)
See Section 6.5.10.3, “Overflow Exception” on page 283.

58 ***Floating-Point Underflow Exception Enable*** (UE)
See Section 6.5.10.4, “Underflow Exception” on page 284.

59 ***Floating-Point Zero Divide Exception Enable*** (ZE)
See Section 6.5.10.2, “Zero Divide Exception” on page 283.

60 ***Floating-Point Inexact Exception Enable*** (XE)
See Section 6.5.10.5, “Inexact Exception” on page 285

61 Reserved (not used by DFP)

62:63 ***Binary Floating-Point Rounding Control*** (RN)
See Section 6.5.1, “Rounding” on page 276.

00 Round to Nearest
01 Round toward Zero
10 Round toward +Infinity
11 Round toward -Infinity

Result Flags					Result Value Class
C	<	>	=	?	
0	0	0	0	1	Signaling NaN (DFP only)
1	0	0	0	1	Quiet NaN
0	1	0	0	1	- Infinity
0	1	0	0	0	- Normal Number
1	1	0	0	0	- Subnormal Number
1	0	0	1	0	- Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Subnormal Number
0	0	1	0	0	+ Normal Number
0	0	1	0	1	+ Infinity

Figure 71. Floating-Point Result Flags

6.3 DFP Support for Non-DFP Data Types

In addition to the DFP data types, the DFP processor provides limited support for the following non-DFP data types: signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

In unsigned binary fixed-point data, all bits are used to express the absolute value of the number. For signed binary fixed-point data, the leftmost bit represents the sign, which is followed by the numeric field. Positive numbers are represented in true binary notation with the sign bit set to zero. When the value is zero, all bits are zeros, including the sign bit. Negative numbers are represented in two’s complement binary notation with a one in the sign-bit position.

For decimal data, each byte contains a pair of four-bit nibbles; each four-bit nibble contains a binary-coded-decimal (BCD) code. There are two kinds of BCD codes: digit code and sign code. For unsigned decimal data, all nibbles contain a digit code (D) as shown in Figure 72

D	D	D	D	. . .	D	D	D	D

Figure 72. Format for Unsigned Decimal Data

For signed decimal data, the rightmost nibble contains a sign code (S) and all other nibbles contain a digit code as shown in Figure 73.

D	D	D	D	. . .	D	D	D	S

Figure 73. Format for Signed Decimal Data

The decimal digits 0-9 have the binary encoding 0000-1001. The preferred plus-sign codes are 1100 and 1111. The preferred minus sign code is 1101. These are the sign codes generated for the results of the *Decode DPD To BCD* instruction. A selection is provided by this instruction to specify which of the two preferred plus sign codes is to be generated. Alternate sign codes are also recognized as valid in the sign position: 1010 and 1110 are alternate sign codes for plus, and 1011 is an alternate sign code for minus. Alternate sign codes are accepted for any source operand, but are not generated as a result by the instruction. When an invalid digit or sign code is detected by the *Encode BCD To DPD* instruction, an invalid-opera-

tion exception occurs. A summary of digit and sign codes are provided in Figure 74.

Binary Code	Recognized As	
	Digit	Sign
0000	0	Invalid
0001	1	Invalid
0010	2	Invalid
0011	3	Invalid
0100	4	Invalid
0101	5	Invalid
0110	6	Invalid
0111	7	Invalid
1000	8	Invalid
1001	9	Invalid
1010	Invalid	Plus
1011	Invalid	Minus
1100	Invalid	Plus (preferred; option 1)
1101	Invalid	Minus (preferred)
1110	Invalid	Plus
1111	Invalid	Plus (preferred; option 2)

Figure 74. Summary of BCD Digit and Sign Codes

6.4 DFP Number Representation

A DFP finite number consists of three components: a sign bit, a signed exponent, and a significand. The signed exponent is a signed binary integer. The *significand* consists of a number of decimal digits, which are to the left of the implied decimal point. The rightmost digit of the significand is called the *units* digit. The numerical value of a DFP finite number is represented as $(-1)^{sign} \times significand \times 10^{exponent}$ and the unit value of this number is (1×10^{exponent}), which is called the *quantum*.

DFP finite numbers are not normalized. This allows leading zeros and trailing zeros to exist in the significand. This unnormalized DFP number representation allows some values to have redundant forms; each form represents the DFP number with a different combination of the significand value and the exponent value. For example, 1000000×10^5 and 10×10^{10} are two different forms of the same numerical value. A *form* of this number representation carries information about both the numerical value and the quantum of a DFP finite number.

The *significant digits* of a DFP finite number are the digits in the significand beginning with the leftmost non-zero digit and ending with the units digit.

6.4.1 DFP Data Format

DFP numbers and NaNs may be represented in FPRs in any of the three data formats: DFP Short, DFP Long, or DFP Extended. The contents of each data format represent encoded information. Special codes are assigned to NaNs and infinities. Different formats support different sizes in both significand and exponent. Arithmetic, compare, test, quantum-adjustment, and format instructions are provided for DFP Long and DFP Extended formats only.

The *sign* is encoded as a one bit binary value. *Significand* is encoded as an unsigned decimal integer in two distinct parts. The leftmost digit (LMD) of the *significand* is encoded as part of the *combination* field*;* the remaining digits of the *significand* are encoded in the *trailing significand* field. The *exponent* is contained in the *combination* field in two parts. However, prior to encoding, the *exponent* is converted to an unsigned binary value called the *biased exponent* by adding a *bias* value which is a constant for each format. The two leftmost bits of the *biased exponent* are encoded with the leftmost digit of the significand in the leftmost bits of the combination field. The rest of the biased exponent occupies the remaining portion of the *combination* field.

6.4.1.1 Fields Within the Data Format

The DFP data representation comprises three fields, as diagrammed below for each of the three formats:

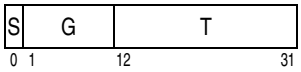


Figure 75. DFP Short format

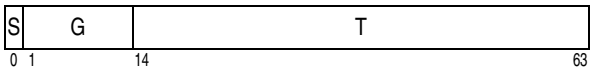


Figure 76. DFP Long format

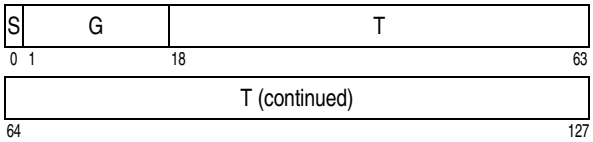


Figure 77. DFP Extended format

The fields are defined as follows:

Sign bit (S)
The sign bit is in bit 0 of each format, and is zero for plus and one for minus.

Combination field (G)
As the name implies, this field provides a combination of the exponent and the left-most digit (LMD) of the significand, for finite numbers, or provides a special code

for denoting the value as either a Not-a-Number or an Infinity.

The first 5 bits of the combination field contain the encoding of NaN or infinity, or the two leftmost bits of the biased exponent and the leftmost digit (LMD) of the significand. The following tables show the encoding:

$G_{0:4}$	Description
11111	NaN
11110	Infinity
All others	Finite Number (see Figure 79)

Figure 78. Encoding of the G field for Special Symbols

LMD	Leftmost 2-bits of biased exponent		
	00	01	10
0	00000	01000	10000
1	00001	01001	10001
2	00010	01010	10010
3	00011	01011	10011
4	00100	01100	10100
5	00101	01101	10101
6	00110	01110	10110
7	00111	01111	10111
8	11000	11010	11100
9	11001	11011	11101

Figure 79. Encoding of bits 0:4 of the G field for Finite Numbers

For DFP finite numbers, the rightmost N-5 bits of the N-bit combination field contain the remaining bits of the *biased exponent*. For NaNs, bit 5 of the combination field is used to distinguish a Quiet NaN from a Signaling NaN; the remaining bits in a source operand are ignored and they are set to zeros in a target operand by most operations. For infinities, the rightmost N-5 bits of the N-bit combination field of a source operand are ignored and they are set to zeros in a target operand by most operations.

Trailing Significand field (T)
For DFP finite numbers, this field contains the remaining *significand* digits. For NaNs, this field may be used to contain diagnostic information. For infinities, contents in this field of a source operand are ignored and they are set to zeros in a target operand by most operations. The trailing significand field is a multiple of 10-bit blocks. The multiple depends on the format. Each 10-bit block is called a declet and represents three decimal digits, using the Densely Packed Decimal (DPD) encoding defined in Appendix B.

6.4.1.2 Summary of DFP Data Formats

The properties of the three DFP formats are summarized in the following table:.

	Format		
	DFP Short	DFP Long	DFP Extended
Widths (bits):			
Format	32	64	128
Sign (S)	1	1	1
Combination (G)	11	13	17
Trailing Significand (T)	20	50	110
Exponent:			
Maximum biased	191	767	12,287
Maximum (X_{max})	90	369	6111
Minimum (X_{min})	-101	-398	-6176
Bias	101	398	6176
Precision (p) (digits)	7	16	34
Magnitude:			
Maximum normal number (N_{max})	$(10^7 - 1) \times 10^{90}$	$(10^{16} - 1) \times 10^{369}$	$(10^{34} - 1) \times 10^{6111}$
Minimum normal number (N_{min})	1×10^{-95}	1×10^{-383}	1×10^{-6143}

	Format		
	DFP Short	**DFP Long**	**DFP Extended**
Minimum subnormal number (D_{min})	1×10^{-101}	1×10^{-398}	1×10^{-6176}

Figure 80. Summary of DFP Formats

6.4.1.3 Preferred DPD Encoding

Execution of DFP instructions decodes source operands from DFP data formats to an internal format for processing, and encodes the operation result before the final result is returned as the target operand.

As part of the decoding process, declets in the trailing significand field of source operands are decoded to their corresponding BCD digit codes using the DPD-to-BCD decoding algorithm. As part of the encoding process, BCD digit codes to be stored into the trailing significand field of the target operand are encoded into declets using the BCD-to-DPD encoding algorithm. Both the decoding and encoding algorithms are defined in Appendix B.

As explained in Appendix B, there are eight 3-digit decimal values that have redundant DPD codes and one preferred DPD code. All redundant DPD codes are recognized in source operands for the associated 3-digit decimal number. DFP operations will always generate the preferred DPD codes for the trailing significand field of the target operand.

6.4.2 Classes of DFP Data

There are six classes of DFP data, which include numerical and nonnumeric entities. The numerical entities include zero, subnormal number, normal number, and infinity data classes. The nonnumeric entities include quiet and signaling NaNs data classes. The value of a DFP finite number, including zero, subnormal number, and normal number, is a quantization of the real number based on the data format. The *Test Data Class* instruction may be used to determine the class of a DFP operand. In general, an operation that returns a DFP result sets the $FPSCR_{FPRF}$ field to indicate the data class of the result.

The following tables show the value ranges for finite-number data classes, and the codes for NaNs and infinities.

Data Class	Sign	Magnitude
Zero	±	0*
Subnormal	±	$D_{min} \leq \|X\| < N_{min}$
Normal	±	$N_{min} \leq \|Y\| \leq N_{max}$
* The significand is zero and the exponent is any representable value		

Figure 81. Value Ranges for Finite Number Data Classes

Data Class	S	G	T
+Infinity	0	11110xxx . . . xxx	xxx . . . xxx
–Infinity	1	11110xxx . . . xxx	xxx . . . xxx
Quiet NaN	x	111110xx . . . xxx	xxx . . . xxx
Signaling NaN	x	111111xx . . . xxx	xxx . . . xxx
x Don't care			

Figure 82. Encoding of NaN and Infinity Data Classes

Zeros
Zeros have a zero significand and any representable value in the exponent. A +0 is distinct from -0, and zeros with different exponents are distinct, except that comparison treats them as equal.

Subnormal Numbers
Subnormal numbers have values that are smaller than N_{min} and greater than zero in magnitude.

Normal Numbers
Normal numbers are nonzero finite numbers whose magnitude is between N_{min} and N_{max} inclusively.

Infinities
Infinities are represented by 0b11110 in the leftmost 5 bits of the combination field. When an operation is defined to generate an infinity as the result, a default infinity is sometimes supplied. A default infinity has all remaining bits in the combination field and trailing significand field set to zeros.

When infinities are used as source operands, only the leftmost 5 bits of the combination field are interpreted (i.e., 0b11110 indicates the value is an infinity). The trailing significand field of infinities is usually ignored. For generated infinities, the leftmost 5 bits of the combination field are set to 0b11110 and all remaining combination bits are set to zero.

Infinities can participate in most arithmetic operations and give a consistent result. In comparisons, any +Infinity compares greater than any finite number, and any -Infinity compares less than any finite number. All +Infinity are compared equal and all -Infinity are compared equal.

Signaling and Quiet NaNs
There are two types of Not-a-Numbers (NaNs), Signaling (SNaN) and Quiet (QNaN).

0b111110 in the leftmost 6 bits of the combination field indicates a Quiet NaN, whereas 0b111111 indicates a Signaling NaN.

A special QNaN is sometimes supplied as the *default QNaN* for a disabled invalid-operation exception; it has a plus sign, the leftmost 6 bits of the combination field set to 0b111110 and remaining bits in the combination field and the trailing significand field set to zero.

Normally, source QNaNs are *propagated* during operations so that they will remain visible at the end. When a QNaN is propagated, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost N-6 bits of the combination field set to zero, where N is the width of the combination field for the format.

A source SNaN generally causes an invalid-operation exception. If the exception is disabled, the SNaN is converted to the corresponding QNaN and propagated. The primary encoding difference between an SNaN and a QNaN is that bit 5 of an SNaN is 1 and bit 5 of a QNaN is 0. When an SNaN is propagated as a QNaN, bit 5 is set to 0, and, just as with QNaN proagation, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost N-6 bits of the combination field set to zero, where N is the width of the combination field for the format. For some format-conversion instructions, a source SNaN does not cause an invalid-operation exception, and an SNaN is returned as the target operand.

For instructions with two source NaNs and a NaN is to be propagated as the result, do the following.

- If there is a QNaN in FRA and an SNaN in FRB, the SNaN in FRB is propagated.
- Otherwise, propagate the NaN is FRA.

6.5 DFP Execution Model

DFP operations are performed as if they first produce an intermediate result correct to infinite precision and with unbounded range. The intermediate result is then rounded to the destination's precision according to one of the eight DFP rounding modes. If the rounded result has only one form, it is delivered as the final result; if the rounded result has redundant forms, then an *ideal exponent* is used to select the form of the final result. The ideal exponent determines the form, not the value, of the final result. (See Section 6.5.3 "Formation of Final Result" on page 278.)

6.5.1 Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit the destination's precision. The destination's precision of an operation defines the set of permissible resultant values. For

most operations, the destination's precision is the target-format precision and the permissible resultant values are those values representable in the target format. For some special operations, the destination precision is constrained by both the target format and some additional restrictions, and the permissible resultant values are a subset of the values representable in the target format.

Rounding sets FPSCR bits FR and FI. When an inexact exception occurs, FI is set to one; otherwise, FI is set to zero. When an inexact exception occurs and if the rounded result is greater in magnitude than the intermediate result, then FR is set to one; otherwise, FR is set to zero. The exception is the *Round to FP Integer Without Inexact* instruction, which always sets FR and FI to zero. Rounding may cause an overflow exception or underflow exception; it may also cause an inexact exception.

Refer to Figure 83 below for rounding. Let Z be the intermediate result of a DFP operation. Z may or may not fit in the destination's precision. If Z is exactly one of the permissible representable resultant values, then the final result in all rounding modes is Z. Otherwise, either Z1 or Z2 is chosen to approximate the result, where Z1 and Z2 are the next larger and smaller permissible resultant values, respectively.

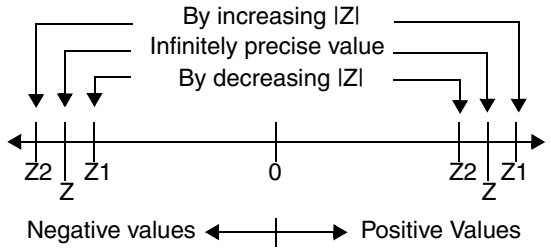


Figure 83. Rounding

Round to Nearest, Ties to Even
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one whose units digit would have been even in the form with the largest common quantum of the two permissible resultant values. However, an infinitely precise result with magnitude at least (N_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round toward 0
Choose the smaller in magnitude (Z1 or Z2).

Round toward +∞
Choose Z1.

Round toward -∞
Choose Z2.

Round to Nearest, Ties away from 0
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the larger in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude at least (N_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round to Nearest, Ties toward 0
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the smaller in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude greater than (N_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round away from 0
Choose the larger in magnitude (Z1 or Z2).

Round to prepare for shorter precision
Choose the smaller in magnitude (Z1 or Z2). If the selected value is inexact and the units digit of the selected value is either 0 or 5, then the digit is incremented by one and the incremented result is delivered. In all other cases, the selected value is delivered. When a value has redundant forms, the units digit is determined by using the form that has the smallest exponent.

6.5.2 Rounding Mode Specification

Unless otherwise specified in the instruction definition, the rounding mode used by an operation is specified in the DFP rounding control (DRN) field of the FPSCR. The eight DFP rounding modes are encoded in the DRN field as specified in the table below.

DRN	Rounding Mode
000	Round to Nearest, Ties to Even
001	Round toward 0
010	Round toward +Infinity
011	Round toward -Infinity
100	Round to Nearest, Ties away from 0
101	Round to Nearest, Ties toward 0
110	Round away from 0
111	Round to Prepare for Shorter Precision

Figure 84. Encoding of DFP Rounding-Mode Control (DRN)

For the quantum-adjustment, a 2-bit immediate field, called RMC (*Rounding Mode Control*), in the instruction specifies the rounding mode used. The RMC field may contain a primary encoding or a secondary encoding. For *Quantize*, *Quantize Immediate*, and *Reround*, the RMC field contains the primary encoding. For *Round to FP Integer* the field contains either encoding, depending on the setting of a RMC-encoding-selection

bit. The following tables define the primary encoding and the secondary encoding.

Primary RMC	Rounding Mode
00	Round to nearest, ties to even
01	Round toward 0
10	Round to nearest, ties away from 0
11	Round according to $FPSCR_{DRN}$

Figure 85. Primary Encoding of Rounding-Mode Control

Secondary RMC	Rounding Mode
00	Round to + ∞
01	Round to - ∞
10	Round away from 0
11	Round to nearest, ties toward 0

Figure 86. Secondary Encoding of Rounding-Mode Control

6.5.3 Formation of Final Result

An ideal exponent is defined for each DFP instruction that returns a DFP data operand.

6.5.3.1 Use of Ideal Exponent

For all DFP operations,

- if the rounded intermediate result has only one form, then that form is delivered as the final result.
- if the rounded intermediate result has redundant. forms and is exact, then the form with the exponent closest to the ideal exponent is delivered.
- if the rounded intermediate result has redundant forms and is inexact, then the form with the smallest exponent is delivered.

The following table specifies the ideal exponent for each instruction.

Operations	Ideal Exponent
Add	min(E(FRA), E(FRB))
Subtract	min(E(FRA), E(FRB))
Multiply	E(FRA) + E(FRB)
Divide	E(FRA) - E(FRB)
Quantize-Immediate	See Instruction Description
Quantize	E(FRA)
Reround	See Instruction Description
Round to FP Integer	max(0, E(FRA))
Convert to DFP Long	E(FRA)
Convert to DFP Extended	E(FRA)
Round to DFP Short	E(FRA)
Round to DFP Long	E(FRA)
Convert from Fixed	0
Encode BCD to DPD	0
Insert Biased Exponent	E(FRA)
Notes: E(x) - exponent of the DFP operand in register x.	

Figure 87. Summary of Ideal Exponents

6.5.4 Arithmetic Operations

Four arithmetic operations are provided: Add, Subtract, Multiply, and Divide.

6.5.4.1 Sign of Arithmetic Result

The following rules govern the sign of an arithmetic operation when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the source operand having the larger absolute value. If both source operands have the same sign, the sign of the result of an add operation is the same as the sign of the source operands. When the sum of two operands with opposite signs is exactly zero, the sign of the result is positive in all rounding modes except Round toward -∞, in which case the sign is negative.
- The sign of the result of the subtract operation x - y is the same as the sign of the result of the add operation x + (-y).
- The sign of the result of a multiply or divide operation is the exclusive-OR of the signs of the source operands.

6.5.5 Compare Operations

Two sets of instructions are provided for comparing numerical values: *Compare Ordered* and *Compare Unordered*. In the absence of NaNs, these instructions work the same. These instructions work differently when either of the followings is true:

1. At least one source operand of the instruction is an SNaN and the invalid-operation exception is disabled.
2. When there is no SNaN in any source operand, at least one source operand of the instruction is a QNaN

In case 1, *Compare Unordered* recognizes an invalid-operation exception and sets the $FPSCR_{VXSNAN}$ flag, but *Compare Ordered* recognizes the exception and sets both the $FPSCR_{VXSNAN}$ and $FPSCR_{VXVC}$ flags. In case 2, *Compare Unordered* does not recognize an exception, but *Compare Ordered* recognizes an invalid-operation exception and sets the $FPSCR_{VXVC}$ flag.

For finite numbers, comparisons are performed on values, that is, all redundant forms of a DFP number are treated equal.

Comparisons are always exact and cannot cause an inexact exception.

Comparison ignores the sign of zero, that is, +0 equals -0.

Infinities with like sign compare equal, that is, $+\infty$ equals $+\infty$, and $-\infty$ equals $-\infty$.

A NaN compares as unordered with any other operand, whether a finite number, an infinity, or another NaN, including itself.

Execution of a compare instruction always completes, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

6.5.6 Test Operations

Four kinds of test operations are provided: *Test Data Class*, *Test Data Group*, *Test Exponent*, and *Test Significance*.

The *Test Data Class* instruction examines the contents of a source operand and determines if the operand is one of the specified data classes. The test result and the sign of the source operand are indicated in the $FPSCR_{FPCC}$ field and CR field BF.

The *Test Data Group* instruction examines the contents of a source operand and determines if the operand is one of the specified data groups. The test result and the sign of the source operand are indicated in the $FPSCR_{FPCC}$ field and CR field BF.

The *Test Exponent* instruction compares the exponent of the two source operands. The test operation ignores the sign and significand of operands. Infinities compare equal, and NaNs compare equal. The test result is indicated in the $FPSCR_{FPCC}$ field and CR field BF.

The *Test Significance* instruction compares the number of significant digits of one source operand with the referenced number of significant digits in another source operand. The test result is indicated in the $FPSCR_{FPCC}$ field and CR field BF.

Execution of a test instruction does not cause any DFP exception.

6.5.7 Quantum Adjustment Operations

Four kinds of quantum-adjustment operations are provided: *Quantize*, *Quantize Immediate*, *Reround*, and *Round To FP Integer*. Each of them has an immediate field which specifies whether the rounding mode in FPSCR or a different one is to be used.

The *Quantize* instruction is used to adjust a DFP number to the form that has the specified target exponent. The *Quantize Immediate* instruction is similar to the *Quantize* instruction, except that the target exponent is specified in a 5-bit immediate field as a signed binary integer and has a limited range.

The *Reround* instruction is used to simulate a DFP operation of a precision other than that of DFP Long or DFP Extended. For the *Reround* instruction to produce a result which accurately reflects that which would have resulted from a DFP operation of the desired precision *d* in the range {1: 33} inclusively, the following conditions must be met:

- The precision of the preceding DFP operation must be at least one digit larger than *d*.
- The rounding mode used by the preceding DFP operation must be *round-to-prepare-for-shorter-precision.*

The *Round To FP Integer* instruction is used to round a DFP number to an integer value of the same format. The target exponent is implicitly specified, and is greater than or equal to zero.

6.5.8 Conversion Operations

There are two kinds of conversion operations: data-format conversion and data-type conversion.

6.5.8.1 Data-Format Conversion

The instructions *Convert To DFP Long* and *Convert To DFP Extended* convert DFP operands to wider formats; the instructions *Round To DFP Short* and *Round To DFP Long* convert DFP operands to narrower formats.

When converting a finite number to a wider format, the result is exact. When converting a finite number to a

narrower format, the source operand is rounded to the target-format precision, which is specified by the instruction, not by the target register size.

When converting a finite number, the ideal exponent of the result is the source exponent.

Conversion of an infinity or NaN to a different format does not preserve the source combination field. Let N be the width of the target format's combination field.

- When the result is an infinity or a QNaN, the contents of the rightmost N-5 bits of the N-bit target combination field are set to zero.
- When the result is an SNaN, bit 5 of the target format's combination field is set to one and the rightmost N-6 bits of the N-bit target combination field are set to zero.

When converting a NaN to a wider format or when converting an infinity from DFP Short to DFP Long, digits in the source trailing significand field are reencoded using the preferred DPD codes with sufficient zeros appended on the left to form the target trailing significand field. When converting a NaN to a narrower format or when converting an infinity from DFP Long to DFP Short, the appropriate number of leftmost digits of the source trailing significand field are removed and the remaining digits of the field are reencoded using the preferred DPD codes to form the target trailing significand field.

When converting an infinity between DFP Long and DFP Extended, a default infinity with the same sign is produced.

When converting an SNaN between DFP Short and DFP Long, it is converted to an SNaN without causing an invalid-operation exception. When converting an SNaN between DFP Long and DFP Extended, the invalid-operation exception occurs; if the invalid-operation exception is disabled, the result is converted to the corresponding QNaN.

6.5.8.2 Data-Type Conversion

The instructions *Convert From Fixed* and *Convert To Fixed* are provided to convert a number between the DFP data type and the signed 64-bit binary-integer data type.

Conversion of a signed 64-bit binary integer to a DFP Extended number is always exact.

Conversion of a DFP number to a signed 64-bit binary integer results in an invalid-operation exception when the converted value does not fit into the target format, or when the source operand is an infinity or NaN. When the exception is disabled, the most positive integer is returned if the source operand is a positive number or $+\infty$, and the most negative integer is returned if the source operand is a negative number, $-\infty$, or NaN.

6.5.9 Format Operations

The format instructions are provided to facilitate composing or decomposing a DFP number, and consist of *Encode BCD To DPD*, *Decode DPD To BCD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*. A source operand of SNaN does not cause an invalid-operation exception, and an SNaN may be produced as the target operand.

6.5.10 DFP Exceptions

This architecture defines the following DFP exceptions:

- Invalid Operation Exception
  - SNaN
  - $\infty - \infty$
  - $\infty \div \infty$
  - $0 \div 0$
  - $\infty \times 0$
  - Invalid Compare
  - Invalid Conversion
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of a DFP instruction.

Each DFP exception, and each category of the Invalid Operation Exception, has an exception status bit in the FPSCR. In addition, each DFP exception has a corresponding enable bit in the FPSCR. The exception status bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see the discussion of FE0 and FE1 below), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its source operands, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions

- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Conversion) for *Convert To Fixed* instructions.

When an exception occurs the instruction execution may be completed or partially completed, depending on the exception and the operation.

For all instructions, except for the Compare and Test instructions, the following exceptions cause the instruction execution to be partially completed. That is, setting of CR field 1(when Rc=1) and exception status flags is performed, but no result is stored into the target FPR or FPR pair. For Compare and Test instructions, instruction execution is always completed, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exceptions, instruction execution is completed, a result, if specified by the instruction, is generated and stored into the target FPR or FPR pair, and appropriate status flags are set. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exceptions that deliver a result in target FPR are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the DFP exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, a FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case: the expectation is that the exception will be detected by software, which will revise the result. A FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to zero and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if DFP exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to one and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1; the *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The location of these bits and the requirements for altering them are described in Book III, *Power AS Operating Environment Architecture*. (The system floating-point enabled exception error handler is never invoked because of a disabled DFP exception.) The effects of the four possible settings of these bits are as follows.

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** DFP exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.
1	0	**Imprecise Recoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.

FE0	FE1	Description
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In all cases, the question of whether a DFP result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. (Recall that, for the two Imprecise modes, the instruction at which the system floating-point enabled exception error handler is invoked need not be the instruction that caused the exception.) The instruction at which the system floating-point enabled exception error handler is invoked has not been executed unless it is the excepting instruction, in which case it has been executed if the exception is not among those listed on page 280 as suppressed.

Programming Note

In the ignore and both imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to zero.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to one for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to one.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

6.5.10.1 Invalid Operation Exception

Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified DFP operation. The invalid DFP operations are:

- Any DFP operation on a signaling NaN (SNaN), except for *Test*, *Round To DFP Short*, *Convert To DFP Long*, *Decode DPD To BCD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*
- For add or subtract operations, magnitude subtraction of infinities $(+\infty) + (-\infty)$
- Division of infinity by infinity $(\infty \div \infty)$
- Division of zero by zero $(0 \div 0)$
- Multiplication of infinity by zero $(\infty \times 0)$
- Ordered comparison involving a NaN (Invalid Compare)
- The *Quantize* operation detects that the significand associated with the specified target exponent would have more significant digits than the target-format precision
- For the *Quantize* operation, when one source operand specifies an infinity and the other specifies a finite number
- The *Reround* operation detects that the target exponent associated with the specified target significance would be greater than X_{max}
- The *Encode BCD To DPD* operation detects an invalid BCD digit or sign code
- The *Convert To Fixed* operation involving a number too large in magnitude to be represented in the target format, or involving a NaN.

Programming Note

In addition, an Invalid Operation Exception occurs if software explicitly requests this by executing an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction that sets $FPSCR_{VXSOFT}$ to 1 (Software Request). The purpose of $FPSCR_{VXSOFT}$ is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a DFP instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.

Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled ($FPSCR_{VE}$=1) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty - \infty$)
   - $FPSCR_{VXIDI}$ (if $\infty \div \infty$)
   - $FPSCR_{VXZDZ}$ (if $0 \div 0$)
   - $FPSCR_{VXIMZ}$ (if $\infty \times 0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXCVI}$ (if invalid conversion)
2. If the operation is an arithmetic, quantum-adjustment, conversion, or format,
   the target FPR is unchanged,
   $FPSCR_{FR\ FI}$ are set to zero, and
   $FPSCR_{FPRF}$ is unchanged.
3. If the operation is a compare,
   $FPSCR_{FR\ FI\ C}$ are unchanged, and
   $FPSCR_{FPCC}$ is set to reflect unordered.

When Invalid Operation Exception is disabled ($FPSCR_{VE}$=0) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty - \infty$)
   - $FPSCR_{VXIDI}$ (if $\infty \div \infty$)
   - $FPSCR_{VXZDZ}$ (if $0 \div 0$)
   - $FPSCR_{VXIMZ}$ (if $\infty \times 0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXCVI}$ (if invalid conversion)
2. If the operation is an arithmetic, quantum-adjustment, *Round to DFP Long*, *Convert to DFP Extended*, or format
   the target FPR is set to a Quiet NaN
   $FPSCR_{FR\ FI}$ are set to zero
   $FPSCR_{FPRF}$ is set to indicate the class of the result (Quiet NaN)
3. If the operation is a *Convert To Fixed*
   the target FPR is set as follows:
   FRT is set to the most positive 64-bit binary integer if the operand in FRB is a positive or $+\infty$, and to the most negative 64-bit binary integer if the operand in FRB is a negative number, $-\infty$, or NaN.
   $FPSCR_{FR\ FI}$ are set to zero
   $FPSCR_{FPRF}$ is unchanged
4. If the operation is a compare,
   $FPSCR_{FR\ FI\ C}$ are unchanged
   $FPSCR_{FPCC}$ is set to reflect unordered

6.5.10.2 Zero Divide Exception

Definition

A Zero Divide Exception occurs when a Divide instruction is executed with a zero divisor value and a finite nonzero dividend value.

Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled ($FPSCR_{ZE}$=1) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is unchanged
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is unchanged

When Zero Divide Exception is disabled ($FPSCR_{ZE}$=0) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is set to $\pm\infty$, where the sign is determined by the XOR of the signs of the operands
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result ($\pm\infty$)

6.5.10.3 Overflow Exception

Definition

An overflow exception occurs whenever the target format's largest finite number is exceeded in magnitude by what would have been the rounded result if the exponent range were unbounded.

Action

Except for *Reround*, the following describes the handling of the IEEE overflow exception condition. The *Reround* operation does not recognize an overflow exception condition.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled ($FPSCR_{OE}$=1) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. The infinitely precise result is divided by 10^{α}. That is, the exponent adjustment α is subtracted from the exponent. This is called the *wrapped result*. The exponent adjustment for all operations, except for *Round To DFP Short* and *Round To DFP Long*, is 576 for DFP Long and 9216 for DFP Extended. For *Round To DFP Short* and *Round To DFP Long*, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the *wrapped rounded result*.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of subtracting the exponent adjustment from the ideal exponent.
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled ($FPSCR_{OE}$=0) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
3. The result is determined by the rounding mode and the sign of the intermediate result as follows.

Rounding Mode	Sign of intermediate result	
	Plus	**Minus**
Round to Nearest, Ties to Even	$+\infty$	$-\infty$
Round toward 0	$+N_{max}$	$-N_{max}$
Round toward $+\infty$	$+\infty$	$-N_{max}$
Round toward $-\infty$	$+N_{max}$	$-\infty$
Round to Nearest, Ties away from 0	$+\infty$	$-\infty$
Round to Nearest, Ties toward 0	$+\infty$	$-\infty$
Round away from 0	$+\infty$	$-\infty$
Round to prepare for shorter precision	$+N_{max}$	$-N_{max}$

Figure 88. Overflow Results When Exception Is Disabled

4. The result is placed into the target FPR
5. $FPSCR_{FR}$ is set to one if the returned result is $\pm\infty$, and is set to zero if the returned result is $\pm N_{max}$
6. $FPSCR_{FI}$ is set to one
7. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result ($\pm\infty$ or ± Normal number)

6.5.10.4 Underflow Exception

Definition

Except for *Reround*, the following describes the handling of the IEEE underflow exception condition. The *Reround* operation does not recognize an underflow exception condition.

The Underflow Exception is defined differently for the enabled and disabled states. However, a tininess condition is recognized in both states when a result computed as though both the precision and exponent range were unbounded would be nonzero and less than the target format's smallest normal number, N_{min}, in magnitude.

Unless otherwise defined in the instruction description, an underflow exception occurs as follows:

- Enabled:
  When the tininess condition is recognized.
- Disabled:
  When the tininess condition is recognized and when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}$=1) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The infinitely precise result is multiplied by 10^{α}. That is, the exponent adjustment α is added to the exponent. This is called the *wrapped result*. The exponent adjustment for all operations, except for *Round To DFP Short* and *Round To DFP Long*, is 576 for DFP Long and 9216 for DFP Extended. For *Round To DFP Short* and *Round To DFP Long*, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the *wrapped rounded result*.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the

wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of adding the exponent adjustment to the ideal exponent.

5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal number)

When Underflow Exception is disabled ($FPSCR_{UE}$=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The infinitely precise result is rounded to the target-format precision.
3. The rounded result is returned. If this result has redundant forms, the result of the form that is closest to the ideal exponent is returned.
4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal number, ± Subnormal Number, or ± Zero)

6.5.10.5 Inexact Exception

Definition

Except for *Round to FP Integer Without Inexact*, the following describes the handling of the IEEE inexact exception condition. The *Round to FP Integer Without Inexact* does not recognize an inexact exception condition.

An Inexact Exception occurs when either of two conditions occur during rounding:

1. The delivered result differs from what would have been computed were both the precision and exponent range unbounded.
2. The rounded result overflows and Overflow Exception is disabled.

Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

6.5.11 Summary of Normal Rounding And Range Actions

Figure 89 and Figure 90 summarize rounding and range actions, with the following exceptions:

- The *Reround* operation recognizes neither an underflow nor an overflow exception.
- The *Round to FP Integer Without Inexact* operation does not recognize the inexact operation exception.

Range of v	Case	Result (r) when Rounding Mode Is							
		RNE	RNTZ	RNAZ	RAFZ	RTMI	RFSP	RTPI	RTZ
v < -Nmax, q < -Nmax	Overflow	$-\infty^1$	$-\infty^1$	$-\infty^1$	$-\infty^1$	$-\infty^1$	-Nmax	-Nmax	-Nmax
v < -Nmax, q = -Nmax	Normal	-Nmax	-Nmax	-Nmax	—	—	-Nmax	-Nmax	-Nmax
-Nmax ≤ v ≤ -Nmin	Normal	b	b	b	b	b	b	b	b
-Nmin < v ≤ -Dmin	Tiny	b*	b*	b*	b*	b*	b*	b	b
-Dmin < v < -Dmin/2	Tiny	-Dmin	-Dmin	-Dmin	-Dmin	-Dmin	-Dmin	-0	-0
v = -Dmin/2	Tiny	-0	-0	-Dmin	-Dmin	-Dmin	-Dmin	-0	-0
-Dmin/2 < v < 0	Tiny	-0	-0	-0	-Dmin	-Dmin	-Dmin	-0	-0
v = 0	EZD	+0	+0	+0	+0	-0	+0	+0	+0
0 < v < +Dmin/2	Tiny	+0	+0	+0	+Dmin	+0	+Dmin	+Dmin	+0
v = +Dmin/2	Tiny	+0	+0	+Dmin	+Dmin	+0	+Dmin	+Dmin	+0
+Dmin/2 < v < +Dmin	Tiny	+Dmin	+Dmin	+Dmin	+Dmin	+0	+Dmin	+Dmin	+0
+Dmin ≤ v < +Nmin	Tiny	b*	b*	b*	b*	b	b*	b*	b
+Nmin ≤ v ≤ +Nmax	Normal	b	b	b	b	b	b	b	b
+Nmax < v, q = +Nmax	Normal	+Nmax	+Nmax	+Nmax	—	+Nmax	+Nmax	—	+Nmax
+Nmax < v, q > +Nmax	Overflow	$+\infty^1$	$+\infty^1$	$+\infty^1$	$+\infty^1$	+Nmax	+Nmax	$+\infty^1$	+Nmax

Explanation:

—	This situation cannot occur.
1	The normal result r is considered to have been incremented.
*	The rounded value, in the extreme case, may be Nmin. In this case, the exception conditions are underflow, inexact, and incremented.
b	The value derived when the precise result v is rounded to the destination's precision, including both bounded precision and bounded exponent range.
q	The value derived when the precise result v is rounded to the destination's precision, but assuming an unbounded exponent range.
r	This is the returned value when neither overflow nor underflow is enabled.
v	Precise result before rounding, assuming unbounded precision and an unbounded exponent range. For data-format conversion operations, v is the source value.
Dmin	Smallest (in magnitude) representable subnormal number in the target format.
EZD	The result r of the exact-zero-difference case applies only to ADD and SUBTRACT with both source operands having opposite signs. (For ADD and SUBTRACT, when both source operands have the same sign, the sign of the zero result is the same sign as the sign of the source operands.)
Nmax	Largest (in magnitude) representable finite number in the target format.
Nmin	Smallest (in magnitude) representable normalized number in the target format.
RAFZ	Round away from 0.
RFSP	Round to Prepare for Shorter Precision.
RNAZ	Round to Nearest, Ties away from 0.
RNE	Round to Nearest, Ties to even.
RNTZ	Round to Nearest, Ties toward 0.
RTPI	Round toward +∞.
RTMI	Round toward -∞.
RTZ	Round toward 0.

Figure 89. Rounding and Range Actions (Part 1)

Case	Is r inexact (r≠v)	OE=1	UE=1	XE=1	Is r Incre-mented (|r|>|v|)	Is q inexact (q≠v)	Is q Incre-mented (|q|>|v|)	Returned Results and Status Setting*
Overflow	Yes[1]	No	—	No	No	—	—	T(r), OX← 1, FI← 1, FR← 0, XX ← 1
Overflow	Yes[1]	No	—	No	Yes	—	—	T(r), OX← 1, FI← 1, FR← 1, XX ← 1
Overflow	Yes[1]	No	—	Yes	No	—	—	T(r), OX← 1, FI← 1, FR← 0, XX ← 1, TX
Overflow	Yes[1]	No	—	Yes	Yes	—	—	T(r), OX← 1, FI← 1, FR← 1, XX ← 1, TX
Overflow	Yes[1]	Yes	—	—	—	No	No[1]	Tw(q÷β), OX← 1, FI← 0, FR← 0, TO
Overflow	Yes[1]	Yes	—	—	—	Yes	No	Tw(q÷β), OX← 1, FI← 1, FR← 0, XX← 1,TO
Overflow	Yes[1]	Yes	—	—	—	Yes	Yes	Tw(q÷β), OX← 1, FI← 1, FR← 1, XX← 1,TO
Normal	No	—	—	—	—	—	—	T(r), FI← 0, FR← 0
Normal	Yes	—	—	No	No	—	—	T(r), FI← 1, FR← 0, XX ← 1
Normal	Yes	—	—	No	Yes	—	—	T(r), FI← 1, FR← 1, XX ← 1
Normal	Yes	—	—	Yes	No	—	—	T(r), FI← 1, FR← 0, XX ← 1, TX
Normal	Yes	—	—	Yes	Yes	—	—	T(r), FI← 1, FR← 1, XX ← 1, TX
Tiny	No	—	No	—	—	—	—	T(r), FI← 0, FR← 0
Tiny	No	—	Yes	—	—	No[1]	No[1]	Tw(q•β), UX← 1, FI← 0, FR← 0, TU
Tiny	Yes	—	No	No	No	—	—	T(r), UX← 1, FI← 1, FR← 0, XX ← 1
Tiny	Yes	—	No	No	Yes	—	—	T(r), UX← 1, FI← 1, FR← 1, XX ← 1
Tiny	Yes	—	No	Yes	No	—	—	T(r), UX← 1, FI← 1, FR← 0, XX ← 1, TX
Tiny	Yes	—	No	Yes	Yes	—	—	T(r), UX← 1, FI← 1, FR← 1, XX ← 1, TX
Tiny	Yes	—	Yes	—	—	No	No[1]	Tw(q•β), UX← 1, FI← 0, FR← 0, TU
Tiny	Yes	—	Yes	—	—	Yes	No	Tw(q•β), UX← 1, FI← 1, FR← 0, XX ← 1,TU
Tiny	Yes	—	Yes	—	—	Yes	Yes	Tw(q•β), UX← 1, FI← 1, FR← 1, XX ← 1,TU

Explanation:

—	The results do not depend on this condition.
1	This condition is true by virtue of the state of some condition to the left of this column.
*	Rounding sets only the FI and FR status flags. Setting of the OX, XX, or UX flag is part of the exception actions. They are listed here for reference.
β	Wrap adjust, which depends on the type of operation and operand format. For all operations except *Round to DFP Short* and *Round to DFP Long*, the wrap adjust depends on the target format: $\beta = 10^{\alpha}$, where α is 576 for DFP Long, and 9216 for DFP Extended. For *Round to DFP Short* and *Round to DFP Long*, the wrap adjust depends on the source format: $\beta = 10^{\kappa}$ where κ is 192 for DFP Long and 3072 for DFP Extended.
q	The value derived when the precise result v is rounded to destination's precision, but assuming an unbounded exponent range.
r	The result as defined in Part 1 of this figure.
v	Precise result before rounding, assuming unbounded precision and unbounded exponent range.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$. This status flag is non-sticky.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{oX}$.
TO	The system floating-point enabled exception error handler is invoked for the overflow exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TU	The system floating-point enabled exception error handler is invoked for the underflow exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TX	The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
T(x)	The value x is placed at the target operand location.
Tw(x)	The wrapped rounded result x is placed at the target operand location. For all operations except data format conversions, the wrapped rounded result is in the same format and length as normal results at the target location. For data format conversions, the wrapped rounded result is in the same format and length as the source, but rounded to the target-format precision.
UX	Floating-Point-Underflow-Exception status flag, $FPSCR_{UX}$
XX	Float-Point-Inexact-Exception Status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Figure 90. Rounding and Range Actions (Part 2)

6.6 DFP Instruction Descriptions

The following sections describe the DFP instructions. When a 128-bit operand is used, it is held in a FPR pair and the instruction mnemonic uses a letter “q” to mean the quad-precision operation. Note that in the following descriptions, FPXp denotes a FPR pair and must address an even-odd pair. If the FPXp field specifies an odd-numbered register, then the instruction form is invalid. The notation FPX[p] means either a FPR, FPX, or a FPR pair, FPXp.

For DFP instructions, if a DFP operand is returned, the trailing significand field of the target operand is encoded using preferred DPD codes.

6.6.1 DFP Arithmetic Instructions

All DFP arithmetic instructions are X-form instructions. They all set the FI and FR status flags, and also set the $FPSCR_{FPRF}$ field. Furthermore, they all have an ideal exponent assigned and employ the record bit (Rc).

The arithmetic instructions consist of Add, Divide, Multiply, and Subtract.

DFP Add [Quad] *X-form*

dadd FRT,FRA,FRB (Rc=0)
dadd. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	2	Rc
0	6	11	16	21	31

daddq FRTp,FRAp,FRBp (Rc=0)
daddq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	2	Rc
0	6	11	16	21	31

The DFP operand in FRA[p] is added to the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

Figure 91 summarizes the actions for Add. Figure 91 does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

DFP Subtract [Quad] *X-form*

dsub FRT,FRA,FRB (Rc=0)
dsub. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	514	Rc
0	6	11	16	21	31

dsubq FRTp,FRAp,FRBp (Rc=0)
dsubq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	514	Rc
0	6	11	16	21	31

The DFP operand in FRB[p] is subtracted from the DFP operand in FRA[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

The execution of Subtract is identical to that of Add, except that the operand in FRB participates in the operation with its sign bit inverted. See Figure 91. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Operand a in FRA[p] is	Actions for Add (a + b) when operand b in FRB[p] is				
	-∞	**F**	**+∞**	**QNaN**	**SNaN**
-∞	T(-dINF)	T(-dINF)	V_{XISI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
F	T(-dINF)	S(a + b)	T(+dINF)	P(b)	V_{XSNAN}: U(b)
+∞	V_{XISI}: T(dNaN)	T(+dINF)	T(+dINF)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

a + b	The value a added to b, rounded to the target-format precision and returned in the appropriate form. (See Section 6.5.11 on page 286)
+dINF	Default plus infinity.
- dINF	Default minus infinity.
dNaN	Default quiet NaN.
F	All finite numbers, including zeros.
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
S(x)	The value x is placed in FRT[p] with the sign set by the rules of algebra. When the source operands have the same sign, the sign of the result is the same as the sign of the operands, including the case when the result is zero. When the operands have opposite signs, the sign of a zero result is positive in all rounding modes, except round toward -∞, in which case, the sign is minus.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XISI}	The Invalid-Operation Exception (VXISI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
V_{XSNAN}	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)

Figure 91. Actions: Add

DFP Multiply [Quad] *X-form*

dmul FRT,FRA,FRB (Rc=0)
dmul. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	34	Rc
0	6	11	16	21	31

dmulq FRTp,FRAp,FRBp (Rc=0)
dmulq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	34	Rc
0	6	11	16	21	31

The DFP operand in FRA[p] is multiplied by the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the sum of the two exponents of the source operands.

Figure 92 summarizes the actions for Multiply. Figure 92 does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXIMZ
CR1 (if Rc=1)

Operand a in FRA[p] is	Actions for Multiply (a*b) when operand b in FRB[p] is				
	0	**Fn**	**∞**	**QNaN**	**SNaN**
0	S(a * b)	S(a * b)	V_{XIMZ}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
Fn	S(a * b)	S(a * b)	S(dINF)	P(b)	V_{XSNAN}: U(b)
∞	V_{XIMZ}: T(dNaN)	S(dINF)	S(dINF)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

a * b	The value a multiplied by b, rounded to the target-format precision and returned in the appropriate form. (See Section 6.5.11 on page 286)
dINF	Default infinity.
dNaN	Default quiet NaN.
Fn	Finite nonzero number (includes both normal and subnormal numbers).
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
S(x)	The value x is placed in FRT[p] with the sign set to the exclusive-OR of the source-operand signs.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XIMZ}:	The Invalid-Operation Exception (VXIMZ) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
V_{XSNAN}:	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)

Figure 92. Actions: Multiply

DFP Divide [Quad] *X-form*

ddiv FRT,FRA,FRB (Rc=0)
ddiv. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	546	Rc
0	6	11	16	21	31

ddivq FRTp,FRAp,FRBp (Rc=0)
ddivq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	546	Rc
0	6	11	16	21	31

The DFP operand in FRA[p] is divided by the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the difference of subtracting the exponent of the divisor from the exponent of the dividend.

Figure 93 summarizes the actions for Divide. Figure 93 does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation and enabled zero-divide exceptions, in which cases the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ
CR1 (if Rc=1)

Operand a in FRA[p] is	Actions for Divide (a ÷ b) when operand b in FRB[p] is				
	0	**Fn**	**∞**	**QNaN**	**SNaN**
0	V_{XZDZ}: T(dNaN)	S(a ÷ b)	S(zt)	P(b)	V_{XSNAN}: U(b)
Fn	Zx: S(dINF)	S(a ÷ b)	S(zt)	P(b)	V_{XSNAN}: U(b)
∞	S(dINF)	S(dINF)	V_{XIDI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

a ÷ b	The value a divided by b, rounded to the target-format precision and returned in the appropriate form. (See Section 6.5.11 on page 286.)
dINF	Default infinity.
dNaN	Default quiet NaN.
Fn	Finite nonzero number (includes both normal and subnormal numbers).
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
S(x)	The value x is placed in FRT[p] with the sign set to the exclusive-OR of the source-operand signs.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XIDI}:	The Invalid-Operation Exception (VXIDI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
V_{XSNAN}:	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
V_{XZDZ}:	The Invalid-Operation Exception (VXZDZ) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 “Invalid Operation Exception” on page 282 for the exception actions.)
zt	True zero (zero significand and most negative exponent).
Zx	The Zero-Divide Exception occurs. The result is produced only when the exception is disabled (See Section 6.5.10.2 “Zero Divide Exception” on page 283 for the exception actions.)

Figure 93. Actions: Divide

6.6.2 DFP Compare Instructions

The DFP compare instructions consist of the *Compare Ordered* and *Compare Unordered* instructions. The compare instructions do not provide the record bit.

The comparison sets the designated CR field to indicate the result. The $FPSCR_{FPCC}$ is set in the same way.

The codes in the CR field BF and $FPSCR_{FPCC}$ are defined for the DFP compare operations as follows.

Bit	Name	Description
0	FL	(FRA[p]) < (FRB[p])
1	FG	(FRA[p]) > (FRB[p])
2	FE	(FRA[p]) = (FRB[p])
3	FU	(FRA[p]) ? (FRB[p])

DFP Compare Unordered [Quad] X-form

dcmpu BF,FRA,FRB

59	BF	//	FRA	FRB	642	/
0	6	9	11	16	21	31

dcmpuq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	642	/
0	6	9	11	16	21	31

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the $FPSCR_{FPCC}$.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN

Operand a in FRA[p] is	Actions for Compare Unordered (a:b) when operand b in FRB[p] is				
	-∞	**F**	**+∞**	**QNaN**	**SNaN**
-∞	AeqB	AltB	AltB	AuoB	Fu, V_{XSNAN}
F	AgtB	C(a:b)	AltB	AuoB	Fu, V_{XSNAN}
+∞	AgtB	AgtB	AeqB	AuoB	Fu, V_{XSNAN}
QNaN	AuoB	AuoB	AuoB	AuoB	Fu, V_{XSNAN}
SNaN	Fu, V_{XSNAN}	Fu, V_{XSNAN}	Fu, V_{XSNAN}	Fu, V_{XSNAN}	Fu, V_{XSNAN}

Explanation:

C(a:b)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0010.
AgtB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0100.
AltB	CR field BF and $FPSCR_{FPCC}$ are set to 0b1000.
AuoB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0001.
V_{XSNAN}	The invalid-operation exception (VXSNAN) occurs. See Section 6.5.10.1 for actions.

Relation of Value a to Value b	Action for C(a:b)
a = b	AeqB
a < b	AltB
a > b	AgtB

Figure 94. Actions: Compare Unordered

DFP Compare Ordered [Quad] X-form

dcmpo BF,FRA,FRB

59	BF	//	FRA	FRB	130	/
0	6	9	11	16	21	31

dcmpoq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	130	/
0	6	9	11	16	21	31

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the $FPSCR_{FPCC}$.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN VXVC

Operand a in FRA[p] is	Actions for Compare ordered (a:b) when operand b in FRB[p] is				
	-∞	**F**	**+∞**	**QNaN**	**SNaN**
-∞	AeqB	AltB	AltB	AuoB, V_{XVC}	AuoB, V_{XSV}
F	AgtB	C(a:b)	AltB	AuoB, V_{XVC}	AuoB, V_{XSV}
+∞	AgtB	AgtB	AeqB	AuoB, V_{XVC}	AuoB, V_{XSV}
QNaN	AuoB, V_{XVC}	AuoB, V_{XVC}	AuoB, V_{XVC}	AuoB, V_{XVC}	AuoB, V_{XSV}
SNaN	AuoB, V_{XSV}	AuoB, V_{XSV}	AuoB, V_{XSV}	AuoB, V_{XSV}	AuoB, V_{XSV}

Explanation:

C(a:b)	Algebraic comparison. See the table below
F	All finite numbers, including zeros
AeqB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0010.
AgtB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0100.
AltB	CR field BF and $FPSCR_{FPCC}$ are set to 0b1000.
AuoB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0001.
V_{XSV}	The invalid-operation exception (VXSNAN) occurs. Additionally, if the exception is disabled ($FPSCR_{VE}$=0), then $FPSCR_{VXVC}$ is also set to one. See Section 6.5.10.1 for actions.
V_{XVC}	The invalid-operation exception (VXVC) occurs. See Section 6.5.10.1 for actions.

Relation of Value a to Value b	Action for C(a:b)
a = b	AeqB
a < b	AltB
a > b	AgtB

Figure 95. Actions: Compare Ordered

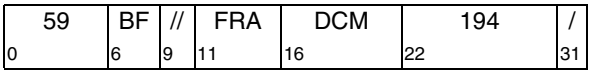
6.6.3 DFP Test Instructions

The DFP test instructions consist of the *Test Data Class*, *Test Data Group*, *Test Exponent*, and *Test Significance* instructions, and they do not provide the record bit.

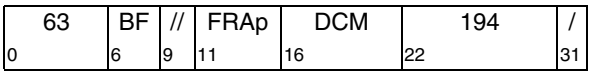
The test instructions set the designated CR field to indicate the result. The $FPSCR_{FPCC}$ is set in the same way.

DFP Test Data Class [Quad] Z22-form

dtstdc BF,FRA,DCM



dtstdcq BF,FRAp,DCM



Let the DCM (Data Class Mask) field specify one or more of the 6 possible data classes, where each bit corresponds to a specific data class.

DCM Bit	Data Class
0	Zero
1	Subnormal
2	Normal
3	Infinity
4	Quiet NaN
5	Signaling NaN

CR field BF and $FPSCR_{FPCC}$ are set to indicate the sign of the DFP operand in FRA[p] and whether the data class of the DFP operand in FRA[p] matches any of the data classes specified by DCM.

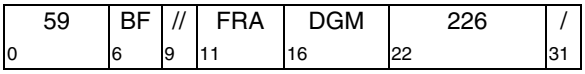
Field	Meaning
0000	Operand positive with no match
0010	Operand positive with match
1000	Operand negative with no match
1010	Operand negative with match

Special Registers Altered:

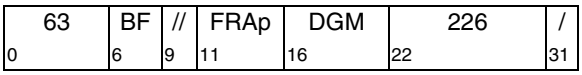
CR field BF
FPCC

DFP Test Data Group [Quad] Z22-form

dtstdg BF,FRA,DGM



dtstdgq BF,FRAp,DGM



Let the DGM (Data Group Mask) field specify one or more of the 6 possible data groups, where each bit corresponds to a specific data group.

The term extreme exponent means either the maximum exponent, X_{max}, or the minimum exponent, X_{min}.

DGM Bit	Data Group
0	Zero with non-extreme exponent
1	Zero with extreme exponent
2	Subnormal or (Normal with extreme exponent)
3	Normal with non-extreme exponent and leftmost zero digit in significand
4	Normal with non-extreme exponent and leftmost nonzero digit in significand
5	Special symbol (Infinity, QNaN, or SNaN)

CR field BF and $FPSCR_{FPCC}$ are set to indicate the sign of the DFP operand in FRA[p] and whether the data group of the DFP operand in FRA[p] matches any of the data groups specified by DGM.

Field	Meaning
0000	Operand positive with no match
0010	Operand positive with match
1000	Operand negative with no match
1010	Operand negative with match

Special Registers Altered:

CR field BF
FPCC

DFP Test Exponent [Quad] *X-form*

dtstex BF,FRA,FRB

59	BF	//	FRA	FRB	162	/
0	6	9	11	16	21	31

dtstexq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	162	/
0	6	9	11	16	21	31

The exponent value (Ea) of the DFP operand in FRA[p] is compared to the exponent value (Eb) of the DFP operand in FRB [p]. The result of the compare is placed into CR field BF and the $FPSCR_{FPCC.}$

The codes in the CR field BF and $FPSCR_{FPCC}$ are defined for the *DFP Test Exponent* operations as follows.

Bit	Description
0	Ea < Eb
1	Ea > Eb
2	Ea = Eb
3	Ea ? Eb

Special Registers Altered:

CR field BF
FPCC

Operand a in FRA[p] is	Actions for Test Exponent (Ea:Eb) when operand b in FRB[p] is			
	F	**∞**	**QNaN**	**SNaN**
F	C(Ea:Eb)	AuoB	AuoB	AuoB
∞	AuoB	AeqB	AuoB	AuoB
QNaN	AuoB	AuoB	AeqB	AeqB
SNaN	AuoB	AuoB	AeqB	AeqB

Explanation:

C(Ea:Eb)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros
AeqB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0010.
AgtB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0100.
AltB	CR field BF and $FPSCR_{FPCC}$ are set to 0b1000.
AuoB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0001.

Relation of Value Ea to Value Eb	Action for C(Ea:Eb)
Ea = Eb	AeqB
Ea < Eb	AltB
Ea > Eb	AgtB

Figure 96. Actions: Test Exponent

DFP Test Significance [Quad] *X-form*

dtstsf BF,FRA,FRB

59	BF	//	FRA	FRB	674	/
0	6	9	11	16	21	31

dtstsfq BF,FRA,FRBp

63	BF	//	FRA	FRBp	674	/
0	6	9	11	16	21	31

Let k be the contents of bits 58:63 of FRA that specifies the reference significance.

The number of significant digits of the DFP operand in FRB[p], NSDb, is compared to the reference significance, k. For this instruction, the number of significant digits of the value 0 is considered to be zero. The result of the compare is placed into CR field BF and the $FPSCR_{FPCC}$ as follows.

Bit	Description
0	k ≠ 0 and k < NSDb
1	k ≠ 0 and k > NSDb, or k = 0
2	k ≠ 0 and k = NSDb
3	k ? NSDb

Special Registers Altered:

CR field BF
FPCC

Programming Note

The reference significance can be loaded into a FPR using a *Load Float as Integer Word Algebraic* instruction

Actions for Test Significance when the operand in FRB[p] is			
F	∞	**QNaN**	**SNaN**
C(k: NSDb)	AuoB	AuoB	AuoB

Explanation:

C(k: NSDb)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0010.
AgtB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0100.
AltB	CR field BF and $FPSCR_{FPCC}$ are set to 0b1000.
AuoB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0001.

Relation of Value NSDb to Value k	**Action for C(k:NSDb)**
k ≠ 0 and k = NSDb	AeqB
k ≠ 0 and k < NSDb	AltB
k ≠ 0 and k > NSDb, or k = 0	AgtB

Figure 97. Actions: Test Significance

6.6.4 DFP Quantum Adjustment Instructions

The *Quantum Adjustment* operations consist of the *Quantize*, *Quantize Immediate*, *Reround*, and *Round To FP Integer* operations.

The *Quantum Adjustment* instructions are Z23-form instructions and have an immediate RMC (Rounding-Mode-Control) field, which specifies the rounding mode used. For *Quantize*, *Quantize Immediate*, and *Reround*, the RMC field contains the primary encoding. For *Round to FP Integer*, the field contains either primary or secondary encoding, depending on the setting of a RMC-encoding-selection bit. See Section 6.5.2 “Rounding Mode Specification” on page 277 for the definition of RMC encoding.

All *Quantum Adjustment* instructions set the FI and FR status flags, and also set the $FPSCR_{FPRF}$ field. The record bit is provided to each of these instructions. They return the target operand in a form with the ideal exponent.

DFP Quantize Immediate [Quad] Z23-form

dquai TE,FRT,FRB,RMC (Rc=0)
dquai. TE,FRT,FRB,RMC (Rc=1)

59	FRT	TE	FRB	RMC	67	Rc
0	6	11	16	21	23	31

dquaiq TE,FRTp,FRBp,RMC (Rc=0)
dquaiq. TE,FRTp,FRBp,RMC (Rc=1)

63	FRTp	TE	FRBp	RMC	67	Rc
0	6	11	16	21	23	31

The DFP operand in FRB[p] is converted and rounded to the form with the exponent specified by TE based on the rounding mode specified in the RMC field. TE is a 5-bit signed binary integer. The result of that form is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the exponent specified by TE.

When the value of the operand in FRB[p] is greater than $(10^{p}\text{-}1) \times 10^{TE}$, where p is the format precision, an invalid operation exception is recognized.

When the delivered result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

DFP Quantize Immediate can be used to adjust values to a form having the specified exponent in the range -16 to 15. If the adjustment requires the significand to be shifted left, then:

- if the result would cause overflow from the most significant digit, the result is a default QNaN.;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of the RMC field.

DFP Quantize Immediate can round a value to a specific number of fractional digits. Consider the computation of sales tax. Values expressed in U.S. dollars have 2 fractional digits, and sales tax rates typically have 3 fractional digits. The product of value and rate will yield 5 fractional digits. For example:

39.95 * 0.075 = 2.99625

This result needs to be rounded to the penny to compute the correct tax of $3.00.

The following sequence computes the sales tax assuming the pre-tax total is in FRA and the tax rate is in FRB. The *DFP Quantize Immediate* instruction rounds the product (FRA * FRB) to 2 fractional digits (TE field = -2) using Round to nearest, ties away from 0 (RMC field = 2). The quantized and rounded result is placed in FRT.

```
dmul  f0,FRA,FRB
dquai -2,FRT,f0,2
```

DFP Quantize [Quad] *Z23-form*

dqua FRT,FRA,FRB,RMC (Rc=0)
dqua. FRT,FRA,FRB,RMC (Rc=1)

59	FRT	FRA	FRB	RMC	3	Rc
0	6	11	16	21	23	31

dquaq FRTp,FRAp,FRBp,RMC (Rc=0)
dquaq. FRTp,FRAp,FRBp,RMC (Rc=1)

63	FRTp	FRAp	FRBp	RMC	3	Rc
0	6	11	16	21	23	31

The DFP operand in register FRB[p] is converted and rounded to the form with the same exponent as that of the DFP operand in FRA[p] based on the rounding mode specified in the RMC field. The result of that form is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the exponent specified in FRA[p].

When the value of the operand in FRB[p] is greater than $(10^{p}\text{-}1) \times 10^{Ea}$, where p is the format precision and Ea is the exponent of the operand in FRA[p], an invalid operation exception is recognized.

When the delivered result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

Figure 99 and Figure 100 summarize the actions. The tables do not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Register Altered:

FPRF FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

DFP Quantize can be used to adjust one DFP value (FRB[p]) to a form having the same exponent as a second DFP value (FRA[p]). If the adjustment requires the significand to be shifted left, then:

- if the result would cause overflow from the most significant digit, the result is a default QNaN.;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of the RMC field. Figure 98 shows examples of these adjustments.

FRA	FRB	FRT when RMC=1	FRT when RMC=2
1 (1×10^{0})	9. (9×10^{0})	9 (9×10^{0})	9 (9×10^{0})
1.00 (100×10^{-2})	9. (9×10^{0})	9.00 (900×10^{-2})	9.00 (900×10^{-2})
1 (1×10^{0})	49.1234 (491234×10^{-4})	49 (49×10^{0})	49 (49×10^{0})
1.00 (100×10^{-2})	49.1234 (491234×10^{-4})	49.12 (4912×10^{-2})	49.12 (4912×10^{-2})
1 (1×10^{0})	49.9876 (499876×10^{-4})	49 (49×10^{0})	50 (50×10^{0})
1.00 (100×10^{-2})	49.9876 (499876×10^{-4})	49.98 (4998×10^{-2})	49.99 (4999×10^{-2})
0.01 (1×10^{-2})	49.9876 (499876×10^{-4})	49.98 (4998×10^{-2})	49.99 (4999×10^{-2})
1 (1×10^{0})	9999999999999999 ($9999999999999999 \times 10^{0}$)	9999999999999999 ($9999999999999999 \times 10^{0}$)	9999999999999999 ($9999999999999999 \times 10^{0}$)
1.0 (10×10^{-1})	9999999999999999 ($9999999999999999 \times 10^{0}$)	QNaN	QNaN

Figure 98. DFP Quantize examples

Operand a in FRA[p] is	Actions for Quantize when operand b in FRB[p] is				
	0	**Fn**	∞	**QNaN**	**SNaN**
0	*	*	V_{XCVI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
Fn	*	*	V_{XCVI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
•	V_{XCVI}: T(dNaN)	V_{XCVI}: T(dNaN)	T(dINF)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

*	See next table.
dINF	Default infinity
dNaN	Default quiet NaN
Fn	Finite nonzero numbers (includes both subnormal and normal numbers)
P(x)	The QNaN of operand x is propagated and placed in FRT[p]
T(x)	The value x is placed in FRT[p]
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XCVI}	The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions)
V_{XSNAN}	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions)

Figure 99. Actions (part 1) Quantize

	Actions for Quantize when operand b in FRB[p] is		
		0	**Fn**
Te < Se	$V_b > (10^p - 1) \times 10^{Te}$	E(0)	V_{XCVI}: T(dNaN)
	$V_b \leq (10^p - 1) \times 10^{Te}$	E(0)	L(b)
Te = Se		E(0)	W(b)
Te > Se		E(0)	QR(b)

Explanation:

dNaN	Default quiet NaN
E(0)	The value of zero with the exponent value Te is placed in FRT[p].
L(x)	The operand x is converted to the form with the exponent value Te.
p	The precision of the format.
QR(x)	The operand x is rounded to the result of the form with the exponent value Te based on the specified rounding mode. The result of that form is placed in FRT[p].
Se	The exponent of the operand in FRB[p].
Te	The target exponent; FRA[p] for ***dqua***[***q***], or TE, a 5-bit signed binary integer for ***dquai***[***q***].
T(x)	The value x is placed in FRT[p].
V_b	The value of the operand in FRB[p].
W(x)	The value and the form of operand x is placed in FRT[p].
V_{XCVI}:	The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions.)

Figure 100.Actions (part2) Quantize

DFP Reround [Quad] Z23-form

drrnd FRT,FRA,FRB,RMC (Rc=0)
drrnd. FRT,FRA,FRB,RMC (Rc=1

59	FRT	FRA	FRB	RMC	35	Rc
0	6	11	16	21	23	31

drrndq FRTp,FRA,FRBp,RMC (Rc=0)
drrndq. FRTp,FRA,FRBp,RMC (Rc=1)

63	FRTp	FRA	FRBp	RMC	35	Rc
0	6	11	16	21	23	31

Let k be the contents of bits 58:63 of FRA that specifies the reference significance.

When the DFP operand in FRB[p] is a finite number, and if the reference significance is zero, or if the reference significance is nonzero and the number of significant digits of the source operand is less than or equal to the reference significance, then the value and the form of the source operand is placed in FRT[p]. If the reference significance is nonzero and the number of significant digits of the source operand is greater than the reference significance, then the source operand is converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field. The result of the form with the specified number of significant digits is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p].

For this instruction, the number of significant digits of the value 0 is considered to be zero. The ideal exponent is the greater value of the exponent of the operand in FRB[p] and the referenced exponent. The referenced exponent is the resultant exponent if the operand in FRB[p] would have been converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field.

If the exponent of the rounded result of the form that has the specified number of significant digits would be greater than $X_{max,}$ an invalid operation exception (VXCVI) occurs. When the invalid-operation exception occurs, and if the exception is disabled, a default QNaN is returned. When an invalid-operation exception occurs, no inexact exception is recognized.

In the absence of an invalid-operation exception, if the result differs in value from the operand in FRB[p], an inexact exception is recognized.

This operation causes neither an overflow nor an underflow exception.

Figure 102 summarizes the actions for *Reround*. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

DFP Reround can be used to adjust a DFP value (FRB[p]) to have no more than a specified number (FRA[p]58:63) of significant digits. The result (FRT[p]) is right-justified leaving the specified number of digits and rounded as specified by the RMC field. If rounding increases the number of significant digits, the result is adjusted again (the significand is shifted right 1 digit and the exponent is incremented by 1). Figure 101 has example results from *DFP Reround* for 1, 2, and 10 significant digits.

Programming Note

DFP Reround is primarily used to round a DFP value to a specific number of digits before conversion to string format for printing or display. Another use for *DFP Reround* is to obtain the effective exponent of the most significant digit by specifying a reference significance of 1. The exponent can be extracted and used to compute the number of significant digits or to left-justify a value.

For example, the following sequence computes the number of significant digits and returns it as an integer. FRB is the DFP value for which we want the number of significant digits; f13 contains the reference significance value 0x0000000000000001; and r1 is the stack pointer, with free space for doublewords at offsets -8 and -16. These doublewords are used to transfer the biased exponents from the FPRs to GPRs for integer computation. R3 contains the result of E(reround(1,FRA)) - E(FRA) + 1, where E(x) represents the biased exponent of x.

```
dxex   f0,FRB
stfd   f0,-16(r1)
drrnd  f1,f13,FRB,1 # reround 1 digit toward 0
dxex   f1,f1
stfd   f1,-8(r1)
lfd    r11,-16(r1)
lfd    r3,-8(r1)
subf   r3,r11,r3
addi   r3,r3,1
```

Given the value 412.34 the result is E(4 x 102) - E(41234 x 10-2) + 1 = (398+2) - (398-2) + 1 = 400 - 396 + 1 = 5. Additional code is required to detect and handle special values like Subnormal, Infinity, and NAN.

$FRA_{58:63}$ (binary)	FRB	FRT when RMC=1	FRT when RMC=2
1	0.41234 (41234×10^{-5})	0.4 (4×10^{-1})	0.4 (4×10^{-1})
1	4.1234 (41234×10^{-4})	4 (4×10^{0})	4 (4×10^{0})
1	41.234 (41234×10^{-3})	4 (4×10^{1})	4 (4×10^{1})
1	412.34 (41234×10^{-2})	4 (4×10^{2})	4 (4×10^{2})
2	0.491234 (491234×10^{-6})	0.49 (49×10^{-2})	0.49 (49×10^{-2})
2	0.499876 (499876×10^{-6})	0.49 (49×10^{-2})	0.50 (50×10^{-2})
2	0.999876 (999876×10^{-6})	0.99 (99×10^{-2})	1.0 (10×10^{-1})
10	0.491234 (491234×10^{-6})	0.491234 (491234×10^{-6})	0.491234 (491234×10^{-6})
10	999.999 (999999×10^{-3})	999.999 (999999×10^{-3})	999.999 (999999×10^{-3})
10	9999999999999999 ($9999999999999999 \times 10^{0}$)	9.999999999E+14 (9999999999×10^{5})	1.000000000E+15 (1000000000×10^{6})

Figure 101.DFP Reround examples

Programming Note

DFP Reround combined with *DFP Quantize* can be used to left justify a value (as needed by the frexp function). FRB is the DFP value for which we want to left justify; f13 contains the reference significance value 0x0000000000000001; and r1 is the stack pointer, with free space for a doubleword at offset -8. This doubleword is used to transfer the biased exponents from the FPR to a GPR, for integer computation. The adjusted biased exponent (+ format precision - 1) is transferred back into an FPR so it can be inserted into the rerounded value. The adjusted rerounded value becomes the quantize reference value. The quantize instruction returns the left justified result in FRT.

```
drrnd  f1,f13,FRB,1 # reround 1 digit toward 0
dxex   f0,f1
stfd   f0,-8(r1)
lfd    r11,-8(r1)
addi   r11,r11,15  # biased exp + precision - 1
lfd    r11,-8(r1)
stfd   f0,-8(r1)
diex   f1,f0,f1    # adjust exponent
dqua   FRT,f1,f0,1 # quantize to adjusted
                     exponent
```

	Actions for Reround when operand b in FRB[p] is				
	0*	**Fn**	**∞**	**QNaN**	**SNaN**
k ≠ 0, k < m	-	RR(b) or V_{XCVI}: T(dNaN)	T(dINF)	P(b)	V_{XSNAN}: U(b)
k ≠ 0, k = m	-	W(b)	T(dINF)	P(b)	V_{XSNAN}: U(b)
k ≠ 0 and k > m, or k = 0	W(b)	W(b)	T(dINF)	P(b)	V_{XSNAN}: U(b)

Explanation:

*	The number of significant digits of the value 0 is considered to be zero for this instruction.
-	Not applicable.
dINF	Default infinity.
Fn	Finite nonzero numbers (includes both subnormal and normal numbers).
k	Reference significance, which specifies the number of significant digits in the target operand.
m	Number of significant digits in the operand in FRB[p].
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
RR(x)	The value x is rounded to the form that has the specified number of significant digits. If $RR(x) \leq (10^k\text{-}1) \times 10^{Xmax}$, then RR(x) is returned; otherwise an invalid-operation exception is recognized.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XCVI}	The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 6.5.10.1 for actions.)
V_{XSNAN}:	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. See Section 6.5.10.1 for actions.
W(x)	The value and the form of x is placed in FRT[p].

Figure 102.Actions: Reround

DFP Round To FP Integer With Inexact [Quad] *Z23-form*

drintx R,FRT,FRB,RMC (Rc=0)
drintx. R,FRT,FRB,RMC (Rc=1)

59	FRT	///	R	FRB	RMC	99	Rc
0	6	11	15	16	21	23	31

drintxq R,FRTp,FRBp,RMC (Rc=0)
drintxq. R,FRTp,FRBp,RMC (Rc=1)

63	FRTp	///	R	FRBp	RMC	99	Rc
0	6	11	15	16	21	23	31

The DFP operand in FRB[p] is rounded to a floating-point integer and placed into FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the larger value of zero and the exponent of the operand in FRB[p].

The rounding mode used is specified in the RMC field. When the RMC-encoding-selection (R) bit is zero, the RMC field contains the primary encoding; when the bit is one, the field contains the secondary encoding.

In addition to coercion of the converted value to fit the target format, the special rounding used by *Round To FP Integer* also coerces the target exponent to the ideal exponent.

When the operand in FRB[p] is a finite number and the exponent is less than zero, the operand is rounded to the result with an exponent of zero. When the exponent is greater than or equal to zero, the result is set to the numerical value and the form of the operand in FRB[p].

When the result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

Figure 103 summarizes the actions for *Round To FP Integer With Inexact*. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN
CR1 (if Rc=1)

Programming Note

The *DFP Round To FP Integer With Inexact* and *DFP Round To FP Integer With Inexact Quad* instructions can be used to implement the decimal equivalent of the C99 rint function by specifying the primary RMC encoding for round according to $FPSCR_{DRN}$ (R=0, RMC=11). The specification for rint requires the inexact exception be raised if detected.

Operand b in FRB is	Is n not precise (n ≠ b)	Inv.-Op. Exception Enabled	Inexact Exception Enabled	Is n Incremented (|n| > |b|)	Actions*
-∞	No[1]	-	-	-	T(-dINF), FI ← 0, FR ← 0
F	No	-	-	-	W(n), FI ← 0, FR ← 0
F	Yes	-	No	No	W(n), FI ← 1, FR ← 0, XX ← 1
F	Yes	-	No	Yes	W(n), FI ← 1, FR ← 1, XX ← 1
F	Yes	-	Yes	No	W(n), FI ← 1, FR ← 0, XX ← 1, TX
F	Yes	-	Yes	Yes	W(n), FI ← 1, FR ← 1, XX ← 1, TX
+∞	No[1]	-	-	-	T(+dINF), FI ← 0, FR ← 0
QNaN	No[1]	-	-	-	P(b), FI ← 0, FR ← 0
SNaN	No[1]	No	-	-	U(b), FI ← 0, FR ← 0, VXSNAN ← 1
SNaN	No[1]	Yes	-	-	VXSNAN ← 1, TV

Explanation:

*	Setting of XX and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR is part of the exception actions.(See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)
-	The actions do not depend on this condition.
1	This condition is true by virtue of the state of some condition to the left of this column.
dINF	Default infinity.
F	All finite numbers, including zeros.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
n	The value derived when the source operand, b, is rounded to an integer using the special rounding for *Round To FP Integer*.
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
T(x)	The value x is placed in FRT[p].
TV	The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TX	The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FPT[p].
W(x)	The value x in the form of zero exponent or the source exponent is placed in FRT[p].
XX	Floating-Point-Inexact-Exception status flag, $FPSCR_{XX}$.

Figure 103.Actions: Round to FP Integer With Inexact

DFP Round To FP Integer Without Inexact [Quad] *Z23-form*

drintn R,FRT,FRB,RMC (Rc=0)
drintn. R,FRT,FRB,RMC (Rc=1)

59	FRT	///	R	FRB	RMC	227	Rc
0	6	11	15	16	21	23	31

drintnq R,FRTp,FRBp,RMC (Rc=0)
drintnq. R,FRTp,FRBp,RMC (Rc=1)

63	FRTp	///	R	FRBp	RMC	227	Rc
0	6	11	15	16	21	23	31

This operation is the same as the *Round To FP Integer With Inexact* operation, except that this operation does not recognize an inexact exception.

Figure 104 summarizes the actions for *Round To FP Integer Without Inexact*. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc=1)

Programming Note

The *DFP Round To FP Integer Without Inexact* and *DFP Round To FP Integer Without Inexact Quad* instructions can be used to implement decimal equivalents of several C99 rounding functions by specifying the appropriate R and RMC field values.

Function	R	RMC
Ceil	1	0b00
Floor	1	0b01
Nearbyint	0	0b11
Round	0	0b10
Trunc	0	0b01

Note that nearbyint is similar to the rint function but without raising the inexact exception. Similarly ceil, floor, round, and trunc do not require the inexact exception.

Operand b in FRB is	Inv.-Op. Exception Enabled	Actions*
-∞	-	T(-dINF), FI ← 0, FR ← 0
F	-	W(n), FI ← 0, FR ← 0
+∞	-	T(+dINF), FI ← 0, FR ← 0
QNaN	-	P(b), FI ← 0, FR ← 0
SNaN	No	U(b), FI ← 0, FR ← 0, VXSNAN←1
SNaN	Yes	VXSNAN ← 1, TV

Explanation:

*	Setting of VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Invalid Operation Exception” for more details.)
-	The actions do not depend on this condition.
dINF	Default infinity.
F	All finite numbers, including zeros.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
n	The value derived when the source operand, b, is rounded to an integer using the special rounding for Round-To-FP-Integer.
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
T(x)	The value x is placed in FRT[p].
TV	The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FPT[p].
W(x)	The value x in the form of zero exponent or the source exponent is placed in FRT[p].

Figure 104.Actions: Round to FP Integer Without Inexact

6.6.5 DFP Conversion Instructions

The DFP conversion instructions consist of data-format conversion instructions and data-type conversion instructions. They are all X-form instructions and employ the record bit (Rc).

6.6.5.1 DFP Data-Format Conversion Instructions

The data-format conversion instructions consist of *Convert To DFP Long*, *Convert To DFP Extended*, *Round To DFP Short*, and *Round To DFP Long*. Figure 105 summarizes the actions for these instructions.

Programming Note

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP *Load Floating-Point Single* and *Store Floating-Point Single* functions, the widening is performed by loading the DFP short value with a *Load Floating as Integer Word Indexed* followed by a *DFP Convert to DFP Long*, and narrowing is performed by a *DFP Round to DFP Short* followed by a *Store Floating-Point as Integer Word Indexed*. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

Instruction	Actions when operand b in FRB[p] is			
	F	∞	QNaN	SNaN
Convert To DFP Long	T(b)[1]	P(b)[2,4]	P(b)[2,4]	P(b)[3,4]
Convert To DFP Extended	T(b)[1]	T(dINF)	P(b)[2,4]	V_{XSNAN}: U(b)[2,4]
Round To DFP Short	R(b)[1]	P(b)[2,5]	P(b)[2,5]	P(b)[3,5]
Round To DFP Long	R(b)[1]	T(dINF)	P(b)[2,5]	V_{XSNAN}: U(b)[2,5]

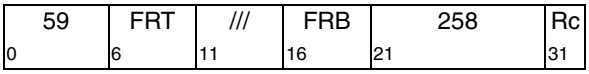
Explanation:

1The ideal exponent is the exponent of the source operand.
2Bits 5:N-1 of the N-bit combination field are set to zero.
3Bit 5 of the N-bit combination field is set to one. Bits 6:N-1 of the combination field are set to zero.
4The trailing significand field is padded on the left with zeros.
5Leftmost digits in the trailing significand field are removed.
dINFDefault infinity.
FAll finite numbers, including zeros.
P(x)The special symbol in operand x is propagated into FRT[p].
R(x)The value x is rounded to the target-format precision; see Section 6.5.11
T(x)The value x is placed in FRT[p].
U(x)The SNaN of operand x is converted to the corresponding QNaN.
V_{XSNAN}The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. See Section 6.5.10.1 for actions.

Figure 105.Actions: Data-Format Conversion Instructions

DFP Convert To DFP Long X-form

dctdp FRT,FRB (Rc=0)
dctdp. FRT,FRB (Rc=1)



The DFP short operand in bits 32:63 of FRB is converted to DFP long format and the converted result is placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP long format and does not cause an invalid-operation exception.

Special Registers Altered:

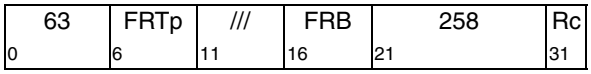
FPRF FR (undefined) FI (undefined)
CR1 (if Rc=1)

Programming Note

Note that DFP short format is a storage-only format, Therefore, conversion of a short SNaN to long format will not cause an exception and the SNaN is preserved. Subsequent operation on that SNaN in long format will cause an exception.

DFP Convert To DFP Extended X-form

dctqpq FRTp,FRB (Rc=0)
dctqpq. FRTp,FRB (Rc=1)



The DFP long operand in the FRB is converted to DFP extended format and placed into FRTp. The sign of the result is the same as the sign of the operand in FRB. The ideal exponent is the exponent of the operand in FRB.

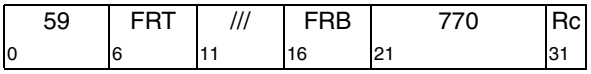
If the operand in FRB is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP extended format.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc=1)

DFP Round To DFP Short X-form

drsp FRT,FRB (Rc=0)
drsp. FRT,FRB (Rc=1)



59	FRT	///	FRB	770	Rc
0	6	11	16	21	31

The DFP long operand in FRB is converted and rounded to DFP short format. The DFP short value is extended on the left with zeros to form a 64-bit entity and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP short format and does not cause an invalid-operation exception.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
CR1 (if Rc=1)

Programming Note

Note that DFP short format is a storage-only format, Therefore, conversion of a long SNaN to short format will not cause an exception. Converting a long format SNaN to short format is an implied move operation.

DFP Round To DFP Long X-form

drdpq FRTp,FRBp (Rc=0)
drdpq. FRTp,FRBp (Rc=1)

63	FRTp	///	FRBp	770	Rc
0	6	11	16	21	31

The DFP extended operand in FRBp is converted and rounded to DFP long format. The result concatenated with 64 0s is placed in FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the operand in FRBp.

If the operand in FRBp is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP long format.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN
CR1 (if Rc=1)

Programming Note

Note that DFP Round to DFP Long, while producing a result in DFP long format, actually targets a register pair, writing 64 0s in FRTp+1.

6.6.5.2 DFP Data-Type Conversion Instructions

The DFP data-type conversion instructions are used to convert data type between DFP and fixed.

The data-type conversion instructions consist of *Convert From Fixed* and *Convert To Fixed*.

DFP Convert From Fixed *X-form*

dcffix FRT,FRB (Rc=0)
dcffix. FRT,FRB (Rc=1)

59	FRT	///	FRB	802	Rc
0	6	11	16	21	31

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Long value and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

The $FPSCR_{FPRF}$ field is set to the class and sign of the result.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

DFP Convert From Fixed Quad *X-form*

dcffixq FRTp,FRB (Rc=0)
dcffixq. FRTp,FRB (Rc=1)

63	FRTp	///	FRB	802	Rc
0	6	11	16	21	31

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Extended value and placed into FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

The $FPSCR_{FPRF}$ field is set to the class and sign of the result.

Special Registers Altered:

FPRF FR (undefined) FI (undefined)
CR1 (if Rc=1)

DFP Convert To Fixed [Quad] *X-form*

dctfix FRT,FRB (Rc=0)
dctfix. FRT,FRB (Rc=1)

59	FRT	///	FRB	290	Rc
0	6	11	16	21	31

dctfixq FRT,FRBp (Rc=0)
dctfixq. FRT,FRBp (Rc=1)

63	FRT	///	FRBp	290	Rc
0	6	11	16	21	31

The DFP operand in FRB[p] is rounded to an integer value and is placed into FRT in the 64-bit signed binary integer format. The sign of the result is the same as the sign of the source operand, except when the source operand is a NaN or a zero.

Figure 106 summarizes the actions for *Convert To Fixed*.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

It is recommended that software pre-round the operand to a floating-point integral using ***drintx***[***q***] or ***drintn***[***q***] is a rounding mode other than the current rounding mode specified by $FPSCR_{DRN}$ is needed. Saving, modifying and restoring the FPSCR just to temporarily change the rounding mode is less efficient than just employing drintx[p] or drint[p] which override the current rounding mode using an immediate control field.

For example if the desired function rounding is Round to Nearest, Ties away from 0 but the default rounding (from $FPSCR_{DRN}$) is Round to Nearest, Ties to Even then following is preferred.

```
drintn    0,f1,f1,2
dctfix    f1,f1
```

Operand b in FRB[p] is	q is	Is n not precise (n ≠ b)	Inv.-Op. Except. Enabled	Inexact Except. Enabled	Is n Incremented (|n| > |b|)	Actions *
-∞ ≤ b < MN	< MN	-	No	-	-	T(MN), FI ← 0, FR ← 0, VXCVI ← 1
-∞ ≤ b < MN	< MN	-	Yes	-	-	VXCVI ← 1, TV
-∞ < b < MN	= MN	-	-	No	-	T(MN), FI ← 1, FR ← 0, XX ← 1
-∞ < b < MN	= MN	-	-	Yes	-	T(MN), FI ← 1, FR ← 0, XX ← 1,TX
MN ≤ b < 0	-	No	-	-	-	T(n), FI ← 0, FR ← 0
MN ≤ b < 0	-	Yes	-	No	No	T(n), FI ← 1, FR ← 0, XX ← 1
MN ≤ b < 0	-	Yes	-	No	Yes	T(n), FI ← 1, FR ← 1, XX ← 1
MN ≤ b < 0	-	Yes	-	Yes	No	T(n), FI ← 1, FR ← 0, XX ← 1, TX
MN ≤ b < 0	-	Yes	-	Yes	Yes	T(n), FI ← 1, FR ← 1, XX ← 1, TX
±0	-	No	-	-	-	T(0), FI ← 0, FR ← 0
0 < b ≤ MP	-	No	-	-	-	T(n), FI ← 0, FR ← 0
0 < b ≤ MP	-	Yes	-	No	No	T(n), FI ← 1, FR ← 0, XX ← 1
0 < b ≤ MP	-	Yes	-	No	Yes	T(n), FI ← 1, FR ← 1, XX ← 1
0 < b ≤ MP	-	Yes	-	Yes	No	T(n), FI ← 1, FR ← 0, XX ← 1, TX
0 < b ≤ MP	-	Yes	-	Yes	Yes	T(n), FI ← 1, FR ← 1, XX ← 1, TX
MP < b < +∞	= MP	-	-	No	-	T(MP), FI ← 1, FR ← 0, XX ← 1
MP < b < +∞	= MP	-	-	Yes	-	T(MP), FI ← 1, FR ← 0, XX ← 1, TX
MP < b ≤ +∞	> MP	-	No	-	-	T(MP), FI ← 0, FR ← 0, VXCVI ← 1
MP < b ≤ +∞	> MP	-	Yes	-	-	VXCVI ← 1, TV
QNaN	-	-	No	-	-	T(MN), FI←0, FR←0, VXCVI←1
QNaN	-	-	Yes	-	-	VXCVI←1, TV
SNaN	-	-	No	-	-	T(MN),FI←0, FR←0, VXCVI←1,VXSNAN ←1
SNaN	-	-	Yes	-	-	VXCVI←1,VXSNAN ← 1, TV

Explanation:

*	Setting of XX, VXCVI, and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)
-	The actions do not depend on this condition.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
MN	Maximum negative number representable by the 64-bit binary integer format
MP	Maximum positive number representable by the 64-bit binary integer format.
n	The value q converted to a fixed-point result.
q	The value derived when the source value b is rounded to an integer using the specified rounding mode
T(x)	The value x is placed in FRT[p].
TV	The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TX	The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
VXCVI	The $FPSCR_{VXCVI}$ invalid operation exception status bit.
VXSNAN	The $FPSCR_{VXSNAN}$ invalid operation exception status bit.
XX	Floating-Point-Inexact-Exception status flag, $FPSCR_{XX.}$

Figure 106.Actions: Convert To Fixed

6.6.6 DFP Format Instructions

The DFP format instructions are used to compose or decompose a DFP operand. A source operand of SNaN does not cause an invalid-operation exception. All format instructions employ the record bit (Rc).

The format instructions consist of *Decode DPD To BCD*, *Encode BCD To DPD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*.

DFP Decode DPD To BCD [Quad] X-form

ddedpd SP,FRT,FRB (Rc=0)
ddedpd. SP,FRT,FRB (Rc=1)

59	FRT	SP	///	FRB	322	Rc
0	6	11	13	16	21	31

ddedpdq SP,FRTp,FRBp (Rc=0)
ddedpdq. SP,FRTp,FRBp (Rc=1)

63	FRTp	SP	///	FRBp	322	Rc
0	6	11	13	16	21	31

A portion of the significand of the DFP operand in FRB[p] is converted to a signed or unsigned BCD number depending on the SP field. For infinity and NaN, the significand is considered to be the contents in the trailing significand field padded on the left by a zero digit.

SP_0 = 0 (unsigned conversion)

The rightmost 16 digits of the significand (32 digits for ***ddedpdq***) is converted to an unsigned BCD number and the result is placed into FRT[p].

SP_0 = 1 (signed conversion)

The rightmost 15 digits of the significand (31 digits for ***ddedpdq***) is converted to a signed BCD number with the same sign as the DFP operand, and the result is placed into FRT[p]. If the DFP operand is negative, the sign is encoded as 0b1101. If the DFP operand is positive, SP_1 indicates which preferred plus sign encoding is used. If SP_1 = 0, the plus sign is encoded as 0b1100 (the option-1 preferred sign code), otherwise the plus sign is encoded as 0b1111(the option-2 preferred sign code).

Special Registers Altered:

CR1 (if Rc=1)

DFP Encode BCD To DPD [Quad] X-form

denbcd S,FRT,FRB (Rc=0)
denbcd. S,FRT,FRB (Rc=1)

59	FRT	S	///	FRB	834	Rc
0	6	11	12	16	21	31

denbcdq S,FRTp,FRBp (Rc=0)
denbcdq. S,FRTp,FRBp (Rc=1)

63	FRTp	S	///	FRBp	834	Rc
0	6	11	12	16	21	31

The signed or unsigned BCD operand, depending on the S field, in FRB[p] is converted to a DFP number. The ideal exponent is zero.

S = 0 (unsigned BCD operand)

The unsigned BCD operand in FRB[p] is converted to a positive DFP number of the same magnitude and the result is placed into FRT[p].

S = 1 (signed BCD operand)

The signed BCD operand in FRB[p] is converted to the corresponding DFP number and the result is placed into FRT[p].

If an invalid BCD digit or sign code is detected in the source operand, an invalid-operation exception (VXCVI) occurs.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exception when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXCVI
CR1 (if Rc=1)

DFP Extract Biased Exponent [Quad]
X-form

dxex FRT,FRB (Rc=0)
dxex. FRT,FRB (Rc=1)

59	FRT	///	FRB	354	Rc
0	6	11	16	21	31

dxexq FRT,FRBp (Rc=0)
dxexq. FRT,FRBp (Rc=1)

63	FRT	///	FRBp	354	Rc
0	6	11	16	21	31

The biased exponent of the operand in FRB[p] is extracted and placed into FRT in the 64-bit signed binary integer format. When the operand in FRB is an infinity, QNaN, or SNaN, a special code is returned.

Operand	Result
Finite Number	biased exponent value
Infinity	-1
QNaN	-2
SNaN	-3

Special Registers Altered:

CR1 (if Rc=1)

Programming Note

The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.

DFP Insert Biased Exponent [Quad]
X-form

diex FRT,FRA,FRB (Rc=0)
diex. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	866	Rc
0	6	11	16	21	31

diexq FRTp,FRA,FRBp (Rc=0)
diexq. FRTp,FRA,FRBp (Rc=1)

63	FRTp	FRA	FRBp	866	Rc
0	6	11	16	21	31

Let *a* be the value of the 64-bit signed binary integer in FRA.

a	**Result**
a > MBE[1]	QNaN
MBE ≥ a ≥ 0	Finite number with biased exponent a
a = -1	Infinity
a = -2	QNaN
a = -3	SNaN
a < -3	QNaN

[1] Maximum biased exponent for the target format

When $0 \le a \le$ MBE, *a* is the biased target exponent that is combined with the sign bit and the significand value of the DFP operand in FRB[p] to form the DFP result in FRT[p]. The ideal exponent is the specified target exponent.

When *a* specifies a special code (a < 0 or a > MBE), an infinity, QNaN, or SNaN is formed in FRT[p] with the trailing significand field containing the value from the trailing significand field of the source operand in FRB[p], and with an N-bit combination field set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-5 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-5 bits are set to zero.

Special Registers Altered:

CR1 (if Rc=1)

Programming Note

The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.

Operand a in FRA[p] specifies	Actions for Insert Biased Exponent when operand b in FRB[p] specifies			
	F	∞	**QNaN**	**SNaN**
F	N, Rb	Z, Rb	Z, Rb	Z, Rb
∞	I, Rb	I, Rb	I, Rb	I, Rb
QNaN	Q, Rb	Q, Rb	Q, Rb	Q, Rb
SNaN	S, Rb	S, Rb	S, Rb	S, Rb

Explanation:

F	All finite numbers, including zeros
I	The combination field in FRT[p] is set to indicate a default Infinity.
N	The combination field in FRT[p] is set to the specified biased exponent in FRA and the leftmost significand digit in FRB[p].
Q	The combination field in FRT[p] is set to indicate a default QNaN.
S	The combination field in FRT[p] is set to indicate a default SNaN.
Z	The combination field in FRT[p] is set to indicate the specific biased exponent in FRA and a leftmost coefficient digit of zero.
Rb	The contents of the trailing significand field in FRB[p] are reencoded using preferred DPD encodings and the reencoded result is placed in the same field in FRT[p]. The sign bit of FRB[p] is copied into the sign bit in FRT[p].

Figure 107.Actions: Insert Biased Exponent

DFP Shift Significand Left Immediate [Quad] Z22-form

dscli FRT,FRA,SH (Rc=0)
dscli. FRT,FRA,SH (Rc=1)

59	FRT	FRA	SH	66	Rc
0	6	11	16	22	31

dscliq FRTp,FRAp,SH (Rc=0)
dscliq. FRTp,FRAp,SH (Rc=1)

63	FRTp	FRAp	SH	66	Rc
0	6	11	16	22	31

The significand of the DFP operand in FRA[p] is shifted left SH digits. For a NaN or infinity, all significand digits are in the trailing significand field. SH is a 6-bit unsigned binary integer. Digits shifted out of the leftmost digit are lost. Zeros are supplied to the vacated positions on the right. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format's N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

Special Registers Altered:

CR1 (if Rc=1)

DFP Shift Significand Right Immediate [Quad] Z22-form

dscri FRT,FRA,SH (Rc=0)
dscri. FRT,FRA,SH (Rc=1)

59	FRT	FRA	SH	98	Rc
0	6	11	16	22	31

dscriq FRTp,FRAp,SH (Rc=0)
dscriq. FRTp,FRAp,SH (Rc=1)

63	FRTp	FRAp	SH	98	Rc
0	6	11	16	22	31

The significand of the DFP operand in FRA[p] is shifted right SH digits. For a NaN or infinity, all significand digits are in the trailing significand field. SH is a 6-bit unsigned binary integer. Digits shifted out of the units digit are lost. Zeros are supplied to the vacated positions on the left. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format's N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

Special Registers Altered:

CR1 (if Rc=1)

6.6.7 DFP Instruction Summary

Mnemonic	Full Name	FORM	Operands	SNaN Vs	SNaN G	Encoding	FPRF C	FPRF FPCC	FP Exception V	Z	O	U	X	FRI	IE	Rc
dadd	DFP Add	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
daddq	DFP Add Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dsub	DFP Subtract	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dsubq	DFP Subtract Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dmul	DFP Multiply	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dmulq	DFP Multiply Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
ddiv	DFP Divide	X	FRT, FRA, FRB	Y	N	RE	Y	Y	V	Z	O	U	X	Y	Y	Y
ddivq	DFP Divide Quad	X	FRTp, FRAp, FRBp	Y	N	RE	Y	Y	V	Z	O	U	X	Y	Y	Y
dcmpo	DFP Compare Ordered	X	BF, FRA, FRB	Y	-	-	N	Y	V					-	-	N
dcmpoq	DFP Compare Ordered Quad	X	BF, FRAp, FRBp	Y	-	-	N	Y	V					-	-	N
dcmpu	DFP Compare Unordered	X	BF, FRA, FRB	Y	-	-	N	Y	V					-	-	N
dcmpuq	DFP Compare Unordered Quad	X	BF, FRAp, FRBp	Y	-	-	N	Y	V					-	-	N
dtstdc	DFP Test Data Class	Z22	BF, FRA, DCM	N	-	-	N	Y[1]						-	-	N
dtstdcq	DFP Test Data Class Quad	Z22	BF, FRAp, DCM	N	-	-	N	Y[1]						-	-	N
dtstdg	DFP Test Data Group	Z22	BF, FRA,DGM	N	-	-	N	Y[1]						-	-	N
dtstdgq	DFP Test Data Group Quad	Z22	BF, FRAp, DGM	N	-	-	N	Y[1]						-	-	N
dtstex	DFP Test Exponent	X	BF, FRA, FRB	N	-	-	N	Y						-	-	N
dtstexq	DFP Test Exponent Quad	X	BF, FRAp, FRBp	N	-	-	N	Y						-	-	N
dtstsf	DFP Test Significance	X	BF, FRA(FIX), FRB	N	-	-	N	Y						-	-	N
dtstsfq	DFP Test Significance Quad	X	BF, FRA(FIX), FRBp	N	-	-	N	Y						-	-	N
dquai	DFP Quantize Immediate	Z23	TE, FRT, FRB, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
dquaiq	DFP Quantize Immediate Quad	Z23	TE, FRTp, FRBp, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
dqua	DFP Quantize	Z23	FRT,FRA,FRB,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
dquaq	DFP Quantize Quad	Z23	FRTp,FRAp,FRBp, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drrnd	DFP Reround	Z23	FRT,FRA(FIX),FRB,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drrndq	DFP Reround Quad	Z23	FRTp, FRA(FIX), FRBp, RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drintx	DFP Round To FP Integer With Inexact	Z23	R,FRT, FRB,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drintxq	DFP Round To FP Integer With Inexact Quad	Z23	R,FRTp,FRBp,RMC	Y	N	RE	Y	Y	V				X	Y	Y	Y
drintn	DFP Round To FP Integer With-out Inexact	Z23	R,FRT, FRB,RMC	Y	N	RE	Y	Y	V					Y[#]	Y	Y
drintnq	DFP Round To FP Integer With-out Inexact Quad	Z23	R,FRTp, FRBp,RMC	Y	N	RE	Y	Y	V					Y[#]	Y	Y
dctdp	DFP Convert To DFP Long	X	FRT, FRB (DFP Short)	N	Y	RE	Y	Y[2]						U	Y	Y
dctqpq	DFP Convert To DFP Extended	X	FRTp, FRB	Y	N	RE	Y	Y	V					Y[#]	Y	Y
drsp	DFP Round To DFP Short	X	FRT (DFP Short), FRB	N	Y	RE	Y	Y[2]				O	U X	Y	Y	Y
drdpq	DFP Round To DFP Long	X	FRTp, FRBp	Y	N	RE	Y	Y	V		O	U	X	Y	Y	Y
dcffixq	DFP Convert From Fixed Quad	X	FRTp, FRB (FIX)	-	N	RE	Y	Y						U	Y	Y
dctfix	DFP Convert To Fixed	X	FRT (FIX), FRB	Y	N	-	U	U	V				X	Y	-	Y
dctfixq	DFP Convert To Fixed Quad	X	FRT (FIX), FRBp	Y	N	-	U	U	V				X	Y	-	Y
ddedpd	DFP Decode DPD To BCD	X	SP, FRT(BCD), FRB	N	-	-	N	N						-	-	Y

Figure 108.Decimal Floating-Point Instructions Summary

Mnemonic	Full Name	FORM	Operands	SNaN Vs	SNaN G	Encoding	FPRF C	FPRF FPCC	FP Exception V Z O U X	FR|FI	IE	Rc
ddedpdq	DFP Decode DPD To BCD Quad	X	SP, FRTp(BCD), FRBp	N	-	-	N	N		-	-	Y
denbcd	DFP Encode BCD To DPD	X	S, FRT, FRB (BCD)	-	N	RE	Y	Y	V	Y#	Y	Y
denbcdq	DFP Encode BCD To DPD Quad	X	S, FRTp, FRBp (BCD)	-	N	RE	Y	Y	V	Y#	Y	Y
dxex	DFP Extract Biased Exponent	X	FRT (FIX), FRB	N	N	-	N	N		-	-	Y
dxexq	DFP Extract Biased Exponent Quad	X	FRT (FIX), FRBp	N	N	-	N	N		-	-	Y
diex	DFP Insert Biased Exponent	X	FRT, FRA(FIX), FRB	N	Y	RE	N	N		-	Y	Y
diexq	DFP Insert Biased Exponent Quad	X	FRTp, FRA(FIX), FRBp	N	Y	RE	N	N		-	Y	Y
dscli	DFP Shift Significand Left Immediate	Z22	FRT,FRA,SH	N	Y	RE	N	N		-	-	Y
dscliq	DFP Shift Significand Left Immediate Quad	Z22	FRTp,FRAp,SH	N	Y	RE	N	N		-	-	Y
dscri	DFP Shift Significand Right Immediate	Z22	FRT,FRA,SH	N	Y	RE	N	N		-	-	Y
dscriq	DFP Shift Significand Right Immediate Quad	Z22	FRTp,FRAp,SH	N	Y	RE	N	N		-	-	Y

Explanation:

#	FI and FR are set to zeros for these instructions.
-	Not applicable.
1	A unique definition of the $FPSCR_{FPCC}$ field is provided for the instruction.
2	These are the only instructions that may generate an SNaN and also set the $FPSC_{FPRF}$ field. Since the BFP $FPSCR_{FPRF}$ field does not include a code for SNaN, these instructions cause the need for redefining the $FPSCR_{FPRF}$ field for DFP.
DCM	A 6-bit immediate operand specifying the data-class mask.
DGM	A 6-bit immediate operand specifying the data-group mask.
G	An SNaN can be generated as the target operand.
IE	An ideal exponent is defined for the instruction.
FI	Setting of the $FPSCR_{FI}$ flag.
FR	Setting of the $FPSCR_{FR}$ flag.
N	No.
O	An overflow exception may be recognized.
Rc	The record bit, Rc, is provided to record $FPSCR_{0:3}$ in CR field 1.
RE	The trailing significand field is reencoded using preferred DPD encodings.The preferred DPD encoding are also used for propagated NaNs, or converted NaNs and infinities.
RMC	A 2-bit immediate operand specifying the rounding-mode control.
S	An one-bit immediate operand specifying if the operation is signed or unsigned.
SP	A two-bit immediate operand: one bit specifies if the operation is signed or unsigned and, for signed operations, another bit specifies which preferred plus sign code is generated.
U	An underflow exception may be recognized.
V	An invalid-operation exception may be recognized.
Vs	An input operand of SNaN causes an invalid-operation exception.
X	An inexact exception may be recognized.
Y	Yes.
U	Undefined
Z	A zero-divide exception may be recognized.

Figure 108.Decimal Floating-Point Instructions Summary (Continued)

Chapter 7. Vector-Scalar Floating-Point Operations [Category: VSX]

7.1 Introduction

7.1.1 Overview of the Vector-Scalar Extension

Category Vector-Scalar Extension (VSX) provides facilities supporting vector and scalar binary floating-point operations. The following VSX features are provided to increase opportunities for vectorization.

– A unified register file, a set of Vector-Scalar Registers (VSR), supporting both scalar and vector operations is provided, eliminating the overhead of vector-scalar data transfer through storage.

– Support for word-aligned storage accesses for both scalar and vector operations is provided.

– Robust support for IEEE-754 for both vector and scalar floating-point operations is provided.

Combining the Floating-Point Registers (FPR) defined in Chapter 4. Floating-Point Facility [Category: Floating-Point] and the Vector Registers (VR) defined in Chapter 5. Vector Facility [Category: Vector] provides additional registers to support more aggressive compiler optimizations for both vector and scalar operations.

Implementations of VSX must also implement the Floating-Point (Chapter 4) and Vector (Chapter 5) categories.

7.1.1.1 Compatibility with Category Floating-Point and Category Decimal Floating-Point Operations

The instruction sets defined in Chapter 4. Floating-Point Facility [Category: Floating-Point] and Chapter 6. Decimal Floating-Point [Category: Decimal Floating-Point] retain their definition with one primary difference. The FPRs are mapped to doubleword element 0 of VSRs 0-31. The contents of doubleword 1 of the VSR corresponding to a source FPR specified by an instruction are ignored. The contents of doubleword 1 of a VSR corresponding to the target FPR specified by an instruction are undefined.

Programming Note

Application binary interfaces extended to support VSX require special care of vector data written to VSRs 0-31 (i.e., VSRs corresponding to FPRs). Legacy scalar function calls employ doubleword-based loads and stores to preserve the contents of any nonvolatile registers, This has the adverse effect of not preserving the contents of doubleword 1 of these VSRs.

7.1.1.2 Compatibility with Category Vector Operations

The instruction set defined in Chapter 5. Vector Facility [Category: Vector], retains its definition with one primary difference. The VRs are mapped to VSRs 32-63.

7.2 VSX Registers

7.2.1 Vector-Scalar Registers

Sixty-four 128-bit VSRs are provided. See Figure 109 All VSX floating-point computations and other data manipulation are performed on data residing in Vector-Scalar Registers, and results are placed into a VSR.

Depending on the instruction, the contents of a VSR are interpreted as a sequence of equal-length elements (words or doublewords) or as a quadword. Each of the elements is aligned at its natural boundary within the VSR, as shown in Figure 109. Many instructions perform a given operation in parallel on all elements in a VSR. Depending on the instruction, a word element can be interpreted as a signed integer word (SW), an unsigned integer word (UW), a logical mask value (MW), or a single-precision floating-point value (SP); a doubleword element can be interpreted as a doubleword signed integer (SD), a doubleword unsigned integer (UD), a doubleword mask (DM), or a double-precision floating-point value (DP). In the instructions descriptions, phrases like *signed integer word element* are used as shorthand for *word element, interpreted as a signed integer*.

Load and *Store* instructions are provided that transfer a byte, halfword, word, doubleword, or quadword between storage and a VSR.

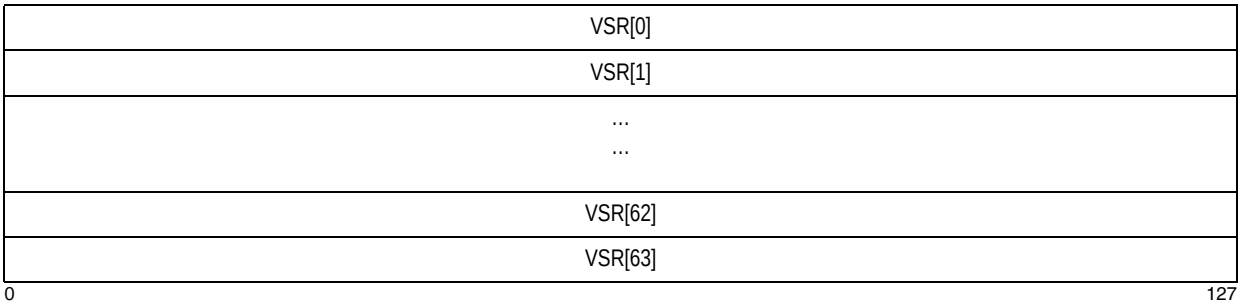


Figure 109.Vector-Scalar Registers

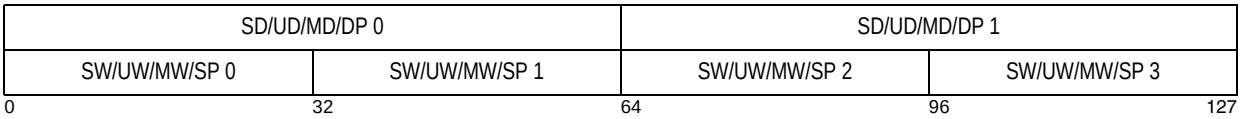


Figure 110.Vector-Scalar Register Elements

7.2.1.1 Floating-Point Registers

Chapter 4. Floating-Point Facility [Category: Floating-Point] provides 32 64-bit FPRs. Chapter 6. Decimal Floating-Point [Category: Decimal Floating-Point] also employs FPRs in decimal floating-point (DFP) operations. When VSX is implemented, the 32 FPRs are mapped to doubleword 0 of VSRs 0-31. For example, FPR[0] is located in doubleword element 0 of VSR[0], FPR[1] is located in doubleword element 0 of VSR[1], and so forth.

All instructions that operate on an FPR are redefined to operate on doubleword element 0 of the corresponding VSR. The contents of doubleword element 1 of the VSR corresponding to a source FPR or FPR pair for these instructions are ignored and the contents of doubleword element 1 of the VSR corresponding to the target FPR or FPR pair for these instructions are undefined.

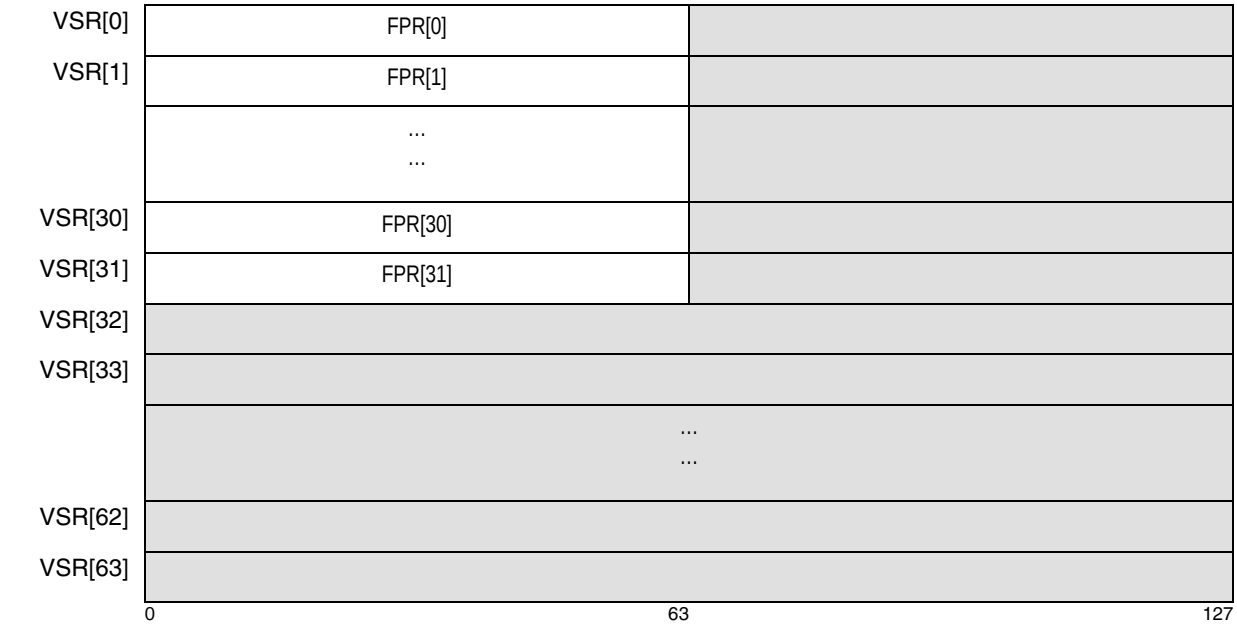


Figure 111.Floating-Point Registers as part of VSRs

7.2.1.2 Vector Registers

Chapter 5. Vector Facility [Category: Vector] provides 32 128-bit VRs. When VSX is implemented, the 32 VRs are mapped to VSRs 32-63. For example, VR[0] is located in VSR[32], VR[1] is located in VSR[33], and so forth.

All instructions that operate on a VR are redefined to operate on the corresponding VSR.

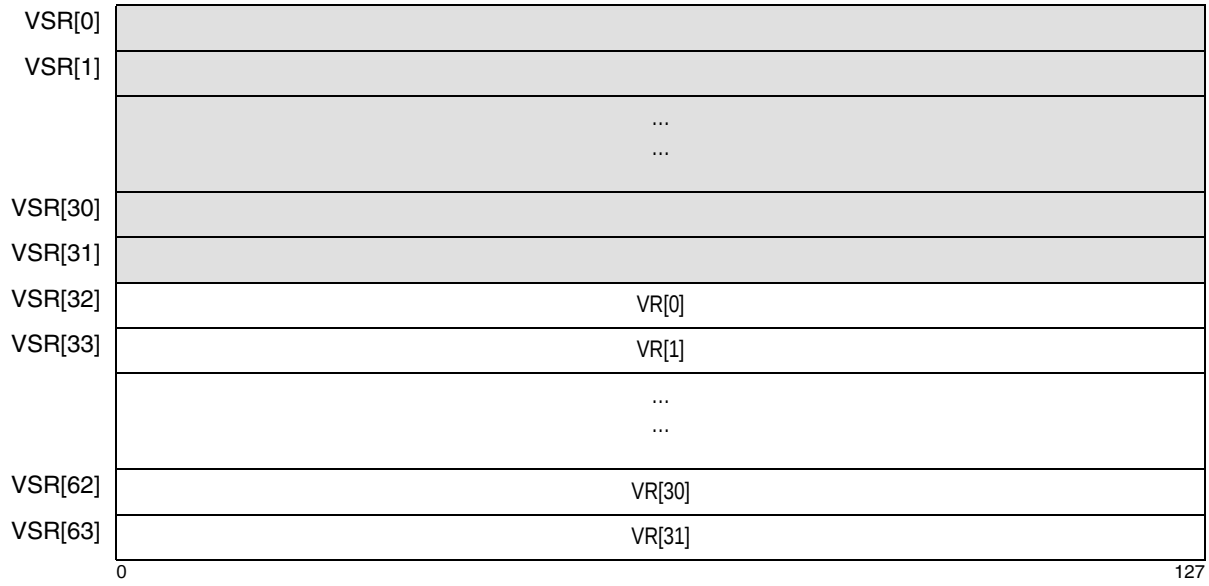


Figure 112.Vector Registers as part of VSRs

7.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 0:19 and 32:55 are status bits. Bits 56:63 are control bits.

The exception status bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mcrfs***, ***mtfsfi***, ***mtfsf***, or ***mtfsb0*** instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be "exception status bits", and only FX is sticky.

Programming Note

Access to *Move To FPSCR* and *Move From FPSCR* instructions requires FP=1.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

The bit definitions for the FPSCR are as follows.

Bits **Definition**

0:28 **Decimal Floating-Point Rounding Control** (DRN)
This field is not used by VSX instructions.

32 **Floating-Point Exception Summary** (FX)
Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets FX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter FX explicitly.

Programming Note

FX is defined not to be altered implicitly by ***mtfsfi*** and ***mtfsf*** because permitting these instructions to alter FX implicitly can cause a paradox. An example is an ***mtfsfi*** or ***mtfsf*** instruction that supplies 0 for FX and 1 for OX, and is executed when OX=0. See also the Programming Notes with the definition of these two instructions.

33 **Floating-Point Enabled Exception Summary** (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter FEX explicitly.

Bits **Definition**

34 **Floating-Point Invalid Operation Exception Summary** (VX)
This bit is the OR of all the Invalid Operation exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter VX explicitly.

35 **Floating-Point Overflow Exception** (OX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion* or *VSX Vector DP-SP Conversion* class instruction causes an Overflow exception. See Section 7.4.3 , "Floating-Point Overflow Exception" on page 351.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

36 **Floating-Point Underflow Exception** (UX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion* or *VSX Vector DP-SP Conversion* class instruction causes an Underflow exception. See Section 7.4.4 , "Floating-Point Underflow Exception" on page 353.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

37 **Floating-Point Zero Divide Exception** (ZX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* or *VSX Vector Floating-Point Arithmetic* class instruction causes an Zero Divide exception. See Section 7.4.2 , "Floating-Point Zero Divide Exception" on page 349.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

38 **Floating-Point Inexact Exception** (XX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar Integer Conversion*, *VSX Vector Integer Conversion*, *VSX Scalar Round to Floating-Point Integer*, or *VSX Vector Round to Floating-Point Integer* class instruction causes an Inexact exception. See Section 7.4.5 , "Floating-Point Inexact Exception" on page 356.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

Bits	Definition
39	**Floating-Point Invalid Operation Exception (SNAN)** (VXSNAN) This bit is set to 1 when a *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instruction causes an SNaN type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
40	**Floating-Point Invalid Operation Exception (Inf-Inf)** (VXISI) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes an Infinity – Infinity type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
41	**Floating-Point Invalid Operation Exception (Inf÷Inf)** (VXIDI) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes an Infinity ÷ Infinity type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
42	**Floating-Point Invalid Operation Exception (Zero÷Zero)** (VXZDZ) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes a Zero ÷ Zero type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
43	**Floating-Point Invalid Operation Exception (Inf×Zero)** (VXIMZ) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes a Infinity × Zero type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
44	**Floating-Point Invalid Operation Exception (Invalid Compare)** (VXVC) This bit is set to 1 when a *VSX Scalar Compare Double-Precision*, *VSX Vector Compare Double-Precision*, or *VSX Vector Compare Single-Precision* class instruction causes an Invalid Compare type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
45	**Floating-Point Fraction Rounded** (FR) This bit is set to 0 or 1 by *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar Integer Conversion*, and *VSX Scalar Round to Floating-Point Integer* class instructions to indicate whether or not the fraction was incremented during rounding. See Section 7.3.2.6 , “Rounding” on page 335. This bit is not sticky.
46	**Floating-Point Fraction Inexact** (FI) This bit is set to 0 or 1 by *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar Integer Conversion*, and *VSX Scalar Round to Floating-Point Integer* class instructions to indicate whether or not the rounded result is inexact or the instruction caused a disabled Overflow exception. See Section 7.3.2.6 on page 335. This bit is not sticky. See the definition of XX, above, regarding the relationship between FI and XX.

Bits	Definition
47:51	**Floating-Point Result Flags** (FPRF) *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined. For *VSX Scalar Convert Double-Precision to Integer* class instructions, the value placed into FPRF is undefined. Additional details are as follows.
47	**Floating-Point Result Class Descriptor** (C) *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set this bit with the FPCC bits, to indicate the class of the result as shown in Table 2, "Floating-Point Result Flags," on page 327.
48:51	**Floating-Point Condition Code** (FPCC) *VSX Scalar Compare Double-Precision* instruction sets one of the FPCC bits to 1 and the other three FPCC bits to 0 based on the relative values of the operands being compared. *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set the FPCC bits with the C bit, to indicate the class of the result as shown in Table 2, "Floating-Point Result Flags," on page 327. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.
48	**Floating-Point Less Than or Negative** (FL)
49	**Floating-Point Greater Than or Positive** (FG)
50	**Floating-Point Equal or Zero** (FE)
51	**Floating-Point Unordered or NaN** (FU)

Bits	Definition
52	Reserved
53	**Floating-Point Invalid Operation Exception (Software-Defined Condition)** (VXSOFT) This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 343. **Programming Note** VXSOFT can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.
54	**Floating-Point Invalid Operation Exception (Invalid Square Root)** (VXSQRT) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* or *VSX Vector Floating-Point Arithmetic* class instruction causes a Invalid Square Root type Invalid Operation exception. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
55	**Floating-Point Invalid Operation Exception (Invalid Integer Convert)** (VXCVI) This bit is set to 1 when a *VSX Scalar Convert Double-Precision to Integer*, *VSX Vector Convert Double-Precision to Integer*, or *VSX Vector Convert Single-Precision to Integer* class instruction causes a Invalid Integer Convert type Invalid Operation exception. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 343. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
56	**Floating-Point Invalid Operation Exception Enable** (VE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Invalid Operation exceptions. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 343.

Bits	Definition
57	**Floating-Point Overflow Exception Enable** (OE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Overflow exceptions. See Section 7.4.3 , "Floating-Point Overflow Exception" on page 351.
58	**Floating-Point Underflow Exception Enable** (UE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Underflow exceptions. See Section 7.4.4 , "Floating-Point Underflow Exception" on page 353.
59	**Floating-Point Zero Divide Exception Enable** (ZE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Zero Divide exceptions. See Section 7.4.2 , "Floating-Point Zero Divide Exception" on page 349.
60	**Floating-Point Inexact Exception Enable** (XE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Inexact exceptions. See Section 7.4.5 , "Floating-Point Inexact Exception" on page 356.
61	**Floating-Point Non-IEEE Mode** (NI) Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved, and the remainder of the definition of this bit does not apply. If floating-point non-IEEE mode is implemented, this bit has the following meaning. 0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard). 1 The processor is in floating-point non-IEEE mode.

Bits	Definition
61	**Floating-Point Non-IEEE Mode** (NI) (*continued*) When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits is permitted to have meanings different from those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a given floating-point instruction with NI=1, and any additional requirements for using non-IEEE mode, are implementation-dependent. The results of executing a given instruction in non-IEEE mode is permitted to vary between implementations, and between different executions on the same implementation. **Programming Note** When the processor is in floating-point non-IEEE mode, the results of floating-point operations is permitted to be approximate, and performance for these operations might be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation is permitted to return 0 instead of a denormalized number and return a large number instead of an infinity.
62:63	**Floating-Point Rounding Control** (RN) This field is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions that round their result and the rounding mode is not implied by the opcode. This bit can be explicitly set or reset by a new Move To FPSCR class instruction. See Section 7.3.2.6 , "Rounding" on page 335. 00 Round to Nearest Even 01 Round toward Zero 10 Round toward +Infinity 11 Round toward -Infinity

Result Flags					Result Value Class
C	FL	FG	FE	FU	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	- Infinity
0	1	0	0	0	- Normalized Number
1	1	0	0	0	- Denormalized Number
1	0	0	1	0	- Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Denormalized Number
0	0	1	0	0	+ Normalized Number
0	0	1	0	1	+ Infinity

Table 2. Floating-Point Result Flags

7.3 VSX Operations

7.3.1 VSX Floating-Point Arithmetic Overview

This section describes the floating-point arithmetic and exception model supported by category Vector-Scalar Extension. Except for extensions to support 32-bit single-precision floating-point vector operations, the models are identical to that described in Chapter 4. Floating-Point Facility [Category: Floating-Point].

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic* (hereafter referred to as *the IEEE standard*). That standard defines certain required "operations" (addition, subtraction, and so on). Herein, the term, floating-point operation, is used to refer to one of these required operations and to additional operations defined (e.g., those performed by *Multiply-Add* or *Reciprocal Estimate* instructions). A Non-IEEE mode is also provided. This mode, which is permitted to produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in VSRs; to move floating-point data between storage and these registers.

These instructions are divided into two categories.

– computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. There are two forms of computational instructions, scalar, which perform a single floating-point operation, and vector, which perform either two double-precision floating-point operations or four single-precision operations. Computational instructions place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 7.6.1.3 through 7.6.1.7.2.

– noncomputational instructions

  The noncomputational instructions are those that perform loads and stores, move the contents of a VSR to another floating-point register possibly altering the sign, and select the value from one of two VSRs based on the value in a third VSR. The operations performed by these instructions are not considered floating-point operations. These instructions do not alter the Floating-Point Status and Control Register. They are the instructions listed in Sections 7.6.1.1, 7.6.1.2.1, and 7.6.1.8 through 7.6.1.9.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values that are “Not a Number” (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. NaNs might be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to categories Vector-Scalar Extension and Floating-Point: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the FPSCR. They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

– Invalid Operation exception (VX)
  - SNaN (VXSNAN)
  - Infinity - Infinity (VXISI)
  - Infinity÷Infinity (VXIDI)
  - Zero÷Zero (VXZDZ)
  - Infinity×Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)
  - Invalid Integer Convert (VXCVI)

– Zero Divide exception (ZX)
– Overflow exception (OX)
– Underflow exception (UX)
– Inexact exception (XX)

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 323 for a description of these exception and enable bits, and Section 7.3.3 , “VSX Floating-Point Execution Models” on page 337 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

7.3.2 VSX Floating-Point Data

7.3.2.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats, 32-bit single-precision format and 64-bit double-precision format. The single-precision format is used for SP data in storage and registers. The double-precision format is used for DP data in storage and registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single-precision and double-precision formats is shown below.

Values in floating-point format are composed of three fields:

S	sign bit
EXP	exponent+bias
FRACTION	fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (that is, the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Table 3.

Figure 113. Floating-point single-precision format

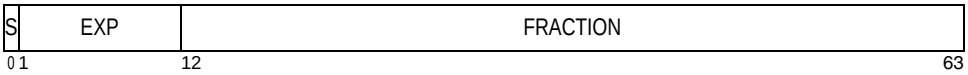


Figure 114.Floating-point double-precision format

	Single-Precision Format	Double-Precision Format
Exponent Bias	+127	+1023
Maximum Exponent (Emax)	+127	+1023
Minimum Exponent (Emin)	-126	-1022
Widths (bits):Format Sign Exponent Fraction Significand	32 1 8 23 24	64 1 11 52 53
Nmax	$(1-2^{-24}) \times 2^{128} \approx 3.4 \times 10^{38}$	$(1-2^{-53}) \times 2^{1024} \approx 1.8 \times 10^{308}$
Nmin	$1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$	$1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308}$
Dmin	$1.0 \times 2^{-149} \approx 1.4 \times 10^{-45}$	$1.0 \times 2^{-1074} \approx 4.9 \times 10^{-324}$

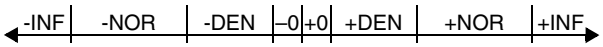
≈ Value is approximate
DminSmallest (in magnitude) representable denormalized number.
NmaxLargest (in magnitude) representable number.
NminSmallest (in magnitude) representable normalized number.

Table 3. IEEE floating-point fields

7.3.2.2 Value Representation

This architecture defines numeric and nonnumeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The nonnumeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 115.

Figure 115.Approximation to real numbers



The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers

Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (±NOR)

These are values that have a biased exponent value in the range:

1 to 254 in single-precision format
1 to 2046 in double-precision format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$$NOR = (-1)^s \times 2^E \times (1.fraction)$$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

Zero values (±0)

These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to -0).

Denormalized numbers (±DEN)

These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

$$DEN = (-1)^s \times 2^{Emin} \times (0.fraction)$$

where Emin is the minimum representable exponent value (-126 for single-precision, -1022 for double-precision).

Infinities (±INF)

These are values that have the maximum biased exponent value:

255 in single-precision format
2047 in double-precision format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

-Infinity < every finite number < +Infinity

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 343.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.

Not a Numbers (NaNs)

These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0, the NaN is a *Signaling NaN*; otherwise it is a *Quiet NaN*.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation exception is disabled (VE=0). Quiet NaNs propagate through all floating-point operations except ordered comparison and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

Assume the following generic arithmetic templates.

```
f(src1,src3,src2)
  ex: result = (src1 x src3) - src2

f(src1,src2)
  ex: result = src1 x src2
  ex: result = src1 + src2

f(src1)
  ex: result = f(src1)
```

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a trap-disabled Invalid Operation exception, the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if src1 is a NaN
  then result = Quiet(src1)
  else if src2 is a NaN (if there is a src2)
    then result = Quiet(src2)
    else if src3 is a NaN (if there is a src3)
       then result = Quiet(src3)
       else if disabled invalid operation exception
          then result = generated QNaN
```

where Quiet(x) means x if x is a QNaN and x converted to a QNaN if x is an SNaN. Any instruction that generates a QNaN as the result of a disabled Invalid Operation exception generates the value 0x7FF8_0000_0000_0000 for double-precision and 0x7FC0_0000 for single-precision.

Note that the M-form multiply-add-type instructions use the B source operand to specify src3 and the T target operand to specify src2, whereas A-form multiply-add-type instructions use the B source operand to specify src2 and the T target operand to specify src3.

A double-precision NaN is considered to be representable in single-precision format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

7.3.2.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

– The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same signs, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation x-y is the same as the sign of the result of the add operation x+(-y).

  When the sum of two operands with opposite sign, or the difference of two operands with the same signs, is exactly zero, the sign of the result is positive in all rounding modes except Round toward -Infinity, in which mode the sign is negative.

– The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

– The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is -Infinity.

– The sign of the result of a *Convert From Integer* or *Round to Floating-Point Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

7.3.2.4 Normalization and Denormalization

The intermediate result of an arithmetic instruction can require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the

significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 7.3.3.1, "VSX Execution Model for IEEE Operations" on page 337) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result can have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be "Tiny" and the stored result is determined by the rules described in Section 7.4.4 , "Floating-Point Underflow Exception" on page 353. These rules can require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process, "Loss of Accuracy" has occurred (See Section 7.4.4 , "Floating-Point Underflow Exception" on page 353) and Underflow exception is signaled.

Engineering Note

When denormalized numbers are operands of multiply, divide, and square root operations, some implementations might prenormalize the operands internally before performing the operations.

7.3.2.5 Data Handling and Precision

Scalar double-precision floating-point data is represented in double-precision format in VSRs and storage.

Vector double-precision floating-point data is represented in double-precision format in VSRs and storage.

Scalar single-precision floating-point data is represented in double-precision format in VSRs and in single-precision format in storage.

Vector single-precision floating-point data is represented in single-precision format in VSRs and storage.

Double-precision operands may be used as input for double-precision scalar arithmetic operations.

Double-precision operands may be used as input for single-precision scalar arithmetic operations when trapping on overflow and underflow exceptions is disabled.

Single-precision operands may be used as input for double-precision and single-precision scalar arithmetic operations.

Double-precision operands may be used as input for double-precision vector arithmetic operations.

Single-precision operands may be used as input for single-precison vector arithmetic operations.

Instructions are also provided for manipulations which do not require double-precision or single-precision. In addition, instructions are provided to access an integer representation in GPRs.

Single-Precision Operands

For single-precision scalar data, a conversion from single-precision format to double-precision format is performed when loading from storage into a VSR and a conversion from double-precision format to single-precision format is performed when storing from a VSR to storage. No floating-point exceptions are caused by these instructions.

Instructions are provided to convert between single-precision and double-precision formats for scalar and vector data in VSRs.

An instruction is provided to explicitly convert a double format operand in a VSR to single-precision. Scalar single-precision floating-point is enabled with six types of instruction.

1. Load Scalar Single-Precision

   This form of instruction accesses a floating-point operand in single-precision format in storage, converts it to double-precision format, and loads it into a VSR. No floating-point exceptions are caused by these instructions.

2. *Scalar Round to Single-Precision*

   xsrsp rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into a VSR in double-precision format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of

xsrsp, ***xsrsp*** does not alter the value. Values greater in magnitude than 2319 when Overflow is enabled (OE=1) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than 2-318 when Underflow is enabled (UE=1) produce undefined results because the value cannot be scaled back into the normalized range.

3. Scalar Convert Single-Precision to Double-Precision

   xscvspdp accesses a floating-point operand in single-precision format from word element 0 of the source VSR, converts it to double-precision format, and places it into doubleword element 0 of the target VSR.

4. Scalar Convert Double-Precision to Single-Precision

   xscvdpsp rounds the double-precision floating-point value in doubleword element 0 of the source VSR to single-precision, and places the result into word element 0 of the target VSR in single-precision format. This function would be used to port scalar floating-point data to a format compatible for single-precision vector operations. Values greater in magnitude than 2319 when Overflow is enabled (OE=1) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than 2-318 when Underflow is enabled (UE=1) produce undefined results because the value cannot be scaled back into the normalized range.

5. VSX Scalar Single-Precision Arithmetic

   This form of instruction takes operands from the VSRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single-precision format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then placed into the target VSR in double-precision format. The result lies in the range supported by the single format.

   If any input value is not representable in single-precision format and either OE=1 or UE=1, the result placed into the target VSR and the setting of status bits in the FPSCR are undefined.

   For ***xsresp*** or ***xsrsqrtesp***, if the input value is finite and has an unbiased exponent greater than +127, the input value is interpreted as an Infinity.

6. Store VSX Scalar Single-Precision

   stxsspx converts a single-precision value that is in double-precision format to single-precision format and stores that operand into storage. No floating-point exceptions are caused by ***stxsspx***. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding five types.)

When the result of a *Load VSX Scalar Single-Precision* (***lxsspx***), a *VSX Scalar Round to Single-Precision* (***xsrsp***), or a *VSX Scalar Single-Precision Arithmetic*[1] instruction is stored in a VSR, the low-order 29 bits of FRACTION are zero.

Programming Note

VSX Scalar Round to Single-Precision (***xsrsp***) is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. ***xsrsp*** should be used to convert double-precision floating-point values to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by an ***xsrsp***.

1. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***

Programming Note

A single-precision value can be used in double-precision scalar arithmetic operations.

Except for ***xsresp*** or ***xsrsqrtesp***, any double-precision value can be used in single-precision scalar arithmetic operations when `OE=0` and `UE=0`. When `OE=1` or `UE=1`, or if the instruction is ***xsresp*** or ***xsrsqrtesp***, source operands must be respresentable in single-precision format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and integer processing, instructions are provided to convert between floating-point double and single-precision format and integer word and doubleword format in a `VSR`. Computation on integer-valued operands can be performed using arithmetic instructions of the required precision. (The results might not be integer values.) The three groups of instructions provided specifically to support integer-valued operands are described below.

1. Rounding to a floating-point integer

   VSX Scalar Round to Double-Precision Integer[1] instructions round a double-precision operand to an integer value in double-precision format.

   VSX Vector Round to Double-Precision Integer[2] instructions round each double-precision vector operand element to an integer value in double-precision format.

   VSX Vector Round to Single-Precision Integer[3] instructions round each single-precision vector operand element to an integer value in single-precision format.

   Except for ***xsrdpic***, ***xvrdpic***, and ***xvrspic***, rounding is performed using the rounding mode specified by the opcode. For ***xsrdpic***, ***xvrdpic***, and ***xvrspic***, rounding is performed using the rounding mode specified by `RN`.

   VSX Round to Floating-Point Integer[4] instructions can cause Invalid Operation (VXSNAN) exceptions.

   xsrdpic, ***xvrdpic***, and ***xvrspic*** can also cause Inexact exception.

   See Sections 7.3.2.6 and 7.3.3.1 for more information about rounding.

2. Converting floating-point format to integer format

   VSX Scalar Double-Precision to Integer Format Conversion[5] instructions convert a double-precision operand to 32-bit or 64-bit signed or unsigned integer format.

   VSX Vector Double-Precision to Integer Format Conversion[6] instructions convert either double-precision or single-precision vector operand elements to 32-bit or 64-bit signed or unsigned integer format.

   VSX Vector Single-Precision to Integer Doubleword Format Conversion[7] instructions converts the single-precision value in each odd-numbered word element of the source vector operand to a 64-bit signed or unsigned integer format.

   VSX Vector Single-Precision to Integer Word Format Conversion[8] instructions converts the single-precision value in each word element of the source vector operand to either a 32-bit signed or unsigned integer format.

1. *VSX Scalar Round to Double-Precision Integer* instructions:
   xsrdpi, ***xsrdpip***, ***xsrdpim***, ***xsrdpiz***, ***xsrdpic***
2. *VSX Vector Round to Double-Precision Integer* instructions:
   xvrdpi, ***xvrdpip***, ***xvrdpim***, ***xvrdpiz***, ***xvrdpic***
3. *VSX Vector Round to Single-Precision Integer* instructions:
   xvrspi, ***xvrspip***, ***xvrspim***, ***xvrspiz***, ***xvrspic***
4. *VSX Round to Floating-Point Integer* instructions:
   xsrdpi, ***xsrdpip***, ***xsrdpim***, ***xsrdpiz***, ***xsrdpic***, ***xvrdpi***, ***xvrdpip***, ***xvrdpim***, ***xvrdpiz***, ***xvrdpic***, ***xvrspi***, ***xvrspip***, ***xvrspim***, ***xvrspiz***, and ***xvrspic***
5. *VSX Scalar Double-Precision to Integer Format Conversion* instructions:
   xscvdpsxds, ***xscvdpsxws***, ***xscvdpuxds***, ***xscvdpuxws***
6. *VSX Vector Double-Precision to Integer Format Conversion* instructions:
   xvcvdpsxds, ***xvcvdpsxws***, ***xvcvdpuxds***, ***xvcvdpuxws***
7. *VSX Vector Single-Precision to Integer Doubleword Format Conversion* instructions:
   xvcvspsxds, ***xvcvspuxds***
8. *VSX Vector Single-Precision to Integer Word Format Conversion* instructions:
   xvcvspsxws, ***xvcvspuxws***

Rounding is performed using Round Towards Zero rounding mode. These instructions can cause Invalid Operation (VXSNAN, VXCVI) and Inexact exceptions.

3. Converting integer format to floating-point format

   VSX Scalar Integer Doubleword to Double-Precision Format Conversion[1] instructions convert a 64-bit signed or unsigned integer to a double-precision floating-point value and returns the result in double-precision format.

   VSX Scalar Integer Doubleword to Single-Precision Format Conversion[2] instructions converts a 64-bit signed or unsigned integer to a single-precision floating-point value and returns the result in double-precision format.

   VSX Vector Integer Doubleword to Double-Precision Format Conversion[3] instructions converts the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to double-precision floating-point format.

   VSX Vector Integer Word to Double-Precision Format Conversion[4] instructions converts the 32-bit signed or unsigned integer in each odd-numbered word element in the source vector operand to double-precision floating-point format.

   VSX Vector Integer Doubleword to Single-Precision Format Conversion[5] instructions convert the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to single-precision floating-point format.

   VSX Vector Integer Word to Single-Precision Format Conversion[6] instructions convert the 32-bit signed or unsigned integer in each word element in the source vector operand to single-precision floating-point format.

   Rounding is performed using the rounding mode specificed in RN. Because of the limitations of the source format, only an Inexact exception can be generated.

7.3.2.6 Rounding

The material in this section applies to operations that have numeric operands (that is, operands that are not infinities or NaNs). Rounding the intermediate result of such an operation can cause an Overflow exception, an Underflow exception, or an Inexact exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 7.3.2.2, “Value Representation” and Section 7.4, “VSX Floating-Point Exceptions” for the cases not covered here.

The floating-point arithmetic, and rounding and conversion instructions round their intermediate results. With the exception of the estimate instructions, these instructions produce an intermediate result that can be regarded as having unbounded precision and exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target element of the target VSR in either double or single-precision format.

The scalar round to double-precision integer, vector round to double-precision integer, and convert double-precision to integer instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for round to double-precision integer instructions is normalized and put in double-precision format, and, for the convert double-precision to integer instructions, is converted to a signed or unsigned integer.

The vector round to single-precision integer and vector convert single-precision to integer instructions with biased exponents ranging from 126 through 178 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 179. (Intermediate results with biased exponents 179 or larger are already integers, and with biased exponents 125 or less round to zero.) After rounding, the final result for vector round to single-precision integer is normalized and put in double-precision format, and for

1. *VSX Scalar Integer Doubleword to Double-Precision Format Conversion* instructions: ***xscvsxddp***, ***xscvuxddp***
2. *VSX Scalar Integer Doubleword to Single-Precision Format Conversion* instructions: ***xscvsxdsp***, ***xscvuxdsp***
3. *VSX Vector Integer Doubleword to Double-Precision Format Conversion* instructions: ***xscvsxddp***, ***xscvuxddp***
4. *VSX Vector Integer Word to Double-Precision Format Conversion* instructions: ***xscvsxwdp***, ***xscvuxwdp***
5. *VSX Vector Integer Doubleword to Single-Precision Format Conversion* instructions: ***xscvsxdsp***, ***xscvuxdsp***
6. *VSX Vector Integer Word to Single-Precision Format Conversion* instructions: ***xscvsxwsp***, ***xscvuxwsp***

vector convert single-precision to integer is converted to a signed or unsigned integer.

FR and FI generally indicate the results of rounding. Each of the scalar instructions which rounds its intermediate result sets these bits. There are no vector instructions that modify FR and FI. If the fraction is incremented during rounding, FR is set to 1, otherwise FR is set to 0. If the result is inexact, FI is set to 1, otherwise FI is set to zero. The scalar round to double-precision integer instructions are exceptions to this rule, setting FR and FI to 0. The scalar double-precision estimate instructions set FR and FI to undefined values. The remaining scalar floating-point instructions do not alter FR and FI.

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 323. These are encoded as follows.

RN	Rounding Mode
00	Round to Nearest Even
01	Round towards Zero
10	Round towards +Infinity
11	Round towards -Infinity

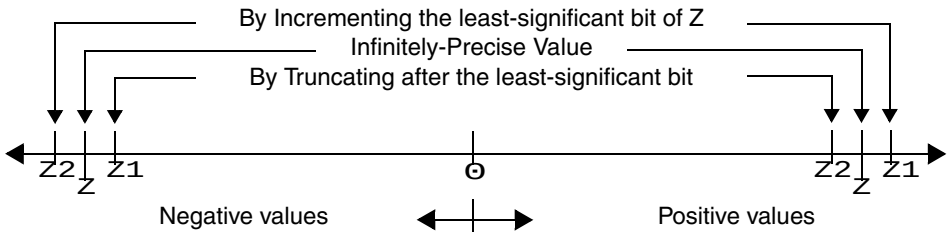
A fifth rounding mode is provided in the round to floating-point integer instructions (Section 7.6.1.7.2 on page 368), Round to Nearest Away.

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 116 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes.

See Section 7.3.3.1, “VSX Execution Model for IEEE Operations” on page 337 for a detailed explanation of rounding.

Figure 116 also summarizes the rounding actions for floating-point intermediate result for all supported rounding modes.



Round to Nearest Away
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is furthest away from 0.

Round to Nearest Even
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit is 0).

Round toward Zero
Choose the smaller in magnitude (Z1 or Z2).

Round toward +Infinity
Choose Z1.

Round toward -Infinity
Choose Z2.

Figure 116.Selection of Z1 and Z2

7.3.3 VSX Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (that is, operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 7.3.2.2 and Section 7.3.3 for the cases not covered here.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow and underflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

– Underflow during multiplication using a denormalized operand.

– Overflow during division using a denormalized divisor.

– Undeflow during division using denormalized dividend and a large divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands.

VSX defines both scalar and vector double-precision floating-point operations to operate only on double-precision operands. VSX also defines vector single-precision floating-point operations to operate only on single-precision operands.

7.3.3.1 VSX Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

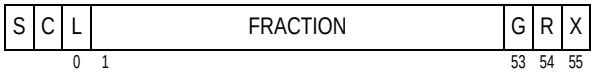


Figure 117.IEEE floating-point execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that appear to the low-order side of the R bit, resulting from either shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Table 4 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	IR closer to NL
0	1	0	
0	1	1	
1	0	0	IR midway between NL and NH
1	0	1	IR closer to NH
1	1	0	
1	1	1	

Table 4. Interpretation of G, R, and X bits

Table 5 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision

floating-point numbers relative to the accumulator illustrated in Figure 117.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of bits 26:52, G, R, X

Table 5. Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction.

Four user-selectable rounding modes are provided through RN as described in Section 7.3.2.6, “Rounding” on page 335. The rules for rounding in each mode are as follows.

– ***Round to Nearest Even***

Guard bit = 0
The result is truncated.

Guard bit = 1
Depends on Round and Sticky bits:

Case a
If the Round or Sticky bit is 1 (inclusive), the result is incremented.

Case b
If the Round and Sticky bits are 0 (result midway between closest representable values), if the low-order bit of the result is 1, the result is incremented. Otherwise (the low-order bit of the result is 0), the result is truncated. This is the case of a tie rounded to even.

– ***Round toward Zero***
Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.
The result is truncated.

– ***Round toward +Infinity***
If positive, the result is incremented.
If negative, the result is truncated.

– ***Round toward -Infinity***
If positive, the result is truncated.
If negative, the result is incremented.

A fifth rounding mode is provided in the *VSX Round to Floating-Point Integer* instructions (Section 7.6.1.7.2 on page 368) with the rules for rounding as follows.

– ***Round to Nearest Away***

Guard bit = 0
The result is truncated.

Guard bit = 1
The result is incremented.

If any of the Guard, Round, or Sticky bits is nonzero, the result is also inexact.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. Fraction bits are stored to the target VSR.

7.3.3.2 VSX Execution Model for Multiply-Add Type Instructions

This architecture provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar, except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

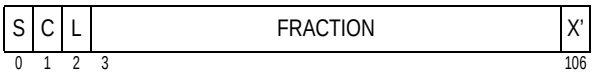


Figure 118.Multiply-add 64-bit execution model

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input’s exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X’ bit. The add

operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 6 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X'
Single	24	25	OR of 26:105, X'

Table 6. Location of the Guard, Round, and Sticky bits in the multiply-add execution model

The rules for rounding the intermediate result are the same as those given in Section 7.3.3.1.

If the instruction is a negative multiply-add or negative multiply-subtract type instruction, the final result is negated.

7.4 VSX Floating-Point Exceptions

This architecture defines the following floating-point exceptions under the IEEE-754 exception model:

– Invalid Operation exception

  SNaN
  Infinity - Infinity
  Infinity÷Infinity
  Zero÷Zero
  Infinity×Zero
  Invalid Compare
  Software-Defined Condition
  Invalid Square Root
  Invalid Integer Convert

– Zero Divide exception
– Overflow exception
– Underflow exception
– Inexact exception

These exceptions, other than Invalid Operation exception resulting from a Software-Defined Condition, can occur during execution of computational instructions. An Invalid Operation exception resulting from a Software-Defined Condition occurs when a *Move To FPSCR* instruction sets VXSOFT to 1.

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 341), whether and how the system floating-point enabled exception error handler is invoked. In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow exception depends on the setting of the enable bit.

A single instruction, other than ***mtfsfi*** or ***mtfsf***, can set more than one exception bit only in the following cases:

– An Inexact exception can be set with an Overflow exception.

– An Inexact exception can be set with an Underflow exception.

– An Invalid Operation exception (SNaN) is set with an Invalid Operation exception (Infinity×0) for multiply-add class instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.

– An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Compare) for ordered comparison instructions.

– An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Integer Convert) for convert to integer instructions.

When an exception occurs, the writing of a result to the target register can be suppressed, or a result can be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the certain kinds of exceptions, based on whether the instruction is a vector or a scalar instruction, so that there is no possibility that one of the operands is lost. For other kinds of exceptions and also depending on whether the instruction is a vector or a scalar instruction, a result is generated and written to the destination specified by the instruction causing the exception. The result can be a different value for the enabled and disabled conditions for some of these exceptions. Table 7 lists the types of exceptions and indicates whether a result is written to the target VSR or suppressed.

On exception type...	Scalar Instruction Results	Vector Instruction Results
Enabled Invalid Operation	suppressed	suppressed
Enabled Zero Divide	suppressed	suppressed
Enabled Overflow	written	suppressed
Enabled Underflow	written	suppressed
Enabled Inexact	written	suppressed
Disabled Invalid Operation	written	written

Table 7. Exception Types Result Suppression

On exception type...	Scalar Instruction Results	Vector Instruction Results
Disabled Zero Divide	written	written
Disabled Overflow	written	written
Disabled Underflow	written	written
Disabled Inexact	written	written

Table 7. Exception Types Result Suppression

The subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of *traps* and *trap handlers*. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the trap enabled case; the expectation is that the exception is detected by software, which revises the result. An FPSCR exception enable bit of 0 causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case. The expectation is that the exception is not detected by software, which uses the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is required for all exceptions, all FPSCR exception enable bits must be set to 0, and Ignore Exceptions Mode (see below) should be used. In this case, the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits, if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1, and a mode other than Ignore Exceptions Mode must be used. In this case, the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1. The *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception. The effects of the four possible settings of these bits are as follows.

FE0 FE1 Description

0 0 **Ignore Exceptions Mode**
Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.

0 1 **Imprecise Nonrecoverable Mode**
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction might have been used by or might have affected subsequent instructions that are executed before the error handler is invoked.

1 0 **Imprecise Recoverable Mode**
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler for it to identify the excepting instruction, the operands, and correct the result. No results produced by the excepting instruction have been used by or affected subsequent instructions that are executed before the error handler is invoked.

1 1 **Precise Mode**
The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

Architecture Note

The FE0 and FE1 bits must be defined in Book III in a manner such that they can be changed dynamically and can easily be treated as part of a process' state.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system

floating-point enabled exception error handler is invoked have been completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction, and there is only one such instruction. Otherwise, it has not begun execution, or has been partially executed in some cases, as described in Book III.

Programming Note

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, because of instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In both Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler that result from instructions initiated before the *Floating-Point Status and Control Register* instruction to occur. This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. It always applies in the latter case.

To obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

– If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.

– If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.

– Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.

– Precise Mode can degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

Engineering Note

It is permissible for the implementation to be precise in any of the three modes that permit interrupts, or to be recoverable in Nonrecoverable Mode.

7.4.1 Floating-Point Invalid Operation Exception

7.4.1.1 Definition

An Invalid Operation exception occurs when an operand is invalid for the specified operation. The invalid operations are:

SNaN
Any floating-point operation on a Signaling NaN.

Infinity–Infinity
Magnitude subtraction of infinities.

Infinity÷Infinity
Floating-point division of infinity by infinity.

Zero÷Zero
Floating-point division of zero by zero.

Infinity × Zero
Floating-point multiplication of infinity by zero.

Invalid Compare
Floating-point ordered comparison involving a NaN.

Invalid Square Root
Floating-point square root or reciprocal square root of a nonzero negative number.

Invalid Integer Convert
Floating-point-to-integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN.

An Invalid Operation exception also occurs when an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets VXSOFT to 1 (Software-Defined Condition).

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

7.4.1.2 Action for VE=1

When Invalid Operation exception is enabled (VE=1) and an Invalid Operation exception occurs, the following actions are taken:

For *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Floating-Point to Integer*, and *VSX Scalar Round to Floating-Point Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity–Infinity)
VXIDI	(if Infinity÷Infinity)
VXZDZ	(if Zero÷Zero)
VXIMZ	(if Infinity×Zero)
VXSQRT	(if Invalid Square Root)
VXCVI	(if Invalid Integer Convert)

2. Update of VSR[XT] is suppressed.

3. FR and FI are set to zero.

4. FPRF is unchanged.

For *VSX Scalar Floating-Point Compare* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXVC	(if Invalid Compare)

2. FR, FI, and C are unchanged.

3. FPCC is set to reflect unordered.

For *VSX Vector Floating-Point Arithmetic*, *VSX Vector Floating-Point Compare*, *VSX Vector DP-SP Conversion*, *VSX Vector Convert Floating-Point to Integer*, and *VSX Vector Round to Floating-Point Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity – Infinity)
VXIDI	(if Infinity ÷ Infinity)
VXZDZ	(if Zero ÷ Zero)
VXIMZ	(if Infinity × Zero)
VXVC	(if Invalid Compare)
VXSQRT	(if Invalid Square Root)
VXCVI	(if Invalid Integer Convert)

2. Update of VSR[XT] is suppressed for all vector elements.

3. FR and FI are unchanged.

4. FPRF is unchanged.

7.4.1.3 Action for VE=0

When Invalid Operation exception is disabled (VE=0) and an Invalid Operation exception occurs, the following actions are taken:

For the *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***) instruction:

1. VXSNAN is set to 1.

2. The single-precision representation of a Quiet NaN is placed into word element 0 of VSR[XT]. The contents of word elements 1-3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is set to indicate the class of the result (Quiet NaN).

For the *VSX Vector Single-Precision Arithmetic* instructions, *VSX Vector Single-Precision Maximum/Minimum* instructions, the *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***) instruction, and the *VSX Vector Round to Single-Precision Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity – Infinity)
VXIDI	(if Infinity ÷ Infinity)
VXZDZ	(if Zero ÷ Zero)
VXIMZ	(if Infinity × Zero)
VXSQRT	(if Invalid Square Root)

2. The single-precision representation of a Quiet NaN is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the *VSX Scalar Double-Precision Arithmetic* instructions, *VSX Scalar Double-Precision Maximum/Minimum* instructions, the *VSX Scalar Convert Single-Precision to Double-Precision format* (***xscvspdp***) instruction, and the *VSX Scalar Round to Double-Precision Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity – Infinity)
VXIDI	(if Infinity ÷ Infinity)
VXZDZ	(if Zero ÷ Zero)
VXIMZ	(if Infinity × Zero)
VXSQRT	(if Invalid Square Root)

2. The double-precision representation of a Quiet NaN is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is set to indicate the class of the result (Quiet NaN).

For the *VSX Vector Double-Precision Arithmetic* instructions, *VSX Vector Double-Precision Maximum/Minimum* instructions, the *VSX Vector Convert Single-Precision to Double-Precision format* (***xvcvspdp***) instruction, and the *VSX Vector Round to Double-Precision Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity – Infinity)
VXIDI	(if Infinity ÷ Infinity)
VXZDZ	(if Zero ÷ Zero)
VXIMZ	(if Infinity × Zero)
VXSQRT	(if Invalid Square Root)

2. The double-precision representation of a Quiet NaN is placed into its respective doubleword element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the *VSX Scalar Convert Double-Precision to Signed Integer Doubleword* (***xscvdpsxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Scalar Convert Double-Precision to Unsigned Integer Doubleword* (***xscvdpuxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Scalar Convert Double-Precision to Signed Integer Word* (***xscvdpsxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Scalar Convert Double-Precision to Unsigned Integer Word* (***xscvdpuxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Vector Convert Double-Precision to Signed Integer Doubleword* (***xvcvdpsxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the double-precision operand in the corresponding doubleword element of VSR[XB] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into its respective doubleword element i of VSR[XT] if the double-precision operand in the corresponding doubleword element of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert Double-Precision to Unsigned Integer Doubleword* (***xvcvdpuxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert Double-Precision to Signed Integer Word* (***xvcvdpsxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed intoword element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element i×2+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert Double-Precision to Unsigned Integer Word* (***xvcvdpuxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   0x0000_0000 is placed into word element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element i×2+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert Single-Precision to Signed Integer Doubleword* (***xvcvspsxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert Single-Precision to Unsigned Integer Doubleword* (***xvcvspuxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert Single-Precision to Signed Integer Word* (***xvcvspsxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed into word element i of VSR[XT] if the single-precision operand in word element i of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element i of VSR[XT] if the single-precision operand in word element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element i×2+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert Single-Precision to Unsigned Integer Word* (***xvcvspuxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element i of VSR[XT] if the single-precision operand in the corresponding word element i×2 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000 is placed into word element i of VSR[XT] if the single-precision operand in word element i i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element i×2+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Scalar Floating-Point Compare* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. FR, FI and C are unchanged.

3. FPCC is set to reflect unordered.

For the VSX *Vector Compare Single-Precision* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x0000_0000 is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the vector double-precision compare instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x0000_0000_0000_0000 is placed into its respective doubleword element of VSR[XT].

3. FR, FI, and FPRF are not modified.

7.4.2 Floating-Point Zero Divide Exception

7.4.2.1 Definition

A Zero Divide exception occurs when a *VSX Floating-Point Divide*[1] instruction is executed with a zero divisor value and a finite nonzero dividend value.

A Zero Divide exception also occurs when a *VSX Floating-Point Reciprocal Estimate*[2] instruction or a *VSX Floating-Point Reciprocal Square Root Estimate*[3] instruction is executed with an operand value of zero.

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

7.4.2.2 Action for ZE=1

When Zero Divide exception is enabled (ZE=1) and a Zero Divide exception occurs, the following actions are taken:

For *VSX Scalar Floating-Point Divide*[4] instructions, *VSX Scalar Floating-Point Reciprocal Estimate*[5] instructions, and *VSX Scalar Floating-Point Reciprocal Square Root Estimate*[6] instructions, do the following.

1. ZX is set to 1.
2. Update of VSR[XT] is suppressed.
3. FR and FI are set to 0.
4. FPRF is unchanged.

For *VSX Vector Floating-Point Divide*[7] instructions, *VSX Vector Floating-Point Reciprocal Estimate*[8] instructions, and *VSX Vector Floating-Point Reciprocal Square Root Estimate*[9] instructions, do the following.

1. ZX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR and FI are unchanged.
4. FPRF is unchanged.

1. *VSX Floating-Point Divide* instructions:
   xsdivdp, ***xsdivsp***, ***xvdivdp***, ***xvdivsp***
2. *VSX Floating-Point Reciprocal Estimate* instructions:
   xsredp, ***xsresp***, ***xvredp***, ***xvresp***
3. *VSX Floating-Point Reciprocal Square Root Estimate* instructions:
   xsrsqrtedp, ***xsrsqrtesp***, ***xvrsqrtedp***, ***xvrsqrtesp***
4. *VSX Scalar Floating-Point Divide* instructions:
   xsdivdp, ***xsdivsp***
5. *VSX Scalar Floating-Point Reciprocal Estimate* instructions:
   xsredp, ***xsresp***
6. *VSX Scalar Floating-Point Reciprocal Square Root Estimate* instructions:
   xsrsqrtedp, ***xsrsqrtesp***
7. *VSX Vector Floating-Point Divide* instructions:
   xvdivdp, ***xvdivsp***
8. *VSX Vector Floating-Point Reciprocal Estimate* instructions:
   xvredp, ***xvresp***
9. *VSX Vector Floating-Point Reciprocal Square Root Estimate* instructions:
   xvrsqrtedp, ***xvrsqrtesp***

7.4.2.3 Action for ZE=0

When Zero Divide exception is disabled (ZE=0) and a Zero Divide exception occurs, the following actions are taken:

For *VSX Scalar Floating-Point Divide*[1] instructions, do the following.

1. ZX is set to 1.
2. An Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR and FI are set to 0.
4. FPRF is set to indicate the class and sign of the result (± Infinity).

For *VSX Vector Divide Double-Precision* (***xvdivdp***), do the following.

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For *VSX Vector Divide Single-Precision* (***xvdivsp***), do the following.

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

For *VSX Scalar Floating-Point Reciprocal Estimate*[2] instructions and *VSX Scalar Floating-Point Reciprocal Square Root Estimate*[3] instructions, do the following.

1. ZX is set to 1.
2. An Infinity, having the sign of the source operand, is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR and FI are set to 0.
4. FPRF is set to indicate the class and sign of the result (± Infinity).

For the *VSX Vector Reciprocal Estimate Double-Precision* (***xvredp***) and *VSX Vector Reciprocal Square Root Estimate Double-Precision* (***xvrsqrtedp***) instructions:

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For the *VSX Vector Reciprocal Estimate Single-Precision* (***xvresp***) and *VSX Vector Reciprocal Square Root Estimate Single-Precision* (***xvrsqrtesp***) instructions:

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

1. *VSX Scalar Floating-Point Divide* instructions: ***xsdivdp***, ***xsdivsp***
2. *VSX Scalar Floating-Point Reciprocal Estimate* instructions: ***xsredp***, ***xsresp***
3. *VSX Scalar Floating-Point Reciprocal Square Root Estimate* instructions: ***xsrsqrtedp***, ***xsrsqrtesp***

7.4.3 Floating-Point Overflow Exception

7.4.3.1 Definition

An Overflow exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

7.4.3.2 Action for OE=1

When Overflow exception is enabled (OE=1) and an Overflow exception occurs, the following actions are taken:

For the *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***) instruction:

1. OX is set to 1.
2. If the unbiased exponent of the normalized intermediate result is less than or equal to 318 (Emax+192), the exponent is adjusted by subtracting 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Double-Precision Arithmetic*[1] instructions, do the following.

1. OX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by subtracting 1536.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Single-Precision Arithmetic*[2] instructions, do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 192.
3. The adjusted and rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Vector Double-Precision Arithmetic*[3] instructions, *VSX Vector Single-Precision Arithmetic*[4] instructions, and *VSX Vector round and Convert Double-Precision to Single-Precision format* instruction (***xvcvdpsp***), do the following.

1. OX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR, FI, and FPRF are not modified.

1. *VSX Scalar Double-Precision Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xsredp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
2. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
3. *VSX Vector Double-Precision Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvredp***, ***xvsubdp***, ***xvmaddadp***, ***xsmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
4. *VSX Vector Single-Precision Arithmetic* instructions:
   xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvresp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvsmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.4.3.3 Action for OE=0

When Overflow exception is disabled (OE=0) and an Overflow exception occurs, the following actions are taken:

1. OX and XX are set to 1.

2. The result is determined by the rounding mode (RN) and the sign of the intermediate result as follows:

   Round to Nearest Even
   For negative overflow, the result is -Infinity.
   For positive overflow, the result is +Infinity.

   Round toward Zero
   For negative overflow, the result is the format's most negative finite number.
   For positive overflow, the result is the format's most positive finite number.

   Round toward +Infinity
   For negative overflow, the result is the format's most negative finite number.
   For positive overflow, the result is +Infinity.

   Round toward -Infinity
   For negative overflow, the result is -Infinity.
   For positive overflow, the result is the format's most positive finite number.

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***):

3. The result is placed into word element 0 of VSR[XT] as a single-precision value. The contents of word elements 1-3 of VSR[XT] are undefined.

4. FR is undefined.

5. FI is set to 1.

6. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Double-Precision Arithmetic*[1] instructions and *VSX Scalar Single-Precision Arithmetic*[2] instructions, do the following.

3. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are undefined.

4. FR is undefined.

5. FI is set to 1.

6. FPRF is set to indicate the class and sign of the result.

For *VSX Vector Double-Precision Arithmetic*[3] instructions, do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.

4. FR, FI, and FPRF are not modified.

For *VSX Vector Single-Precision Arithmetic*[4] instructions and *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format.

4. FR, FI, and FPRF are not modified.

1. *VSX Scalar Double-Precision Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xsredp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
2. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
3. *VSX Vector Double-Precision Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvredp***, ***xvsubdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
4. *VSX Vector Single-Precision Arithmetic* instructions:
   xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvresp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.4.4 Floating-Point Underflow Exception

7.4.4.1 Definition

Underflow exception is defined separately for the enabled and disabled states:

Enabled:
Underflow occurs when the intermediate result is “Tiny”.

Disabled:
Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A *tiny* result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is tiny and Underflow exception is disabled (UE=0), the intermediate result is denormalized (see Section 7.3.2.4 , “Normalization and Denormalization” on page 331) and rounded (see Section 7.3.2.6 , “Rounding” on page 335) before being placed into the target VSR.

Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

7.4.4.2 Action for UE=1

When Underflow exception is enabled (UE=1) and an Underflow exception occurs, the following actions are taken:

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***), do the following.

1. UX is set to 1.
2. If the unbiased exponent of the normalized intermediate result is greater than or equal to -319 (Emin-192), the exponent is adjusted by adding 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Double-Precision Arithmetic*[1] instructions *and VSX Scalar Double-Precision Reciprocal Estimate* (***xsredp***), do the following.

1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 1536.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

1. *VSX Scalar Double-Precision Arithmetic* instructions:
xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***

For *VSX Scalar Single-Precision Arithmetic*[1] instructions and *VSX Scalar Single-Precision Reciprocal Estimate* (***xsresp***), do the following.

1. UX is set to 1.
2. The exponent is adjusted by adding 192.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow exception, to simulate a "trap disabled" environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized and correctly rounded.

For *VSX Vector Floating-Point Arithmetic*[2] instructions, *VSX Vector Floating-Point Reciprocal Estimate*[3] instructions, and *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), do the following.

1. UX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR, FI, and FPRF are not modified.

7.4.4.3 Action for UE=0

When Underflow exception is disabled (UE=0) and an Underflow exception occurs, the following actions are taken:

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***), do the following.

1. UX is set to 1.
2. The result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Floating-Point Arithmetic*[4] instructions and *VSX Scalar Reciprocal Estimate*[5] instructions, do the following.

1. UX is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Vector Double-Precision Arithmetic*[6] instructions and *VSX Vector Reciprocal Estimate Double-Precision* (***xvredp***), do the following.

1. UX is set to 1.
2. For each vector element causing an Underflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

1. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
2. *VSX Vector Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvsubdp***, ***xvaddsp***, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***
3. *VSX Vector Floating-Point Reciprocal Estimate* instructions:
   xvredp, ***xvresp***
4. *VSX Scalar Floating-Point Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsaddsp***, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
5. *VSX Scalar Reciprocal Estimate* instructions:
   xsredp, ***xsresp***
6. *VSX Vector Double-Precision Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvsubdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***

For *VSX Vector Single-Precision Arithmetic*[1] instructions, *VSX Vector Reciprocal Estimate Single-Precision* (***xvresp***), and *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), do the following.

1. UX is set to 1.

2. For each vector element causing an Underflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format.

3. FR, FI, and FPRF are not modified.

1. *VSX Vector Single-Precision Arithmetic* instructions:
xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.4.5 Floating-Point Inexact Exception

7.4.5.1 Definition

An Inexact exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow exception or an enabled Underflow exception, an Inexact exception also occurs only if the significands of the rounded result and the intermediate result differ.)

2. The rounded result overflows and Overflow exception is disabled.

The action to be taken depends on the setting of the Inexact Exception Enable bit of the FPSCR.

7.4.5.2 Action for XE=1

Programming Note

In some implementations, enabling Inexact exceptions can degrade performance more than does enabling other types of floating-point exception.

When Inexact exception is enabled (UE=1) and an Inexact exception occurs, the following actions are taken:

For the *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***) instruction:

1. XX is set to 1.

2. The result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Floating-Point Arithmetic*[1] instructions, *VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode* (***xsrdpic***), and *VSX Scalar Integer to Floating-Point Format Conversion*[2] instructions, do the following.

1. XX is set to 1.

2. The result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Floating-Point to Integer Word Format Conversion*[3] instructions, do the following.

1. XX is set to 1.

2. The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For *VSX Vector Floating-Point Arithmetic*[4] instructions*, VSX Vector Floating-Point Reciprocal Estimate*[5] instructions, *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), *VSX Vector Double-Precision to Integer Format Conversion*[6] instructions, and *VSX Vector Integer to Floating-Point Format Conversion*[7] instructions, do the following.

1. XX is set to 1.

2. Update of VSR[XT] is suppressed for all vector elements.

3. FR, FI, and FPRF are not modified.

1. *VSX Scalar Floating-Point Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsaddsp***, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
2. *VSX Scalar Integer to Floating-Point Format Conversion* instructions:
   xscvsxddp, ***xscvuxddp***, ***xscvsxdsp***, ***xscvuxdsp***
3. *VSX Scalar Floating-Point to Integer Word Format Conversion* instructions:
   xscvdpsxws, ***xscvdpuxws***
4. *VSX Vector Floating-Point Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvsubdp***, ***xsaddsp***, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.4.5.3 Action for XE=0

When Inexact exception is disabled (XE=0) and an Inexact exception occurs, the following actions are taken:

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***), do the following.

1. XX is set to 1.
2. The result is placed into word element 0 of VSR[XT] as a single-precision value. The contents of word elements 1-3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Double-Precision Arithmetic*[1] instructions, *VSX Scalar Single-Precision Arithmetic*[2] instructions, *VSX Scalar Round to Single-Precision* (***xsrsp***), the *VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode* (***xsrdpic***), and *VSX Scalar Integer to Double-Precision Format Conversion*[3] instructions, do the following.

1. XX is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert Double-Precision To Integer Word format with Saturate*[4] instructions, do the following.

1. XX is set to 1.
2. The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Vector Double-Precision Arithmetic*[5] instructions, do the following.

1. XX is set to 1.
2. For each vector element causing an Inexact exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For *VSX Vector Single-Precision Arithmetic*[6] instructions, do the following.

1. XX is set to 1.
2. For each vector element causing an Inexact exception, the result is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

---

5. *VSX Vector Floating-Point Reciprocal Estimate* instructions:
   xvredp, ***xvresp***
6. *VSX Vector Double-Precision to Integer Format Conversion* instructions:
   xvcvdpsxds, ***xvcvdpsxws***, ***xvcvdpuxds***, ***xvcvdpuxws***
7. *VSX Vector Integer to Floating-Point Format Conversion* instructions:
   xvcvsxddp, ***xvcvuxddp***, ***xvcvsxdsp***, ***xvcvuxdsp***, ***xvcvsxwsp***, ***xvcvuxwsp***

1. *VSX Scalar Double-Precision Arithmetic* instructions:
   xsadddp, ***xssubdp***, ***xsmuldp***, ***xsdivdp***, ***xssqrtdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
2. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xssubsp***, ***xsmulsp***, ***xsdivsp***, ***xssqrtsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
3. *VSX Scalar Integer to Double-Precision Format Conversion* instructions:
   xscvsxddp, ***xscvuxddp***
4. *VSX Scalar Convert Double-Precision To Integer Word format with Saturate* instructions:
   xscvdpsxws, ***xscvdpuxws***
5. *VSX Vector Double-Precision Arithmetic* instructions:
   xsadddp, ***xssubdp***, ***xsmuldp***, ***xsdivdp***, ***xssqrtdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
6. *VSX Vector Single-Precision Arithmetic* instructions:
   xvaddsp, ***xvsubsp***, ***xvmulsp***, ***xvdivsp***, ***xvsqrtsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.5 VSX Storage Access Operations

The *VSX Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Power ISA Book I.

7.5.1 Accessing Aligned Storage Operands

The following quadword-aligned array, AH, consists of 8 halfwords.

```
short   AW[4] = { 0x0001_0203,
                  0x0405_0607,
                  0x0809_0A0B,
                  0x0C0D_0E0F };
```

Figure 119 illustrates the big-endian storage image of array AW.

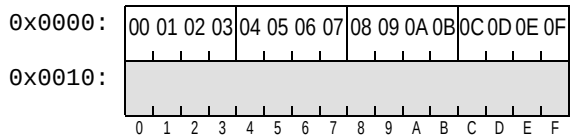


Figure 119.Big-endian storage image of array AW

Figure 120 illustrates the little-endian storage image of array AW.

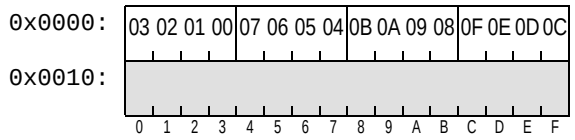


Figure 120.Little-endian storage image of array AW

Figure 121 shows the result of loading that quadword into a VSR or, equivalently, shows the contents that must be in a VSR if storing that VSR is to produce the storage contents shown in Figure 119 for big-endian. Note that Figure shows the effect of loading the quadword from both big-endian storage and little-endian storage.

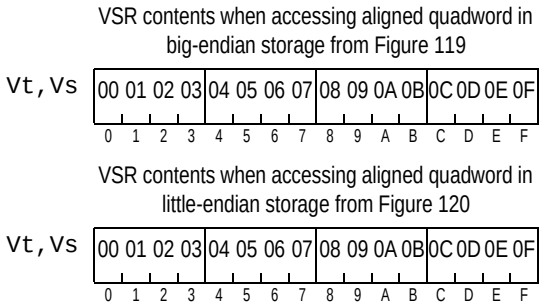


Figure 121.Vector-Scalar Register contents for aligned quadword Load or Store VSX Vector

7.5.2 Accessing Unaligned Storage Operands

The following array, B, consists of 5 word elements.

```
int   B[5];
B[0] = 0x01234567;
B[1] = 0x00112233;
B[2] = 0x44556677;
B[3] = 0x8899AABB;
B[4] = 0xCCDDEEFF;
```

Figure 122 illustrates both big-endian and little-endian storage images of array B.

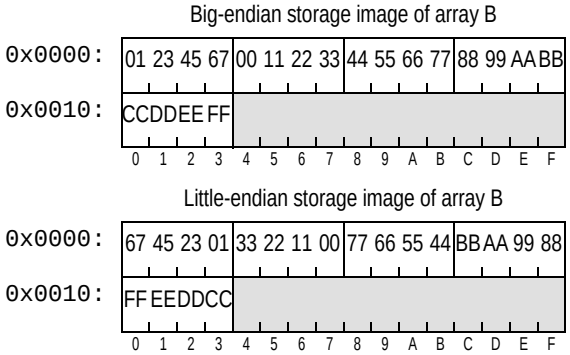


Figure 122.Storage images of array B

Though this example shows the array starting at a quadword-aligned address, if the subject data of interest are elements 1 through 4, accessing elements 1 through 4 of array B involves an unaligned quadword storage access that spans two aligned quadwords.

Loading an Unaligned Quadword from Big-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using big-endian byte ordering.

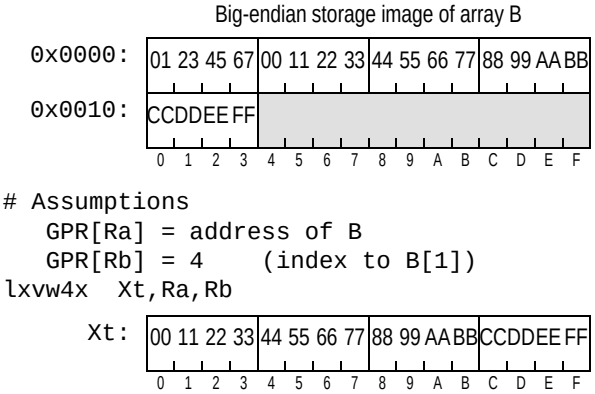


Figure 123.Process to load misaligned quadword from big-endian storage using Load VSX Vector Word*4 Indexed

Loading an Unaligned Quadword from Little-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using little-endian byte ordering.

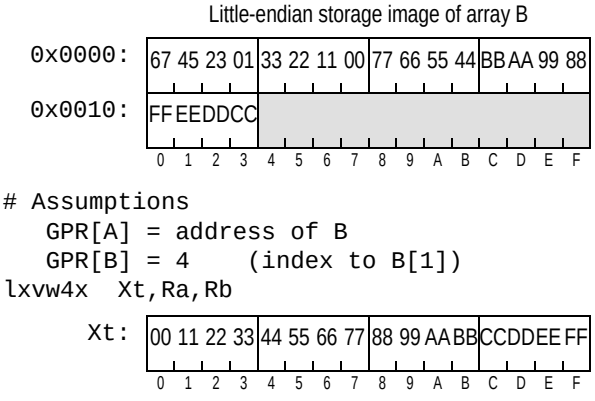
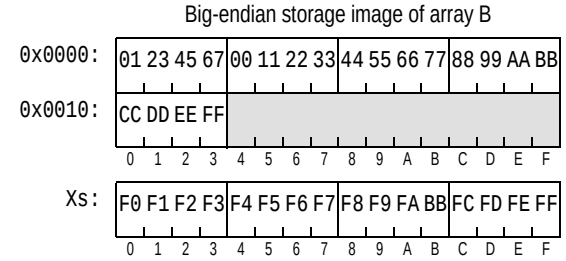


Figure 124.Process to load misaligned quadword from little-endian storage Load VSX Vector Word*4 Indexed

Storing an Unaligned Quadword to Big-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using big-endian byte ordering.




```
# Assumptions
   GPR[Ra] = address of B
   GPR[Rb] = 4    (index to B[1])

stxvw4x  Xs,Ra,Rb
```

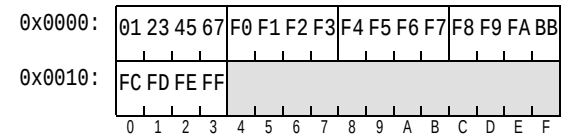
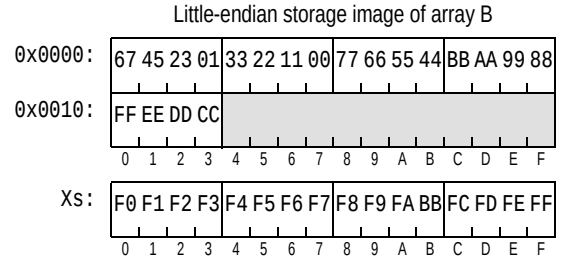


Figure 125.Process to store misaligned quadword to big-endian storage using Store VSX Vector Word*4 Indexed

Storing an Unaligned Quadword to Little-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 122) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using little-endian byte ordering.




```
# Assumptions
   GPR[A] = address of B
   GPR[B] = 4    (index to B[1])

stxvw4x  Xs,Ra,Rb
```

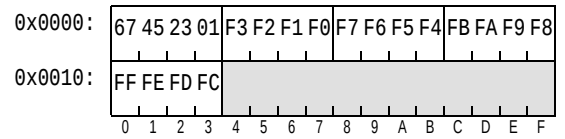


Figure 126.Process to store misaligned quadword to little-endian storage Store VSX Vector Word*4 Indexed

7.5.3 Storage Access Exceptions

Storage accesses cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

The address used for VSX storage accesses must be word-aligned to guarantee the access can be performed in hardware rather than result in an Alignment interrupt.

7.6 VSX Instruction Set

7.6.1 VSX Instruction Set Summary

7.6.1.1 VSX Storage Access Instructions

There are two basic forms of scalar load and scalar store instructions, word and doubleword. *VSX Scalar Load* instructions place a copy of the contents of the addressed word or doubleword in storage into the left-most word or doubleword element of the target VSR. The contents of the right-most element(s) of the target VSR are undefined. *VSX Scalar Store* instructions place a copy of the contents of the left-most word or doubleword element in the source VSR into the addressed word or doubleword in storage.

There are two basic forms of vector load and vector store instructions, a vector of 4 word elements and a vector of two doublewords. Both forms access a quadword in storage.

There is one basic form of vector load and splat instruction, doubleword. *VSX Vector Load and Splat* instruction places a copy of the contents of the addressed doubleword in storage into both doubleword elements of the target VSR.

7.6.1.1.1 VSX Scalar Storage Access Instructions

Mnemonic	Instruction Name	Page
lxsdx	Load VSX Scalar Doubleword Indexed	393
lxsspx	Load VSX Scalar Single-Precision Indexed	394
lxsiwax	Load VSX Scalar as Integer Word Algebraic Indexed	393
lxsiwzx	Load VSX Scalar as Integer Word and Zero Indexed	394

Table 8. VSX Scalar Load Instructions

Mnemonic	Instruction Name	Page
stxsdx	Store VSX Scalar Doubleword Indexed	396
stxsspx	Store VSX Scalar Single-Precision Indexed	397
stxsiwx	Store VSX Scalar as Integer Word Indexed	397

Table 9. VSX Scalar Store Instructions

7.6.1.1.2 VSX Vector Storage Access Instructions

Mnemonic	Instruction Name	Page
lxvd2x	Load VSX Vector Doubleword*2 Indexed	395
lxvw4x	Load VSX Vector Word*4 Indexed	396

Table 10.VSX Vector Load Instructions

Mnemonic	Instruction Name	Page
lxvdsx	Load VSX Vector Doubleword and Splat Indexed	395

Table 11.VSX Vector Load and Splat Instruction

Mnemonic	Instruction Name	Page
stxvd2x	Store VSX Vector Doubleword*2 Indexed	398
stxvw4x	Store VSX Vector Word*4 Indexed	398

Table 12.VSX Vector Store Instructions

7.6.1.2 VSX Move Instructions

7.6.1.2.1 VSX Scalar Move Instructions

Mnemonic	Instruction Name	Page
xsabsdp	VSX Scalar Absolute Value Double-Precision	399
xscpsgndp	VSX Scalar Copy Sign Double-Precision	411
xsnabsdp	VSX Scalar Negative Absolute Value Double-Precision	449
xsnegdp	VSX Scalar Negate Double-Precision	449

Table 13.VSX Scalar Double-Precision Move Instructions

7.6.1.2.2 VSX Vector Move Instructions

Mnemonic	Instruction Name	Page
xvabsdp	VSX Vector Absolute Value Double-Precision	480
xvcpsgndp	VSX Vector Copy Sign Double-Precision	494
xvnabsdp	VSX Vector Negative Absolute Value Double-Precision	545
xvnegdp	VSX Vector Negate Double-Precision	546

Table 14.VSX Vector Double-Precision Move Instructions

Mnemonic	Instruction Name	Page
xvabssp	VSX Vector Absolute Value Single-Precision	481
xvcpsgnsp	VSX Vector Copy Sign Single-Precision	494
xvnabssp	VSX Vector Negative Absolute Value Single-Precision	545
xvnegsp	VSX Vector Negate Single-Precision	546

Table 15.VSX Vector Single-Precision Move Instructions

7.6.1.3 VSX Floating-Point Arithmetic Instructions

7.6.1.3.1 VSX Scalar Floating-Point Arithmetic Instructions

Mnemonic	Instruction Name	Page
xsadddp	VSX Scalar Add Double-Precision	400
xsdivdp	VSX Scalar Divide Double-Precision	425
xsmuldp	VSX Scalar Multiply Double-Precision	445
xsredp	VSX Scalar Reciprocal Estimate Double-Precision	468
xsrsqrtedp	VSX Scalar Reciprocal Square Root Estimate Double-Precision	471
xssqrtdp	VSX Scalar Square Root Double-Precision	473
xssubdp	VSX Scalar Subtract Double-Precision	475
xstdivdp	VSX Scalar Test for software Divide Double-Precision	479
xstsqrtdp	VSX Scalar Test for software Square Root Double-Precision	480

Table 16.VSX Scalar Double-Precision Elementary Arithmetic Instructions

Mnemonic	Instruction Name	Page
xsaddsp	VSX Scalar Add Single-Precision	405
xsdivsp	VSX Scalar Divide Single-Precision	427
xsmulsp	VSX Scalar Multiply Single-Precision	447
xsresp	VSX Scalar Reciprocal Estimate Single-Precision	469
xsrsqrtesp	VSX Scalar Reciprocal Square Root Estimate Single-Precision	472
xssqrtsp	VSX Scalar Square Root Single-Precision	474
xssubsp	VSX Scalar Subtract Single-Precision	477

Table 17.VSX Scalar Single-Precision Elementary Arithmetic Instructions

Mnemonic	Instruction Name	Page
xsmaddadp	VSX Scalar Multiply-Add Type-A Double-Precision	429
xsmaddmdp	VSX Scalar Multiply-Add Type-M Double-Precision	429
xsmsubadp	VSX Scalar Multiply-Subtract Type-A Double-Precision	439
xsmsubmdp	VSX Scalar Multiply-Subtract Type-M Double-Precision	439
xsnmaddadp	VSX Scalar Negative Multiply-Add Type-A Double-Precision	450
xsnmaddmdp	VSX Scalar Negative Multiply-Add Type-M Double-Precision	450
xsnmsubadp	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision	458
xsnmsubmdp	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision	458

Table 18.VSX Scalar Double-Precision Multiply-Add Arithmetic Instructions

Mnemonic	Instruction Name	Page
xsmaddasp	VSX Scalar Multiply-Add Type-A Single-Precision	432
xsmaddmsp	VSX Scalar Multiply-Add Type-M Single-Precision	432
xsmsubasp	VSX Scalar Multiply-Subtract Type-A Single-Precision	442
xsmsubmsp	VSX Scalar Multiply-Subtract Type-M Single-Precision	442
xsnmaddasp	VSX Scalar Negative Multiply-Add Type-A Single-Precision	455
xsnmaddmsp	VSX Scalar Negative Multiply-Add Type-M Single-Precision	455
xsnmsubasp	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision	461
xsnmsubmsp	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision	461

Table 19.VSX Scalar Single-Precision Multiply-Add Arithmetic Instructions

7.6.1.3.2 VSX Vector Floating-Point Arithmetic Instructions

Mnemonic	Instruction Name	Page
xvadddp	VSX Vector Add Double-Precision	482
xvdivdp	VSX Vector Divide Double-Precision	517
xvmuldp	VSX Vector Multiply Double-Precision	541
xvredp	VSX Vector Reciprocal Estimate Double-Precision	564
xvrsqrtedp	VSX Vector Reciprocal Square Root Estimate Double-Precision	568
xvsqrtdp	VSX Vector Square Root Double-Precision	571
xvsubdp	VSX Vector Subtract Double-Precision	573
xvtdivdp	VSX Vector Test for software Divide Double-Precision	577
xvtsqrtdp	VSX Vector Test for software Square Root Double-Precision	579

Table 20.VSX Vector Double-Precision Elementary Arithmetic Instructions

Mnemonic	Instruction Name	Page
xvaddsp	VSX Vector Add Single-Precision	486
xvdivsp	VSX Vector Divide Single-Precision	519
xvmulsp	VSX Vector Multiply Single-Precision	543
xvresp	VSX Vector Reciprocal Estimate Single-Precision	565
xvrsqrtesp	VSX Vector Reciprocal Square Root Estimate Single-Precision	570
xvsqrtsp	VSX Vector Square Root Single-Precision	572
xvsubsp	VSX Vector Subtract Single-Precision	575
xvtdivsp	VSX Vector Test for software Divide Single-Precision	578
xvtsqrtsp	VSX Vector Test for software Square Root Single-Precision	579

Table 21.VSX Vector Single-Precision Elementary Arithmetic Instructions

Mnemonic	Instruction Name	Page
xvmaddadp	VSX Vector Multiply-Add Type-A Double-Precision	521
xvmaddmdp	VSX Vector Multiply-Add Type-M Double-Precision	521
xvmsubadp	VSX Vector Multiply-Subtract Type-A Double-Precision	535
xvmsubmdp	VSX Vector Multiply-Subtract Type-M Double-Precision	535
xvnmaddadp	VSX Vector Negative Multiply-Add Type-A Double-Precision	547
xvnmaddmdp	VSX Vector Negative Multiply-Add Type-M Double-Precision	547
xvnmsubadp	VSX Vector Negative Multiply-Subtract Type-A Double-Precision	555
xvnmsubmdp	VSX Vector Negative Multiply-Subtract Type-M Double-Precision	555

Table 22.VSX Vector Double-Precision Multiply-Add Arithmetic Instructions

Mnemonic	Instruction Name	Page
xvmaddasp	VSX Vector Multiply-Add Type-A Single-Precision	524
xvmaddmsp	VSX Vector Multiply-Add Type-M Single-Precision	524
xvmsubasp	VSX Vector Multiply-Subtract Type-A Single-Precision	538
xvmsubmsp	VSX Vector Multiply-Subtract Type-M Single-Precision	538
xvnmaddasp	VSX Vector Negative Multiply-Add Type-A Single-Precision	552
xvnmaddmsp	VSX Vector Negative Multiply-Add Type-M Single-Precision	552
xvnmsubasp	VSX Vector Negative Multiply-Subtract Type-A Single-Precision	558
xvnmsubmsp	VSX Vector Negative Multiply-Subtract Type-M Single-Precision	558

Table 23.VSX Vector Single-Precision Multiply-Add Arithmetic Instructions

7.6.1.4 VSX Floating-Point Compare Instructions

7.6.1.4.1 VSX Scalar Floating-Point Compare Instructions

Mnemonic	Instruction Name	Page
xscmpodp	VSX Scalar Compare Ordered Double-Precision	407
xscmpudp	VSX Scalar Compare Unordered Double-Precision	409

Table 24.VSX Scalar Compare Double-Precision Instructions

Mnemonic	Instruction Name	Page
xsmaxdp	VSX Scalar Maximum Double-Precision	435
xsmindp	VSX Scalar Minimum Double-Precision	437

Table 25.VSX Scalar Double-Precision Maximum/Minimum Instructions

7.6.1.4.2 VSX Vector Floating-Point Compare Instructions

Mnemonic	Instruction Name	Page
xvcmpeqdp[.]	VSX Vector Compare Equal To Double-Precision	488
xvcmpgedp[.]	VSX Vector Compare Greater Than or Equal To Double-Precision	490
xvcmpgtdp[.]	VSX Vector Compare Greater Than Double-Precision	492

Table 26.VSX Vector Compare Double-Precision Instructions

Mnemonic	Instruction Name	Page
xvcmpeqsp[.]	VSX Vector Compare Equal To Single-Precision	489
xvcmpgesp[.]	VSX Vector Compare Greater Than or Equal To Single-Precision	491
xvcmpgtsp[.]	VSX Vector Compare Greater Than Single-Precision	493

Table 27.VSX Vector Compare Single-Precision Instructions

Mnemonic	Instruction Name	Page
xvmaxdp	VSX Vector Maximum Double-Precision	527
xvmindp	VSX Vector Minimum Double-Precision	531

Table 28.VSX Vector Double-Precision Maximum/Minimum Instructions

Mnemonic	Instruction Name	Page
xvmaxsp	VSX Vector Maximum Single-Precision	529
xvminsp	VSX Vector Minimum Single-Precision	533

Table 29.VSX Vector Single-Precision Maximum/Minimum Instructions

7.6.1.5 VSX DP-SP Conversion Instructions

7.6.1.5.1 VSX Scalar DP-SP Conversion Instructions

Mnemonic	Instruction Name	Page
xscvdpsp	VSX Scalar round and Convert Double-Precision to Single-Precision format	412
xscvspdp	VSX Scalar Convert Single-Precision to Double-Precision format	422

Table 30.VSX Scalar DP-SP Conversion Instructions

7.6.1.5.2 VSX Vector DP-SP Conversion Instructions

Mnemonic	Instruction Name	Page
xvcvdpsp	VSX Vector round and Convert Double-Precision to Single-Precision format	495
xvcvspdp	VSX Vector Convert Single-Precision to Double-Precision format	504

Table 31.VSX Vector DP-SP Conversion Instructions

7.6.1.6 VSX Integer Conversion Instructions

7.6.1.6.1 VSX Scalar Integer Conversion Instructions

Mnemonic	Instruction Name	Page
xscvdpsxds	VSX Scalar truncate Double-Precision to integer and Convert to Signed Fixed-Point Doubleword format with Saturate	413
xscvdpsxws	VSX Scalar truncate Double-Precision to integer and Convert to Signed Fixed-Point Word format with Saturate	416
xscvdpuxds	VSX Scalar truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Doubleword format with Saturate	418
xscvdpuxws	VSX Scalar truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Word format with Saturate	420

Table 32.VSX Scalar Convert Double-Precision to Integer Instructions

Mnemonic	Instruction Name	Page
xscvsxddp	VSX Scalar Convert Signed Fixed-Point Doubleword to floating-point format and round to Double-Precision	423
xscvuxddp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to floating-point format and round to Double-Precision	424

Table 33.VSX Scalar Convert Integer to Double-Precision Instructions

Mnemonic	Instruction Name	Page
xscvsxdsp	VSX Scalar Convert Signed Fixed-Point Doubleword to floating-point format and round to Single-Precision	423
xscvuxdsp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to floating-point format and round to Single-Precision	424

Table 34.VSX Scalar Convert Integer to Single-Precision Instructions

7.6.1.6.2 VSX Vector Integer Conversion Instructions

Mnemonic	Instruction Name	Page
xvcvdpsxds	VSX Vector truncate Double-Precision to integer and Convert to Signed Fixed-Point Doubleword format with Saturate	496
xvcvdpsxws	VSX Vector truncate Double-Precision to integer and Convert to Signed Fixed-Point Word format with Saturate	498
xvcvdpuxds	VSX Vector truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Doubleword format with Saturate	500
xvcvdpuxws	VSX Vector truncate Double-Precision to integer and Convert to Unsigned Fixed-Point Word format with Saturate	502

Table 35.VSX Vector Convert Double-Precision to Integer Instructions

Mnemonic	Instruction Name	Page
xvcvspsxds	VSX Vector truncate Single-Precision to integer and Convert to Signed Fixed-Point Doubleword format with Saturate	505
xvcvspsxws	VSX Vector truncate Single-Precision to integer and Convert to Signed Fixed-Point Word format with Saturate	507
xvcvspuxds	VSX Vector truncate Single-Precision to integer and Convert to Unsigned Fixed-Point Doubleword format with Saturate	509
xvcvspuxws	VSX Vector truncate Single-Precision to integer and Convert to Unsigned Fixed-Point Word format with Saturate	511

Table 36.VSX Vector Convert Single-Precision to Integer Instructions

Mnemonic	Instruction Name	Page
xvcvsxddp	VSX Vector Convert and round Signed Fixed-Point Doubleword to Double-Precision format	513
xvcvsxwdp	VSX Vector Convert Signed Fixed-Point Word to Double-Precision format	514
xvcvuxddp	VSX Vector Convert and round Unsigned Fixed-Point Doubleword to Double-Precision format	515
xvcvuxwdp	VSX Vector Convert Unsigned Fixed-Point Word to Double-Precision format	516

Table 37.VSX Vector Convert Integer to Double-Precision Instructions

Mnemonic	Instruction Name	Page
xvcvsxdsp	VSX Vector Convert and round Signed Fixed-Point Doubleword to Single-Precision format	513
xvcvsxwsp	VSX Vector Convert and round Signed Fixed-Point Word to Single-Precision format	514
xvcvuxdsp	VSX Vector Convert and round Unsigned Fixed-Point Doubleword to Single-Precision format	515
xvcvuxwsp	VSX Vector Convert and round Unsigned Fixed-Point Word to Single-Precision format	516

Table 38.VSX Vector Convert Integer to Single-Precision Instructions

7.6.1.7 VSX Round to Floating-Point Integer Instructions

7.6.1.7.1 VSX Scalar Round to Floating-Point Integer Instructions

Mnemonic	Instruction Name	Page
xsrdpi	VSX Scalar Round to Double-Precision Integer using round to Nearest Away	464
xsrdpic	VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode	465
xsrdpim	VSX Scalar Round to Double-Precision Integer using round towards -Infinity rounding mode	466
xsrdpip	VSX Scalar Round to Double-Precision Integer using round towards +Infinity rounding mode	466
xsrdpiz	VSX Scalar Round to Double-Precision Integer using round towards Zero rounding mode	467

Table 39.VSX Scalar Round to Double-Precision Integer Instructions

7.6.1.7.2 VSX Vector Round to Floating-Point Integer Instructions

Mnemonic	Instruction Name	Page
xvrdpi	VSX Vector Round to Double-Precision Integer using round to Nearest Away	561
xvrdpic	VSX Vector Round to Double-Precision Integer Exact using Current rounding mode	561
xvrdpim	VSX Vector Round to Double-Precision Integer using round towards -Infinity rounding mode	562
xvrdpip	VSX Vector Round to Double-Precision Integer using round towards +Infinity rounding mode	562
xvrdpiz	VSX Vector Round to Double-Precision Integer using round towards Zero rounding mode	563

Table 40.VSX Vector Round to Double-Precision Integer Instructions

Mnemonic	Instruction Name	Page
xvrspi	VSX Vector Round to Single-Precision Integer using round to Nearest Away	566
xvrspic	VSX Vector Round to Single-Precision Integer Exact using Current rounding mode	566
xvrspim	VSX Vector Round to Single-Precision Integer using round towards -Infinity rounding mode	567
xvrspip	VSX Vector Round to Single-Precision Integer using round towards +Infinity rounding mode	567
xvrspiz	VSX Vector Round to Single-Precision Integer using round towards Zero rounding mode	568

Table 41.VSX Vector Round to Single-Precision Integer Instructions

7.6.1.8 VSX Logical Instructions

Mnemonic	Instruction Name	Page
xxland	VSX Logical AND	580
xxlandc	VSX Logical AND with Complement	580
xxlnor	VSX Logical NOR	582
xxlor	VSX Logical OR	583
xxlxor	VSX Logical XOR	583

Table 42.VSX Logical Instructions

Mnemonic	Instruction Name	Page
xxsel	VSX Select	585

Table 43.VSX Vector Select Instruction

7.6.1.9 VSX Permute Instructions

Mnemonic	Instruction Name	Page
xxmrghw	**VSX Merge High Word**	584
xxmrglw	**VSX Merge Low Word**	584

Table 44.VSX Merge Instructions

Mnemonic	Instruction Name	Page
xxspltw	VSX Splat Word	586

Table 45.VSX Splat Instruction

Mnemonic	Instruction Name	Page
xxpermdi	VSX Permute Doubleword Immediate	585

Table 46.VSX Permute Instruction

Mnemonic	Instruction Name	Page
xxsldwi	VSX Shift Left Double by Word Immediate	586

Table 47.VSX Shift Instruction

7.6.2 VSX Instruction Description Conventions

7.6.2.1 VSX Instruction RTL Operators

x.bit[y]
Return the contents of bit y of x.

x.bit[y:z]
Return the contents of bits y:z of x.

x.word[y]
Return the contents of word element y of x.

x.word[y:z]
Return the contents of word elements y:z of x.

x.dword[y]
Return the contents of doubleword element y of x.

x.dword[y:z]
Return the contents of doubleword elements y:z of x.

x = y
The value of y is placed into x.

x |= y
The value of y is ORed with the value x and placed into x.

~x
Return the one's complement of x.

!x
Return 1 if the contents of x are equal to 0, otherwise return 0.

x || y
Return the value of x concatenated with the value of y. For example, `0b010 || 0b111` is the same as `0b010111`.

x ^ y
Return the value of x exclusive ORed with the value of y.

x ? y : z
If the value of x is true, return the value of y, otherwise return the value z.

x+y
x and y are integer values.

Return the sum of x and y.

x–y
x and y are integer values.

Return the difference of x and y.

x!=y
x and y are integer values.

Return 1 if x is not equal to y, otherwise return 0.

x<=y
x and y are integer values.

Return 1 if x is less than or equal to y, otherwise return 0.

x>=y
x and y are integer values.

Return 1 if x is greater than or equal to y, otherwise return 0.

7.6.2.2 VSX Instruction RTL Function Calls

AddDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is an Infinity and y is an Infinity of the opposite sign, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x and y, having unbounded range and precision.

AddSP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is an Infinity and y is an Infinity of the opposite sign, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x added to y, having unbounded range and precision.

ClassDP(x,y)

Return a 5-bit characterization of the double-precision floating-point number x.

0b10001 = Quiet NaN
0b01001 = -Infinity
0b01000 = -Normalized Number
0b11000 = -Denormalized Number
0b10010 = -Zero
0b00010 = +Zero
0b10100 = +Denormalized Number
0b00100 = +Normalized Number
0b00101 = +Infinity

ClassSP(x,y)

Return a 5-bit characterization of the single-precision floating-point number x.

0b10001 = Quiet NaN
0b01001 = -Infinity
0b01000 = -Normalized Number
0b11000 = -Denormalized Number
0b10010 = -Zero
0b00010 = +Zero
0b10100 = +Denormalized Number
0b00100 = +Normalized Number
0b00101 = +Infinity

CompareEQDP(x,y)
x and y are double-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is equal to y, return 1.
Otherwise, return 0.

CompareEQSP(x,y)
x and y are single-precision floating-point values.

If x or y is a NaN, return 0,
Otherwise, if x is equal to y, return 1.
Otherwise, return 0.

CompareGTDP(x,y)
x and y are double-precision floating-point values.

If x or y is a NaN, return 0,
Otherwise, if x is greater than y, return 1.
Otherwise, return 0.

CompareGTSP(x,y)
x and y are single-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is greater than y, return 1.
Otherwise, return 0.

CompareLTDP(x,y)
x and y are double-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is less than y, return 1.
Otherwise, return 0.

CompareLTSP(x,y)
x and y are single-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is less than y, return 1.
Otherwise, return 0.

ConvertDPtoSD(x)
x is a double-precision integer value.

If x is a NaN,
return `0x8000_0000_0000_0000`,
`vxcvi_flag` is set to 1, and
`vxsnan_flag` is set to 1 if x is an SNaN.

Otherwise, do the following.
If x is greater than 2^{63}-1,
return `0x7FFF_FFFF_FFFF_FFFF` and
`vxcvi_flag` is set to 1.

Otherwise, if x is less than -2^{63},
return `0x8000_0000_0000_0000` and
`vxcvi_flag` is set to 1.

Otherwise, return the value x in 64-bit signed integer format.

ConvertDPtoSP(x)

x is a double-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.

If x is a SNaN, returns x, converted to a QNaN, in single-precision floating-point format.

Otherwise, if x is a QNaN, an Infinity, or a Zero, returns x in single-precision floating-point format.

Otherwise, returns x, rounded to single-precision using the rounding mode specified in RN, in single-precision floating-point format.

ox_flag is set to 1 if rounding x resulted in an Overflow exception.
ux_flag is set to 1 if rounding x resulted in an Underflow exception.
xx_flag is set to 1 if rounding x returns an inexact result.
inc_flag is set to 1 if the significand of the result was incremented during rounding.

ConvertDPtoSP_NS(x)

x is a single-precision floating-point value represented in double-precision format.

Returns x in single-precision format.

```
sign     ← x.bit[0]
exponent ← x.bit[1:11]
fraction ← 0b1 || x.bit[12:63]                              // implicit bit set to 1 (for now)

if (exponent == 0) & (fraction.bit[1:52] != 0) then do      // DP Denormal operand
   exponent ← 0b000_0000_0001                               //   exponent override to DP Emin = 1
   fraction.bit[0] ← 0b0                                    //   implicit bit override to 0
end

if (exponent < 897) && (fraction != 0) then do              // SP tiny operand
   fraction ← fraction >>ui (897 - exponent)                //  denormalize until exponent = SP Emin
   exponent ← 0b011_1000_0000                               //  exponent override to SP Emin-1 = 896
end

return(sign || exponent.bit[0] || exponent.bit[4:10] || fraction.bit[1:23])
```

Programming Note

If x is not representable in single-precision, some exponent and/or significand bits will be discarded, likely producing undesirable results. The low-order 29 bits of the significand of x are discarded, more if the unbiased exponent of x is less than -126 (i.e., denormal). Finite values of x having an unbiased exponent less than -155 will return a result of Zero. Finite values of x having an unbiased exponent greater than +127 will result in discarding significant bits of the exponent. SNaN inputs having no significant bits in the upper 23 bits of the signifcand will return Infinity as the result. No status is set for any of these cases.

ConvertDPtoSW(x)

x is a double-precision integer value.

If x is a NaN,
return `0x8000_0000`,
`vxcvi_flag` is set to 1, and
`vxsnan_flag` is set to 1 if x is an SNaN

Otherwise, do the following.
If x is greater than 2^{31}-1,
return `0x7FFF_FFFF` and
`vxcvi_flag` is set to 1.

Otherwise, if x is less than -2^{31},
return `0x8000_0000` and
`vxcvi_flag` is set to 1.

Otherwise, return the value x in 32-bit signed integer format.

ConvertDPtoUD(x)

x is a double-precision integer value.

If x is a NaN,
return `0x0000_0000_0000_0000`,
`vxcvi_flag` is set to 1, and
`vxsnan_flag` is set to 1 if x is an SNaN

Otherwise, do the following.
If x is greater than 2^{64}-1,
return `0xFFFF_FFFF_FFFF_FFFF` and
`vxcvi_flag` is set to 1.

Otherwise, if x is less than 0,
return `0x0000_0000_0000_0000` and
`vxcvi_flag` is set to 1.

Otherwise, return the value x in 64-bit unsigned integer format.

ConvertDPtoUW(x)

x is a double-precision integer value.

If x is a NaN,
return `0x0000_0000`,
`vxcvi_flag` is set to 1, and
`vxsnan_flag` is set to 1 if x is an SNaN

Otherwise, do the following.

If x is greater than 2^{32}-1,
return `0xFFFF_FFFF` and
`vxcvi_flag` is set to 1.

Otherwise, if x is less than 0,
return `0x0000_0000` and
`vxcvi_flag` is set to 1.

Otherwise, return the value x in 32-bit unsigned integer format.

ConvertFPtoDP(x)

Return the floating-point value x in DP format.

ConvertFPtoSP(x)

Return the floating-point value x in single-precision format.

ConvertSDtoFP(x)

x is a 64-bit signed integer value.

Return the value x converted to floating-point format having unbounded significand precision.

ConvertSPtoDP_NS(x)

x is a single-precision floating-point value.

Returns x in double-precision format.

```
sign     ← x.bit[0]
exponent ← (x.bit[1] || ¬x.bit[1] || ¬x.bit[1] || ¬x.bit[1] || x.bit[2:8])
fraction ← 0b0 || x.bit[9:31] || 0b0_0000_0000_0000_0000_0000_0000_0000

if (x.bit[1:8] == 255) then do                          // Infinity or NaN operand
   exponent ← 2047                                      //   override exponent to DP Emax+1
end

else if (x.bit[1:8] == 0) && (fraction == 0) then do    // SP Zero operand
   exponent ← 0                                         //   override exponent to DP Emin-1
end

else if (x.bit[1:8] == 0) && (fraction != 0) then do    // SP Denormal operand
   exponent ← 897                                       //   override exponent to SP Emin
   do while (fraction.bit[0] == 0)                      //   normalize operand
      fraction ← fraction << 1
      exponent ← exponent - 1
   end
end

return(sign || exponent || fraction.bit[1:52])
```

ConvertSP64toSP(x)

x is a single-precision floating-point value in double-precision format.

Returns the value x in single-precision format. x must be representable in single-precision, or else result returned is undefined. x may require denormalization. No rounding is performed. If x is a SNaN, it is converted to a single-precision SNaN having the same payload as x.

```
sign ← x.bit[0]
exp  ← x.bit[1:11] - 1023
frac ← x.bit[12:63]

if      (exp = -1023) & (frac  = 0) & (sign=0) then return(0x0000_0000) // +Zero
else if (exp = -1023) & (frac  = 0) & (sign=1) then return(0x8000_0000) // -Zero
else if (exp = -1023) & (frac != 0)            then return(0xUUUU_UUUU) // DP denorm
else if (exp < -126) then do // denormalization required
   msb = 1
   do while (exp < -126)     // denormalize operand until exp=Emin
      frac.bit[1:51] ← frac.bit[0:50]
      frac.bit[0]    ← msb
      msb            ← 0
      exp            ← exp + 1
   end
   if (frac = 0) then return(0xUUUU_UUUU)  // value not representable in SP format
   else do // return denormal SP
      result.bit[0]    ← sign
      result.bit[1:8]  ← 0
      result.bit[9:31] ← frac.bit[0:22]
      return(result)
   end
end
else if (exp = +1024) & (frac  = 0) & (sign=0) then return(0x7F80_0000) // +Infinity
else if (exp = +1024) & (frac  = 0) & (sign=1) then return(0xFF80_0000) // -Infinity
else if (exp = +1024) & (frac != 0) then do   // QNaN or SNaN
   result.bit[0]    ← sign
   result.bit[1:8]  ← 255
   result.bit[9:31] ← frac.bit[0:22]
   return(result)
end
else if (exp < +1024) & (exp > +126) then return(0xUUUU_UUUU)  // overflow
else do  // normal value
   result.bit[0]    ← sign
   result.bit[1:8]  ← exp.bit[4:11] + 127
   result.bit[9:31] ← frac.bit[0:22]
   return(result)
end
```

ConvertSPtoDP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is an SNaN, return x represented as a QNaN in double-precision floating-point format.
Otherwise, if x is an QNaN, return x in double-precision floating-point format.
Otherwise, return the value x in double-precision floating-point format.

ConvertSPtoSD(x)

x is a single-precision integer value.

If x is a NaN,
return `0x8000_0000_0000_0000` and
`vxcvi_flag` is set to 1, and
`vxsnan_flag` is set to 1 if x is an SNaN

Otherwise, do the following.

If x is greater than 2^{63}-1,
return 0x7FFF_FFFF_FFFF_FFFF and
vxcvi_flag is set to 1.

Otherwise, if x is less than -2^{63},
return 0x8000_0000_0000_0000 and
vxcvi_flag is set to 1.

Otherwise, return the value x in 64-bit signed integer format.

ConvertSPtoSP64(x)

x is a floating-point value in single-precision format.

Returns the value x in double-precision format. If x is a SNaN, it is converted to a double-precision SNaN having the same payload as x.

```
sign ← x.bit[0]
exp  ← x.bit[1:8] - 127
frac ← x.bit[9:31]

if (exp = –127) & (frac != 0) then do // Normalize the Denormal value
   msb  ← frac.bit[0]
   frac ← frac << 1
   do while (msb = 0)
      msb  ← frac.bit[0]
      frac ← frac << 1
      exp  ← exp – 1
   end
end
else if (exp = -127) & (frac = 0) then exp ← –1023  // Zero value
else if (exp = +128)              then exp ← +1024  // Infinity, NaN

result.bit[0]     ← sign
result.bit[1:11]  ← exp + 1023
result.bit[12:34] ← frac
result.bit[35:63] ← 0
return(result)
```

ConvertSPtoSW(x)

x is a single-precision integer value.

If x is a NaN,
return 0x8000_0000,
vxcvi_flag is set to 1, and
vxsnan_flag is set to 1 if x is an SNaN

Otherwise, do the following.
If x is greater than 2^{31}-1,
return 0x7FFF_FFFF and
vxcvi_flag is set to 1.

Otherwise, if x is less than -2^{31},
return 0x8000_0000 and
vxcvi_flag is set to 1.

Otherwise, return the value x in 32-bit signed integer format.

ConvertSPtoUD(x)

x is a single-precision integer value.

If x is a NaN,
return 0x0000_0000_0000_0000,

vxcvi_flag is set to 1, and
vxsnan_flag is set to 1 if x is an SNaN

Otherwise, do the following.
If x is greater than 2^{64}-1,
return 0xFFFF_FFFF_FFFF_FFFF and
vxcvi_flag is set to 1.

Otherwise, if x is less than 0,
return 0x0000_0000_0000_0000 and
vxcvi_flag is set to 1.

Otherwise, return the value x in 64-bit unsigned integer format.

ConvertSPtoUW(x)
x is a single-precision integer value.

If x is a NaN,
return 0x0000_0000,
vxcvi_flag is set to 1, and
vxsnan_flag is set to 1 if x is an SNaN

Otherwise, do the following.
If x is greater than 2^{32}-1,
return 0xFFFF_FFFF and
vxcvi_flag is set to 1.

Otherwise, if x is less than 0,
return 0x0000_0000 and
vxcvi_flag is set to 1.

Otherwise, return the value x in 32-bit unsigned integer format.

ConvertSWtoFP(x)
x is a 32-bit signed integer value.

Return the value x converted to floating-point format having unbounded significand precision.

ConvertUDtoFP(x)
x is a 64-bit unsigned integer value.

Return the value x converted to floating-point format having unbounded significand precision.

ConvertUWtoFP(x)
x is a 32-bit unsigned integer value.

Return the value x converted to floating-point format having unbounded significand precision.

DivideDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is a Zero, `vxzdz_flag` is set to 1.

If x is a finite, nonzero value and y is a Zero, `zx_flag` is set to 1.

If x is an Infinity and y is an Infinity, `vxidi_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is a Zero, return the standard QNaN.
Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.

DivideSP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is a Zero, `vxzdz_flag` is set to 1.

If x is a finite, nonzero value and y is a Zero, `zx_flag` is set to 1.

If x is an Infinity and y is an Infinity, `vxidi_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is a Zero, return the standard QNaN.
Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.

DenormDP(x)

x is a floating-point value having unbounded range and precision.

Return the value x with its significand shifted right by a number of bits equal to the difference of the -1022 and the unbiased exponent of x, and its unbiased exponent set to -1022.

DenormSP(x)

x is a floating-point value having unbounded range and precision.

Return the value x with its significand shifted right by a number of bits equal to the difference of the -126 and the unbiased exponent of x, and its unbiased exponent set to -126.

IsInf(x)

Return 1 if x is an Infinity, otherwise return 0.

IsNaN(x)

Return 1 if x is either an SNaN or a QNaN, otherwise return 0.

IsNeg(x)
Return 1 if x is a negative, nonzero value, otherwise return 0.

IsSNaN(x)
Return 1 if x is an SNaN, otherwise return 0.

IsZero(x)
Return 1 if x is a Zero, otherwise return 0.

MaximumDP(x,y)
x and y are double-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than –0.

MaximumSP(x,y)
x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than –0.

MEM(x,y)
Contents of a sequence of y bytes of storage. The sequence depends on the endianness of the storage access as follows.
- For big-endian storage accesses, the sequence starts with the byte at address x and ends with the byte at address x+y-1.
- For little-endian storage accesses, the sequence starts with the byte at address x+y-1 and ends with the byte at address x.

MinimumDP(x,y)
x and y are double-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where –0 is considered less than +0.

MinimumSP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where –0 is considered less than +0.

MultiplyAddDP(x,y,z)

x, y and z are double-precision floating-point values.

If x, y or z is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y, is an Infinity or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If the product of x and y is an Infinity and z is an Infinity of the opposite sign, `vxisi_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if z is a QNaN, return z.
Otherwise, if z is an SNaN, return z represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.

MultiplyAddSP(x,y,z)

x, y and z are single-precision floating-point values.

If x, y or z is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If the product of x and y is an Infinity and z is an Infinity of the opposite sign, `vxisi_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if z is a QNaN, return z.
Otherwise, if z is an SNaN, return z represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.

MultiplyDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is as Infinity or x is a Infinity and y is an Zero, return the standard QNaN.
Otherwise, return the normalized product of x and y, having unbounded range and precision.

MultiplySP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is as Infinity or x is a Infinity and y is an Zero, return the standard QNaN.
Otherwise, return the normalized product of x and y, having unbounded range and precision.

NegateDP(x)

If the double-precision floating-point value x is a NaN, return x.
Otherwise, return the double-precision floating-point value x with its sign bit complemented.

NegateSP(x)

If the single-precision floating-point value x is a NaN, return x.
Otherwise, return the single-precision floating-point value x with its sign bit complemented.

ReciprocalEstimateDP(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a Zero, return an Infinity with the sign of x.
Otherwise, if x is an Infinity, return a Zero with the sign of x.
Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.

ReciprocalEstimateSP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a Zero, return an Infinity with the sign of x.
Otherwise, if x is an Infinity, return a Zero with the sign of x.
Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.

ReciprocalSquareRootEstimateDP(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a negative, nonzero number, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

ReciprocalSquareRootEstimateSP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a negative, nonzero number, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

reset_xflags()

`vxsnan_flag` is set to 0.
`vximz_flag` is set to 0.
`vxidi_flag` is set to 0.
`vxisi_flag` is set to 0.
`vxzdz_flag` is set to 0.
`vxsqrt_flag` is set to 0.
`vxcvi_flag` is set to 0.
`vxvc_flag` is set to 0.
`ox_flag` is set to 0.
`ux_flag` is set to 0.
`xx_flag` is set to 0.
`zx_flag` is set to 0.

RoundToDP(x,y)

x is a 2-bit unsigned integer specifying one of four rounding modes.

0b00	Round to Nearest Even
0b01	Round towards Zero
0b10	Round towards +Infinity
0b11	Round towards - Infinity

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to double-precision under control of the rounding mode specified by x.

```
if IsQNaN(y) then return ConvertFPtoDP(y)
if IsInf(y)  then return ConvertFPtoDP(y)
if IsZero(y) then return ConvertFPtoDP(y)
if y<Nmin then do
   if UE=0 then do
      if x=0b00 then r ← RoundToDPNearEven( DenormDP(y) )
      if x=0b01 then r ← RoundToDPTrunc( DenormDP(y) )
      if x=0b10 then r ← RoundToDPCeil( DenormDP(y) )
      if x=0b11 then r ← RoundToDPFloor( DenormDP(y) )
      ux_flag ← xx_flag
      return(ConvertFPtoDP(r))
   end
   else do
      y ← Scalb(y,+1536)
      ux_flag ← 1
   end
end
if x=0b00 then r ← RoundToDPNearEven(y)
if x=0b01 then r ← RoundToDPTrunc(y)
if x=0b10 then r ← RoundToDPCeil(y)
if x=0b11 then r ← RoundToDPFloor(y))
if r>Nmax then do
   if OE=0 then do
      if x=0b00 then r ← sign ? -Inf : +Inf
      if x=0b01 then r ← sign ? -Nmax : +Nmax
      if x=0b10 then r ← sign ? -Nmax : +Inf
      if x=0b11 then r ← sign ? -Inf : +Nmax
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(ConvertFPtoDP(r))
   end
   else do
      r ← Scalb(r,-1536)
      ox_flag ← 1
   end
end
return(ConvertFPtoDP(r))
```

RoundToDPCeil(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPFloor(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but double-precision significand precision that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerCeil(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the smallest double-precision floating-point integer value that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerFloor(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest double-precision floating-point integer value that is lesser or equal in value to x

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerNearAway(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest double-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest double-precision floating-point integer that is greater or equal in value to x-0.5 if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerNearEven(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the double-precision floating-point integer value that is nearest in value to x (in case of a tie, the double-precision floating-point integer value with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerTrunc(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest double-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest double-precision floating-point integer value that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPNearEven(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the floating-point number having unbounded exponent range but double-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but double-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPTrunc(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but double-precision significand precision that is lesser or equal in value to x if x>0, or the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSP(x,y)

x is a 2-bit unsigned integer specifying one of four rounding modes.

0b00	Round to Nearest Even
0b01	Round towards Zero
0b10	Round towards +Infinity
0b11	Round towards - Infinity

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to single-precision under control of the rounding mode specified by x.

```
if IsQNaN(y) then return ConvertFPtoSP(y)
if IsInf(y)  then return ConvertFPtoSP(y)
if IsZero(y) then return ConvertFPtoSP(y)
if y<Nmin then do
   if UE=0 then do
      if x=0b00 then r ← RoundToSPNearEven( DenormSP(y) )
      if x=0b01 then r ← RoundToSPTrunc( DenormSP(y) )
      if x=0b10 then r ← RoundToSPCeil( DenormSP(y) )
      if x=0b11 then r ← RoundToSPFloor( DenormSP(y) )
      ux_flag ← xx_flag
      return(ConvertFPtoSP(r))
   end
   else do
      y ← Scalb(y,+192)
      ux_flag ← 1
   end
end
if x=0b00 then r ← RoundToSPNearEven(y)
if x=0b01 then r ← RoundToSPTrunc(y)
if x=0b10 then r ← RoundToSPCeil(y)
if x=0b11 then r ← RoundToSPFloor(y))
if r>Nmax then do
   if OE=0 then do
      if x=0b00 then r ← sign ? -Inf : +Inf
      if x=0b01 then r ← sign ? -Nmax : +Nmax
      if x=0b10 then r ← sign ? -Nmax : +Inf
      if x=0b11 then r ← sign ? -Inf : +Nmax
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(ConvertFPtoSP(r))
   end
   else do
      r ← Scalb(r,-192)
      ox_flag ← 1
   end
end
return(ConvertFPtoSP(r))
```

RoundToSPCeil(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the smallest floating-point number having unbounded exponent range but single-precision significand precision that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPFloor(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but single-precision significand precision that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerCeil(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the smallest single-precision floating-point integer value that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerFloor(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest single-precision floating-point integer value that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerNearAway(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return x if x is a floating-point integer; otherwise return the largest single-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest single-precision floating-point integer value that is greater or equal in value to x-0.5 if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerNearEven(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return x if x is a floating-point integer; otherwise return the single-precision floating-point integer value that is nearest in value to x (in case of a tie, the single-precision floating-point integer value with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerTrunc(x)

x is a single-precision floating-point value.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN, and `vxsnan_flag` is set to 1.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest single-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point integer value that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPNearEven(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the floating-point number having unbounded exponent range but single-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but single-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPTrunc(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but single-precision significand precision that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point number that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

Scalb(x,y)

x is a floating-point value having unbounded range and precision.

y is a signed integer.

Result of multiplying the floating-point value x by 2^y.

SetFX(x)

x is one of the exception flags in the FPSCR.

If the contents of x is 0, FX and x are set to 1.

SquareRootDP(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a negative, nonzero value, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return the normalized square root of x, having unbounded range and precision.

SquareRootSP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a negative, nonzero value, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return the normalized square root of x, having unbounded range and precision.

7.6.3 VSX Instruction Descriptions

Load VSX Scalar Doubleword Indexed XX1-form

lxsdx XT,RA,RB

31	T	RA	RB	588	TX
0	6	11	16	21	31

```
XT       ← TX || T
a{0:63}  ← (RA=0) ? 0 : GPR[RA]
EA{0:63} ← a + GPR[RB]
VSR[XT]  ← MEM(EA,8) || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of the doubleword in storage at address EA are placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

VSR Data Layout for lxsdx

tgt = VSR[XT]

MEM(EA,8)	undefined
0	64 127

Load VSX Scalar as Integer Word Algebraic Indexed XX1-form

lxsiwax XT,RA,RB

31	T	RA	RB	76	TX
0	6	11	16	21	31

```
EA ← ( (RA=0) ? 0 : GPR[RA] ) + GPR[RB]
VSR[32×TX+T].doubleword[0] ← ExtendSign(MEM(EA,4))
VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The 32-bit signed integer value in the word in storage at address EA is sign-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

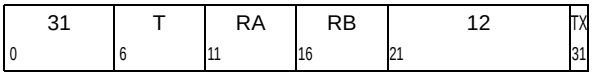
VSR Data Layout for lxsiwax

tgt = VSR[XT]

SD	undefined
0	64 127

Load VSX Scalar as Integer Word and Zero Indexed XX1-form

lxsiwzx XT,RA,RB




```
EA ← ( (RA=0) ? 0 : GPR[RA] ) + GPR[RB]
VSR[32×TX+T].doubleword[0] ← ExtendZero(MEM(EA,4))
VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The 32-bit unsigned integer value in the word in storage at address EA is zero-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

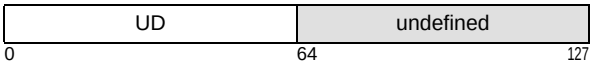
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

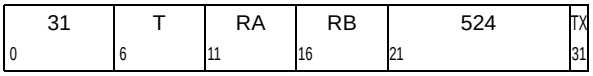
VSR Data Layout for lxsiwzx

tgt = VSR[XT]



Load VSX Scalar Single-Precision Indexed XX1-form

lxsspx XT,RA,RB




```
EA ← ( (RA=0) ? 0 : GPR[RA] ) + GPR[RB]
VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(MEM(EA,4))
VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The single-precision floating-point value in the word in storage at address EA is placed in doubleword element 0 of VSR[XT] in double-precision format.

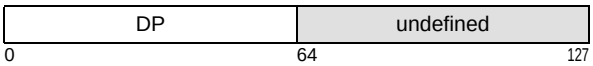
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

VSR Data Layout for lxsspx

tgt = VSR[XT]



*Load VSX Vector Doubleword*2 Indexed XX1-form*

lxvd2x XT,RA,RB

31	T	RA	RB	844	TX
0	6	11	16	21	31

```
XT              ← TX || T
a{0:63}         ← (RA=0) ? 0 : GPR[RA]
EA{0:63}        ← a + GPR[RB]
VSR[XT]{0:63}   ← MEM(EA,8)
VSR[XT]{64:127} ← MEM(EA+8,8)
```

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of the doubleword in storage at address EA are placed into doubleword element 0 of VSR[XT].

The contents of the doubleword in storage at address EA+8 are placed into doubleword element 1 of VSR[XT].

Special Registers Altered

None

VSR Data Layout for lxvd2x

tgt = VSR[XT]

MEM(EA,8)	MEM(EA+8,8)
0	64 127

Load VSX Vector Doubleword & Splat Indexed XX1-form

lxvdsx XT,RA,RB (0x7C00_0298)

31	T	RA	RB	332	TX
0	6	11	16	21	31

```
XT              ← TX || T
a{0:63}         ← (RA=0) ? 0 : GPR[RA]
EA{0:63}        ← a + GPR[RB]
load_data{0:63} ← MEM(EA,8)
VSR[XT]{0:63}   ← load_data
VSR[XT]{64:127} ← load_data
```

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of the doubleword in storage at address EA are copied into doubleword elements 0 and 1 of VSR[XT].

Special Registers Altered

None

VSR Data Layout for lxvdsx

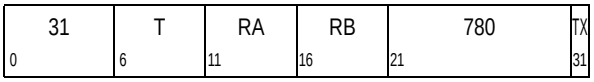
tgt = VSR[XT]

MEM(EA,8)	MEM(EA,8)
0	64 127

Extended Mnemonic		Equivalent To		Usage
lxvx	XT,RA,RB	lxvd2x	XT,RA,RB	can be used for vector load operations using Big-Endian byte-ordering, independent of element size

*Load VSX Vector Word*4 Indexed XX1-form*

lxvw4x XT,RA,RB




```
XT              ← TX || T
a{0:63}         ← (RA=0) ? 0 : GPR[RA]
EA{0:63}        ← a + GPR[RB]
VSR[XT]{0:31}   ← MEM(EA,4)
VSR[XT]{32:63}  ← MEM(EA+4,4)
VSR[XT]{64:95}  ← MEM(EA+8,4)
VSR[XT]{96:127} ← MEM(EA+12,4)
```

Let XT be the value TX concatenated with T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of the word in storage at address EA are placed into word element 0 of VSR[XT].

The contents of the word in storage at address EA+4 are placed into word element 1 of VSR[XT].

The contents of the word in storage at address EA+8 are placed into word element 2 of VSR[XT].

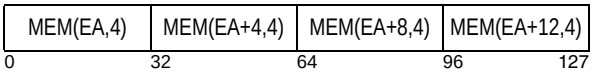
The contents of the word in storage at address EA+12 are placed into word element 3 of VSR[XT].

Special Registers Altered

None

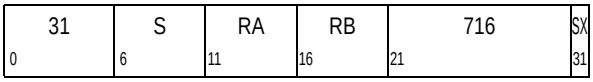
VSR Data Layout for lxvw4x

tgt = VSR[XT]



Store VSX Scalar Doubleword Indexed XX1-form

stxsdx XS,RA,RB




```
XS          ← SX || S
a{0:63}     ← (RA=0) ? 0 : GPR[RA]
EA{0:63}    ← a + GPR[RB]
MEM(EA,8)   ← VSR[XS]{0:63}
```

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

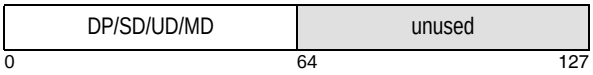
The contents of doubleword element 0 of VSR[XS] are placed in the doubleword in storage at address EA.

Special Registers Altered

None

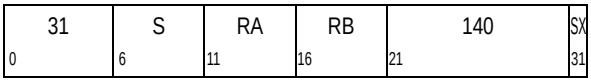
VSR Data Layout for stxsdx

src = VSR[XS]



Store VSX Scalar as Integer Word Indexed XX1-form

stxsiwx XS,RA,RB




```
EA ← ( (RA=0) ? 0 : GPR[RA] ) + GPR[RB]
MEM(EA,4) ← VSR[32×SX+S].word[1]
```

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

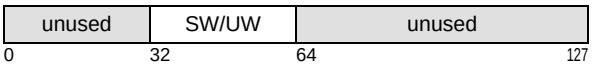
The contents of word element 1 of VSR[XS] is placed in the word in storage at address EA in single-precision format.

Special Registers Altered

None

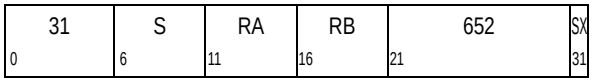
VSR Data Layout for stxsspx

src = VSR[XS]



Store VSX Scalar Single-Precision Indexed XX1-form

stxsspx XS,RA,RB




```
EA ← ( (RA=0) ? 0 : GPR[RA] ) + GPR[RB]
MEM(EA,4) ← ConvertSP64toSP(VSR[32×SX+S].doubleword[0])
```

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

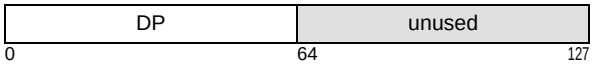
The single-precision value in double-precision floating-point format in doubleword element 0 of VSR[XS] is placed in the word in storage at address EA in single-precision format.

Special Registers Altered

None

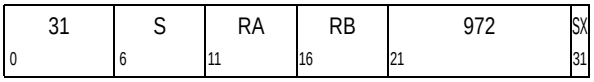
VSR Data Layout for stxsspx

src = VSR[XS]



*Store VSX Vector Doubleword*2 Indexed XX1-form*

stxvd2x XS,RA,RB




```
XS          ← SX || S
a{0:63}     ← (RA=0) ? 0 : GPR[RA]
EA{0:63}    ← a + GPR[RB]
MEM(EA,8)   ← VSR[XS]{0:63}
MEM(EA+8,8) ← VSR[XS]{64:127}
```

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of doubleword element 0 of VSR[XS] are placed in the doubleword in storage at address EA.

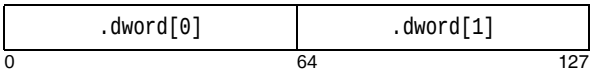
The contents of doubleword element 1 of VSR[XS] are placed in the doubleword in storage at address EA+8.

Special Registers Altered

None

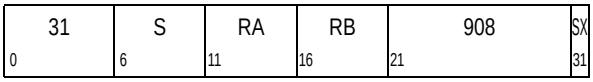
VSR Data Layout for stxvd2x

src = VSR[XS]



*Store VSX Vector Word*4 Indexed XX1-form*

stxvw4x XS,RA,RB




```
XS          ← SX || S
a{0:63}     ← (RA=0) ? 0 : GPR[RA]
EA{0:63}    ← a + GPR[RB]
MEM(EA,4)   ← VSR[XS]{0:31}
MEM(EA+4,4) ← VSR[XS]{32:63}
MEM(EA+8,4) ← VSR[XS]{64:95}
MEM(EA+12,4) ← VSR[XS]{96:127}
```

Let XS be the value SX concatenated with S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

The contents of word element 0 of VSR[XS] are placed in the word in storage at address EA.

The contents of word element 1 of VSR[XS] are placed in the word in storage at address EA+4.

The contents of word element 2 of VSR[XS] are placed in the word in storage at address EA+8.

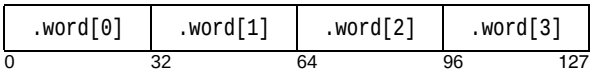
The contents of word element 3 of VSR[XS] are placed in the word in storage at address EA+12.

Special Registers Altered

None

VSR Data Layout for stxvw4x

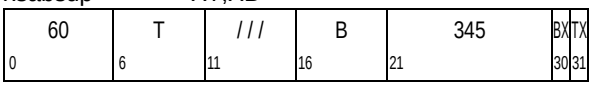
src = VSR[XS]



Extended Mnemonic		Equivalent To		Usage
stxvx	XS,RA,RB	stxvd2x	XS,RA,RB	can be used for vector store operations using Big-Endian byte-ordering, independent of element size

VSX Scalar Absolute Value Double-Precision XX2-form

xsabsdp XT,XB




```
XT           ← TX || T
XB           ← BX || B
result{0:63} ← 0b0 || VSR[XB]{1:63}
VSR[XT]      ← result || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

The absolute value of the double-precision floating-point operand in doubleword element 0 of VSR[XB] is placed into doubleword element 0 of VSR[XT] in double-precision format.

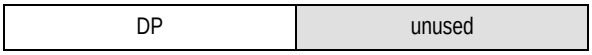
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

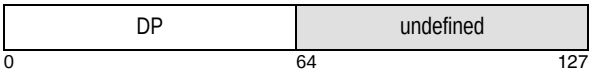
None

VSR Data Layout for xsabsdp

src = VSR[XB]



tgt = VSR[XT]



VSX Scalar Add Double-Precision XX3-form

xsadddp XT,XA,XB

60	T	A	B	32	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
v{0:inf}     ← AddDP(src1,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassSP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src2` is added[1] to `src1`, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 48, “Actions for xsadddp,” on page 401.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

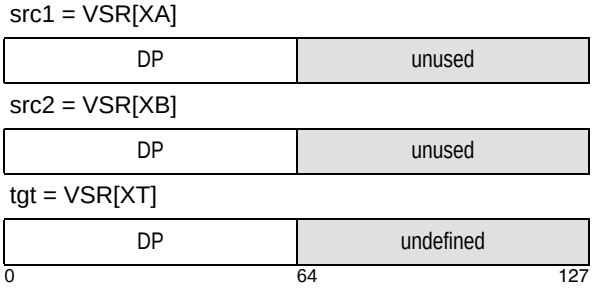
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

VSR Data Layout for xsadddp



1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 48.Actions for xsadddp

Range of v	Case	Rounding Mode			
		RTN	RTZ	RTP	RTM
v is a QNaN	Special	r ← v	r ← v	r ← v	r ← v
v = -Infinity	Special	r ← v	r ← v	r ← v	r ← v
-Infinity < v ≤ -(Nmax + 1ulp)	Overflow	q ← Rnd(v) r ← -Infinity	q ← Rnd(v) r ← -Nmax	q ← Rnd(v) r ← -Nmax	q ← Rnd(v) r ← -Infinity
-(Nmax + 1ulp) < v ≤ -(Nmax + ½ulp)	Overflow	q ← Rnd(v) r ← -Infinity	—	—	q ← Rnd(v) r ← -Infinity
	Normal	—	r ← -Nmax	r ← -Nmax	—
-(Nmax + ½ulp) < v < -Nmax	Overflow	–	—	—	q ← Rnd(v) r ← -Infinity
	Normal	r ← -Nmax	r ← -Nmax	r ← -Nmax	—
v = -Nmax	Normal	r ← -Nmax	r ← -Nmax	r ← -Nmax	r ← -Nmax
-Nmax < v < -Nmin	Normal	r ← Rnd(v)	r ← Rnd(v)	r ← Rnd(v)	r ← Rnd(v)
v = -Nmin	Normal	r ← -Nmin	r ← -Nmin	r ← -Nmin	r ← -Nmin
-Nmin < v < -Zero	Tiny	q ← Rnd(v) r ← Rnd(Den(v))	q ← Rnd(v) r ← Rnd(Den(v))	q ← Rnd(v) r ← Rnd(Den(v))	q ← Rnd(v) r ← Rnd(Den(v))
v = -Zero	Special	r ← v	r ← v	r ← v	r ← v
v = Rezd	Special	r ← +Zero	r ← +Zero	r ← +Zero	r ← -Zero
v = +Zero	Special	r ← v	r ← v	r ← v	r ← v
+Zero < v < +Nmin	Tiny	q ← Rnd(v) r ← Rnd(Den(v))	q ← Rnd(v) r ← Rnd(Den(v))	q ← Rnd(v) r ← Rnd(Den(v))	q ← Rnd(v) r ← Rnd(Den(v))
v = +Nmin	Normal	r ← +Nmin	r ← +Nmin	r ← +Nmin	r ← +Nmin
+Nmin < v < +Nmax	Normal	r ← Rnd(v)	r ← Rnd(v)	r ← Rnd(v)	r ← Rnd(v)
v = +Nmax	Normal	r ← +Nmax	r ← +Nmax	r ← +Nmax	r ← +Nmax
+Nmax < v < +(Nmax + ½ulp)	Overflow	—	—	q ← Rnd(v) r ← +Infinity	—
	Normal	r ← +Nmax	r ← +Nmax	—	r ← +Nmax
+(Nmax + ½ulp) ≤ v < +(Nmax + 1ulp)	Overflow	q ← Rnd(v) r ← +Infinity	—	q ← Rnd(v) r ← +Infinity	—
	Normal	—	r ← +Nmax	—	r ← +Nmax
+(Nmax + 1ulp) ≤ v < +Infinity	Overflow	q ← Rnd(v) r ← -Infinity	q ← Rnd(v) r ← +Nmax	q ← Rnd(v) r ← -Infinity	q ← Rnd(v) r ← +Nmax
v = +Infinity	Special	r ← v	r ← v	r ← v	r ← v

Explanation:

–	This situation cannot occur.
v	The precise intermediate result defined in the instruction having unbounded range and precision.
Den(x)	The value x is denormalized. The significand is shifted left by the amount of the difference between Emin for the target precision (that is, -1022 for double-precision, -126 for single-precision) and the unbiased exponent of x. The unbiased exponent of the denormalized value is Emin. The significand of the denormalized value has unbounded significand precision.
Rezd	Exact-zero-difference result. Applies only to add operations involving source operands having the same magnitude and different signs.
Rnd(x)	The significand of x is rounded to the target precision according to the rounding mode specified in $FPSCR_{RN}$. Exponent range of the rounded result is unbounded. See Section 7.3.2.6.
Nmax	Largest (in magnitude) representable normalized number in the target precision format.
Nmin	Smallest (in magnitude) representable normalized number in the target precision format.
ulp	Least significant bit in the target precision format's significand (Unit in the Last Position).
RTN	Round To Nearest, ties to Even.
RTZ	Round Toward Zero.
RTP	Round Toward +infinity.
RTM	Round Toward –infinity.

Table 49.Floating-Point Intermediate Result Handling

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	0	0	0	0	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0
	–	–	–	0	–	–	–	–	–	–	–	1	–	–	–	–	T(r), FI←0, FR←0, fx(ZX)
	–	–	–	1	–	–	–	–	–	–	–	1	–	–	–	–	fx(ZX), error()
	0	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSQRT)
	0	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXZDZ)
	0	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXIDI)
	0	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSQRT)
	1	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	fx(VXZDZ), error()
	1	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	fx(VXIDI), error()
	1	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()

Explanation:

–	The results do not depend on this condition.
ClassFP(x)	Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.
fx(x)	FX is set to 1 if x=0. x is set to 1.
β	Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{192}$ for single-precision.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
T(x)	The value x is placed in element 0 of VSR[XT] in the target precision format. The contents of the remaining element(s) of VSR[XT] are undefined.
UX	Floating-Point Underflow exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXSQRT	Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, $FPSCR_{VXSQRT}$.
VXIDI	Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, $FPSCR_{VXIDI}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
VXZDZ	Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, $FPSCR_{VXZDZ}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
ZX	Floating-Point Zero Divide Exception status flag, $FPSCR_{ZX}$.

Table 50.Scalar Floating-Point Final Result

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Normal	–	–	–	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0
	–	–	–	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←0, fx(XX)
	–	–	–	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←1, fx(XX)
	–	–	–	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←0, fx(XX), error()
	–	–	–	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←1, fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←?, fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←?, fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	no	–	T(q÷β), FPRF←ClassFP(q÷β), FI←0, FR←0, fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	no	T(q÷β), FPRF←ClassFP(q÷β), FI←1, FR←0, fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	yes	T(q÷β), FPRF←ClassFP(q÷β), FI←1, FR←1, fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0
	–	–	0	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←0, fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←1, fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←0, fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), FPRF←ClassFP(r), FI←1, FR←1, fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	no	–	T(q×β), FPRF←ClassFP(q×β), FI←0, FR←0, fx(UX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	no	T(q×β), FPRF←ClassFP(q×β), FI←1, FR←0, fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	yes	T(q×β), FPRF←ClassFP(q×β), FI←1, FR←1, fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
ClassFP(x)	Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.
fx(x)	FX is set to 1 if x=0. x is set to 1.
β	Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{192}$ for single-precision.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
T(x)	The value x is placed in element 0 of VSR[XT] in the target precision format. The contents of the remaining element(s) of VSR[XT] are undefined.
UX	Floating-Point Underflow exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXSQRT	Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, $FPSCR_{VXSQRT}$.
VXIDI	Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, $FPSCR_{VXIDI}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
VXZDZ	Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, $FPSCR_{VXZDZ}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
ZX	Floating-Point Zero Divide Exception status flag, $FPSCR_{ZX}$.

Table 50.Scalar Floating-Point Final Result (Continued)

VSX Scalar Add Single-Precision XX3-form

xsaddsp XT,XA,XB

60	T	A	B	0	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

src1   ← VSR[32×AX+A].doubleword[0]
src2   ← VSR[32×BX+B].doubleword[0]
v      ← AddDP(src1,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConverSPtoDP(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 51, “Actions for xsaddsp,” on page 406.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

VSR Data Layout for xsaddsp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 51.Actions for xsaddsp

VSX Scalar Compare Ordered Double-Precision XX3-form

xscmpodp BF,XA,XB

60	BF	/ /	A	B	43	AX	BX	/
0	6	9	11	16	21	29	30	31

```
XA          ← AX || A
XB          ← BX || B
reset_xflags()
src1        ← VSR[XA]{0:63}
src2        ← VSR[XB]{0:63}

if( IsSNaN(src1) | IsSNaN(src2) ) then do
   vxsnan_flag ← 0b1
   if(VE=0) then vxvc_flag ← 0b1
end
else if( IsQNaN(src1) | IsQNaN(src2) ) then vxvc_flag = 0b1

FL          ← CompareLTDP(src1,src2)
FG          ← CompareGTDP(src1,src2)
FE          ← CompareEQDP(src1,src2)
FU          ← IsNAN(src1) | IsNAN(src2)
CR[BF]      ← FL || FG || FE || FU
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxvc_flag)   then SetFX(VXVC)
```

Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src1` is compared to `src2`.

Zeros of same or opposite signs compare equal.

Infinities of same signs compare equal.

See Table 52, “Actions for xscmpodp - Part 1: Compare Ordered,” on page 408.

The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set, and Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, VXVC is set.

See Table 53, “Actions for xscmpodp - Part 2: Result,” on page 408.

Special Registers Altered

CR[BF]
FPCC FX VXSNAN VXVC

VSR Data Layout for xscmpodp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	undefined

0 64 127

		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
	–NZF	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
	–Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
	+Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
	+NZF	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
	+Infinity	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
	QNaN	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
	SNaN	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
NZF	Nonzero finite number.
C(x,y)	The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results. 0b1000 when x is greater than y 0b0100 when x is less than y 0b0010 when x is equal to y
cc	The 4-bit result compare code.

Table 52.Actions for xscmpodp - Part 1: Compare Ordered

VE	vxsnan_flag	vxvc_flag	Returned Results and Status Setting
–	0	0	FPCC←cc, CR[BF]←cc
0	0	1	FPCC←cc, CR[BF]←cc, fx(VXVC)
0	1	0	FPCC←cc, CR[BF]←cc, fx(VXSNAN)
0	1	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN), fx(VXVC)
1	0	1	FPCC←cc, CR[BF]←cc, fx(VXVC), error()
1	1	–	FPCC←cc, CR[BF]←cc, fx(VXSNAN), error()

Explanation:

–	The results do not depend on this condition.
cc	The 4-bit result as defined in Table 52.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
FX	Floating-Point Summary Exception status flag, $FPSCR_{FX}$.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. See Section 7.4.1.
VXC	Floating-Point Invalid Operation Exception (Invalid Compare) status flag, $FPSCR_{VXVC}$. See Section 7.4.1.

Table 53.Actions for xscmpodp - Part 2: Result

VSX Scalar Compare Unordered Double-Precision XX3-form

xscmpudp BF,XA,XB

60	BF	/ /	A	B	35	AX	BX	/
0	6	9	11	16	21	29	30	31

```
XA     ← AX || A
XB     ← BX || B
reset_xflags()
src1   ← VSR[XA]{0:63}
src2   ← VSR[XB]{0:63}

if( IsSNaN(src1) | IsSNaN(src2) ) then vxsnan_flag ← 1

FL     ← CompareLTDP(src1,src2)
FG     ← CompareGTDP(src1,src2)
FE     ← CompareEQDP(src1,src2)
FU     ← IsNAN(src1) | IsNAN(src2)
CR[BF] ← FL || FG || FE || FU
if(vxsnan_flag) then SetFX(VXSNAN)
```

Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src1` is compared to `src2`.

Zeros of same or opposite signs compare equal equal.

Infinities of same signs compare equal.

See Table 54, “Actions for xscmpudp - Part 1: Compare Unordered,” on page 410.

The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set.

See Table 55, “Actions for xscmpudp - Part 2: Result,” on page 410.

Special Registers Altered

CR[BF]
FPCC FX VXSNAN

VSR Data Layout for xscmpudp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	undefined

0 64 127

		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	cc = 0b0010	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
	–NZF	cc = 0b0100	cc = C(src1,src2)	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
	–Zero	cc = 0b0100	cc = 0b0100	cc = 0b0010	cc = 0b0010	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
	+Zero	cc = 0b0100	cc = 0b0100	cc = 0b0010	cc = 0b0010	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
	+NZF	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = C(src1,src2)	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
	+Infinity	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0010	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
	QNaN	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
	SNaN	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
NZF	Nonzero finite number.
C(x,y)	The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results. 0b1000 when x is greater than y 0b0100 when x is less than y 0b0010 when x is equal to y
cc	The 4-bit result compare code.

Table 54.Actions for xscmpudp - Part 1: Compare Unordered

VE	vxsnan_flag	Returned Results and Status Setting
–	0	FPCC←cc, CR[BF]←cc
0	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN)
1	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN), error()

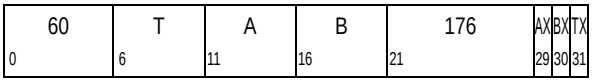
Explanation:

–	The results do not depend on this condition.
cc	The 4-bit result as defined in Table 54.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
FX	Floating-Point Summary Exception status flag, $FPSCR_{FX}$.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. See Section 7.4.1.

Table 55.Actions for xscmpudp - Part 2: Result

VSX Scalar Copy Sign Double-Precision XX3-form

xscpsgndp XT,XA,XB




```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
result{0:63} ← VSR[XA]{0} || VSR[XB]{1:63}
VSR[XT]      ← result || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Bit 0 of VSR[XT] is set to the contents of bit 0 of VSR[XA].

Bits 1:63 of VSR[XT] are set to the contents of bits 1:63 of VSR[XB].

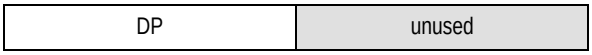
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

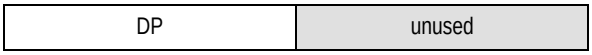
None

VSR Data Layout for xscpsgndp

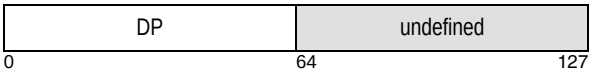
src1 = VSR[XA]



src2 = VSR[XB]

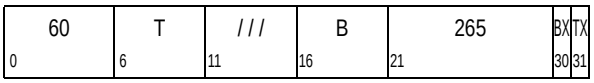


tgt = VSR[XT]



VSX Scalar round Double-Precision to single-precision and Convert to Single-Precision format XX2-form

xscvdpsp XT,XB




```
reset_xflags()
src ← VSR[32×BX+B].dword[0]
result ← ConvertDPtoSP(src)
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(xx_flag) then SetFX(FPSCR.XX)
if(ox_flag) then SetFX(FPSCR.OX)
if(ux_flag) then SetFX(FPSCR.UX)
vex_flag ← FPSCR.VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[32×TX+T].word[0] ← result
   VSR[32×TX+T].word[1] ← 0xUUUU_UUUU
   VSR[32×TX+T].word[2] ← 0xUUUU_UUUU
   VSR[32×TX+T].word[3] ← 0xUUUU_UUUU
   FPSCR.FPRF ← ClassSP(result)
   FPSCR.FR ← inc_flag
   FPSCR.FI ← xx_flag
end
else do
   FPSCR.FR ← 0b0
   FPSCR.FI ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a SNaN, the result is src converted to a QNaN (i.e., bit 12 of src is set to 1). VXSNAN is set to 1.

Otherwise, if src is a QNaN, an Infinity, or a Zero, the result is src.

Otherwise, the result is src rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element 0 of VSR[XT] in single-precision format.

The contents of word elements 1, 2, and 3 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

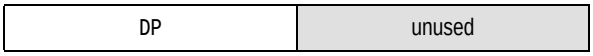
See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

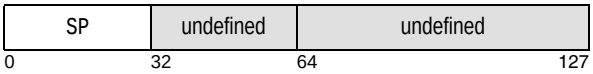
FPRF FR FI FX OX UX XX VXSNAN

VSR Data Layout for xscvdpsp

src = VSR[XB]

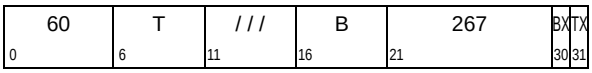


tgt = VSR[XT]



VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form

xscvdpspn XT,XB




```
reset_xflags()
src ← VSR[32×BX+B].dword[0]
result ← ConvertDPtoSP_NS(src)
VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← 0xUUUU_UUUU
VSR[32×TX+T].word[2] ← 0xUUUU_UUUU
VSR[32×TX+T].word[3] ← 0xUUUU_UUUU
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the single-precision floating-point value in doubleword element 0 of VSR[XB] represented in double-precision format.

src is placed into word element 0 of VSR[XT] in single-precision format.

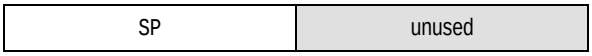
The contents of word elements 1, 2, and 3 of VSR[XT] are undefined.

Special Registers Altered

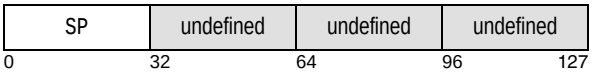
None

VSR Data Layout for xscvdpspn

src = VSR[XB]



tgt = VSR[XT]



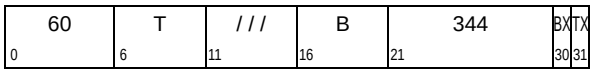
Programming Note

xscvdpsp should be used to convert a scalar double-precision value to vector single-precision format.

xscvdpspn should be used to convert a scalar single-precision value to vector single-precision format.

VSX Scalar truncate Double-Precision to integer and Convert to Signed Integer Doubleword format with Saturate XX2-form

xscvdpsxds XT,XB




```
XT          ← TX || T
XB          ← BX || B
inc_flag    ← 0b0
reset_xflags()
rnd{0:63}   ← RoundToDPIntegerTrunc(VSR[XB]{0:63})
result{0:63} ← ConvertDPtoSD(rnd)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{63}-1$, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

If a trap-enabled invalid operation exception occurs,
- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

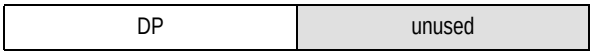
See Table 56.

Special Registers Altered

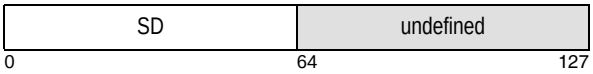
FPRF=0bUUUUU FR FI FX XX
VXSNAN VXCVI

VSR Data Layout for xscvdpsxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xscvdpsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0 Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The signed integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

Table 56.Actions for xscvdpsxds

VSX Scalar truncate Double-Precision to integer and Convert to Signed Integer Word format with Saturate XX2-form

xscvdpsxws XT,XB

60	T	///	B	88	BX	TX
0	6	11	16	21	30	31

```
XT           ← TX || T
XB           ← BX || B
inc_flag     ← 0b0
reset_xflags()
rnd{0:63}    ← RoundToDPIntegerTrunc(VSR[XB]{0:63})
result{0:31} ← ConvertDPtoSW(rnd)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← 0xUUUU_UUUU || result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If `src` is a NaN, the result is the value `0x8000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{31}-1, the result is `0x7FFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, the result is `0x8000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

If a trap-enabled invalid operation exception occurs,

– VSR[XT] and FPRF are not modified

– FR and FI are set to 0.

Otherwise,

– The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

– FPRF is set to an undefined value.

– FR is set to indicate if the result was incremented when rounded.

– FI is set to indicate the result is inexact.

See Table 57.

Special Registers Altered

FPRF=0bUUUUU FR FI FX XX
VXSNAN VXCVI

VSR Data Layout for xscvdpsxws

src = VSR[XB]

DP	unused

tgt = VSR[XT]

undefined	SW	undefined
0	32	64 127

Programming Note

xscvdpsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

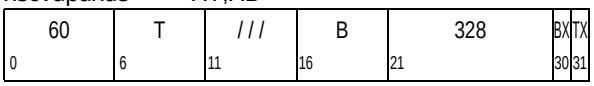
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest signed integer word value, -2^{31} (0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x7FFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The signed integer word value x is placed in word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

Table 57.Actions for xscvdpsxws

VSX Scalar truncate Double-Precision integer and Convert to Unsigned Integer Doubleword format with Saturate XX2-form

xscvdpuxds XT,XB




```
XT           ← TX || T
XB           ← BX || B
inc_flag     ← 0b0
reset_xflags()
rnd{0:63}    ← RoundToDPIntegerTrunc(VSR[XB]{0:63})
result{0:63} ← ConvertDPtoUD(rnd)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If `src` is a NaN, the result is the value `0x0000_0000_0000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is `0xFFFF_FFFF_FFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is `0x0000_0000_0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

If a trap-enabled invalid operation exception occurs,

– VSR[XT] and FPRF are not modified

– FR and FI are set to 0.

Otherwise,

– The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

– FPRF is set to an undefined value.

– FR is set to indicate if the result was incremented when rounded.

– FI is set to indicate the result is inexact.

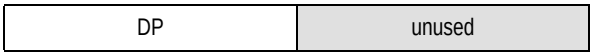
See Table 58.

Special Registers Altered

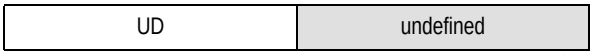
FPRF=0bUUUUU FR FI FX XX
VXSNAN VXCVI

VSR Data Layout for xscvdpuxds

src = VSR[XB]



tgt = VSR[XT]



0 64 127

Programming Note

xscvdpuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0 Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The unsigned integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

Table 58.Actions for xscvdpuxds

VSX Scalar truncate Double-Precision to integer and Convert to Unsigned Integer Word format with Saturate XX2-form

xscvdpuxws XT,XB

60	T	///	B	72	BX	TX
0	6	11	16	21	30	31

```
XT           ← TX || T
XB           ← BX || B
inc_flag     ← 0b0
reset_xflags()
rnd{0:63}    ← RoundToDPIntegerTrunc(VSR[XB]{0:63})
result{0:31} ← ConvertDPtoUW(rnd)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← 0xUUUU_UUUU || result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If `src` is a NaN, the result is the value `0x0000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{32}-1, the result is `0xFFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is `0x0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

If a trap-enabled invalid operation exception occurs,

– VSR[XT] and FPRF are not modified

– FR and FI are set to 0.

Otherwise,

– The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

– FPRF is set to an undefined value.

– FR is set to indicate if the result was incremented when rounded.

– FI is set to indicate the result is inexact.

See Table 59.

Special Registers Altered

FPRF=0bUUUUU FR FI FX XX
VXSNAN VXCVI

VSR Data Layout for xscvdpuxws

src = VSR[XB]

DP	unused

tgt = VSR[XT]

undefined	UW	undefined
0	32	64 127

Programming Note

xscvdpuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

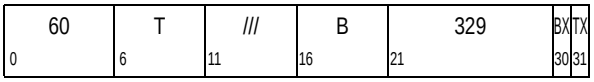
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, 2^{32}-1 (0xFFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The unsigned integer word value x is placed in word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

Table 59.Actions for xscvdpuxws

VSX Scalar Convert Single-Precision to Double-Precision format XX2-form

xscvspdp XT,XB




```
reset_xflags()
src ← VSR[32×BX+B].word[0]
result ← ConvertVectorSPtoScalarSP(src)
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPSCR.FPRF ← ClassDP(result)
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

If src is a SNaN, the result is src, converted to a QNaN (i.e., bit 9 of src set to 1). VXSNAN is set to 1.

Otherwise, the result is src.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

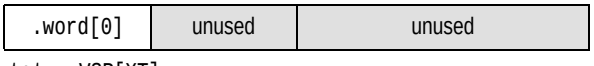
If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified, FPRF is not modified, FR is set to 0, and FI is set to 0.

Special Registers Altered

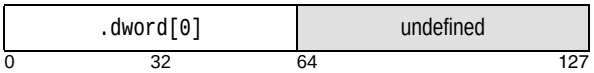
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xscvspdp

src = VSR[XB]



tgt = VSR[XT]

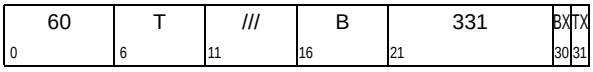


Programming Note

xscvspdp can be used to convert a single-precision value in single-precision format to double-precision format for use by category Floating-Point scalar single-precision operations.

VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form

xscvspdpn XT,XB




```
reset_xflags()
src ← VSR[32×BX+B].word[0]
result ← ConvertSPtoDP_NS(src)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

src is placed into doubleword element 0 of VSR[XT] in double-precision format.

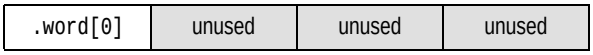
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

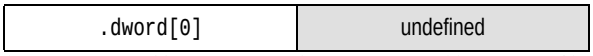
None

VSR Data Layout for xscvspdpn

src = VSR[XB]



tgt = VSR[XT]



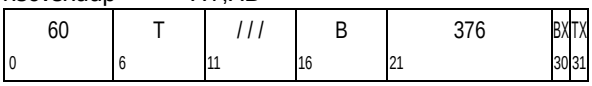
Programming Note

xscvspdp should be used to convert a vector single-precision floating-point value to scalar double-precision format.

xscvspdpn should be used to convert a vector single-precision floating-point value to scalar single-precision format.

VSX Scalar Convert Signed Integer Doubleword to floating-point format and round to Double-Precision format XX2-form

xscvsxddp XT,XB




```
XT           ← TX || T
XB           ← BX || B
reset_xflags()
v{0:inf}     ← ConvertSDtoFP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,v)
VSR[XT]      ← result || 0xUUUU_UUUU_UUUU_UUUU
if(xx_flag) then SetFX(XX)
FPRF         ← ClassDP(result)
FR           ← inc_flag
FI           ← xx_flag
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the signed integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

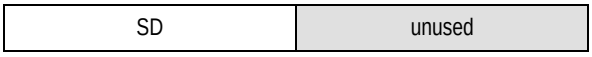
FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

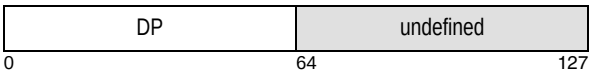
FPRF FR FI FX XX

VSR Data Layout for xscvsxddp

src = VSR[XB]

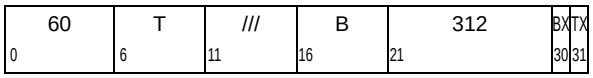


tgt = VSR[XT]



VSX Scalar Convert Signed Integer Doubleword to floating-point format and round to Single-Precision XX2-form

xscvsxdsp XT,XB




```
reset_xflags()

src    ← ConvertSDtoDP(VSR[32×BX+B].doubleword[0])
result ← RoundToSP(RN,src)
VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF ← ClassSP(result)
FR   ← inc_flag
FI   ← xx_flag
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the two's-complement integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

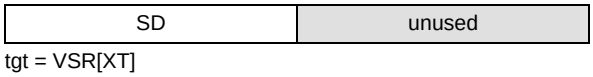
FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

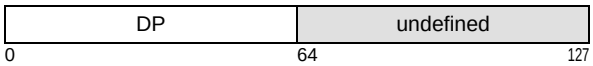
FPRF FR FI FX XX

VSR Data Layout for xscvsxdsp

src = VSR[XB]

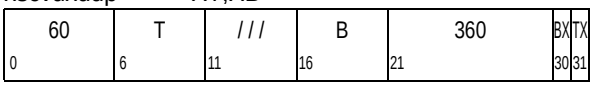


tgt = VSR[XT]



VSX Scalar Convert Unsigned Integer Doubleword to floating-point format and round to Double-Precision format XX2-form

xscvuxddp XT,XB




```
XT           ← TX || T
XB           ← BX || B
reset_xflags()
src{0:inf}   ← ConvertUDtoFP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,src)
VSR[XT]      ← result || 0xUUUU_UUUU_UUUU_UUUU
if(xx_flag) then SetFX(XX)
FPRF         ← ClassDP(result)
FR           ← inc_flag
FI           ← xx_flag
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the unsigned integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

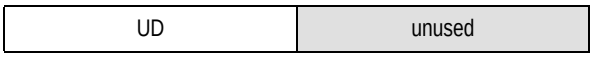
FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

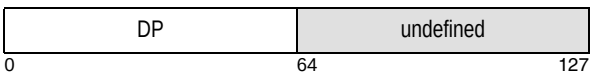
FPRF FR FI FX XX

VSR Data Layout for xscvuxddp

src = VSR[XB]

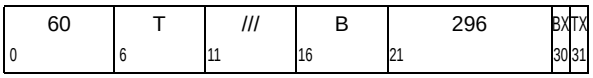


tgt = VSR[XT]



VSX Scalar Convert Unsigned Integer Doubleword to floating-point format and round to Single-Precision XX2-form

xscvuxdsp XT,XB




```
reset_xflags()

src    ← ConvertUDtoDP(VSR[32×BX+B].doubleword[0])
result ← RoundToSP(RN,src)
VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF ← ClassSP(result)
FR   ← inc_flag
FI   ← xx_flag
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the unsigned-integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

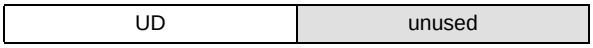
FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

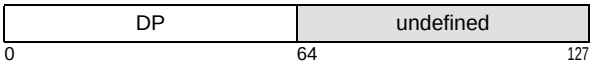
FPRF FR FI FX XX

VSR Data Layout for xscvuxdsp

src = VSR[XB]



tgt = VSR[XT]



VSX Scalar Divide Double-Precision XX3-form

xsdivdp XT,XA,XB

60	T	A	B	56	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
v{0:inf}     ← DivideFP(src1,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxidi_flag)  then SetFX(VXIDI)
if(vxzdz_flag)  then SetFX(VXZDZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
if(zx_flag)     then SetFX(ZX)
vex_flag     ← VE & (vxsnan_flag | vxidi_flag | vxzdz_flag)
zex_flag     ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[XT] = result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    = ClassDP(result)
   FR      = inc_flag
   FI      = xx_flag
end
else do
   FR      = 0b0
   FI      = 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src1` is divided[1] by `src2`, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See *Actions for xsdivdp* (p. 426).

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xsdivdp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← −Infinity	v ← −Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← −Infinity zx_flag ← 1	v ← D(src1,src2)	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← −Zero	v ← −Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← −Zero	v ← −Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Zero	v ← D(src1,src2)	v ← −Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 60.Actions for xsdivdp

VSX Scalar Divide Single-Precision XX3-form

xsdivsp XT,XA,XB

60	T	A	B	24	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

src1   ← VSR[32×AX+A].doubleword[0]
src2   ← VSR[32×BX+B].doubleword[0]
v      ← DivideDP(src1,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxidi_flag)  then SetFX(VXIDI)
if(vxzdz_flag)  then SetFX(VXZDZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
if(zx_flag)     then SetFX(ZX)

vex_flag ← VE & (vxsnan_flag|vxidi_flag|vxzdz_flag)
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided[1] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 61, “Actions for xsdivsp,” on page 428.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.
If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.
See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xsdivsp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined
0	64 127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← –Infinity zx_flag ← 1	v ← D(src1,src2)	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← –Zero	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Zero	v ← –Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Zero	v ← D(src1,src2)	v ← –Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 61.Actions for xsdivsp

VSX Scalar Multiply-Add Double-Precision XX3-form

xsmaddadp XT,XA,XB

60	T	A	B	33	AX	BX	TX
0	6	11	16	21	29	30	31

xsmaddmdp XT,XA,XB

60	T	A	B	41	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← "xsmaddadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3         ← "xsmaddadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}     ← MultiplyAddFP(src1,src3,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsmaddadp***, do the following.
- Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmaddmdp***, do the following.
- Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 62.

`src2` is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 62.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, "Scalar Floating-Point Final Result," on page 403.

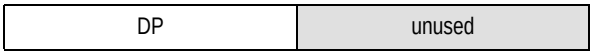
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

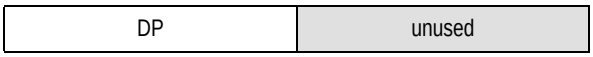
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsmadd(a|m)dp

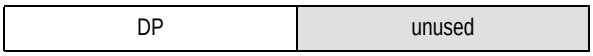
src1 = VSR[XA]



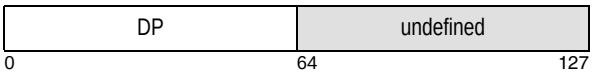
src2 = ***xsmaddadp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmaddadp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply		src3							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 62.Actions for xsmadd(a|m)dp

VSX Scalar Multiply-Add Single-Precision XX3-form

xsmaddasp XT,XA,XB

60	T	A	B	1	AX	BX	TX
0	6	11	16	21	29	30	31

xsmaddmsp XT,XA,XB

60	T	A	B	9	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if "xsmaddasp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×TX+T].doubleword[0]
   src3 ← VSR[32×BX+B].doubleword[0]
end
if "xsmaddmsp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×BX+B].doubleword[0]
   src3 ← VSR[32×TX+T].doubleword[0]
end

v      ← MultiplyAddDP(src1,src3,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For ***xsmaddasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmaddmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 63, "Actions for xsmadd(a|m)sp," on page 434.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 63, "Actions for xsmadd(a|m)sp," on page 434.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, "Scalar Floating-Point Final Result," on page 403.

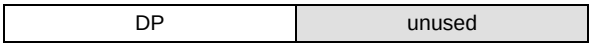
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Special Registers Altered

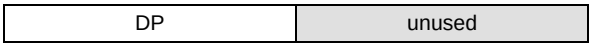
FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

VSR Data Layout for xsmadd(a|m)sp

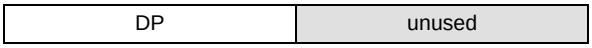
src1 = VSR[XA]



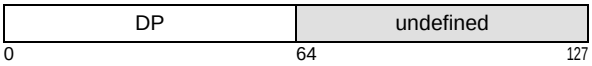
src2 = ***xsmaddasp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmaddasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 63.Actions for xsmadd(a|m)sp

VSX Scalar Maximum Double-Precision XX3-form

xsmaxdp XT,XA,XB

60	T	A	B	160	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
result{0:63} ← MaximumDP(src1,src2)
if(vxsnan_flag) then SetFX(VXSNAN)
vex_flag     ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If `src1` is greater than `src2`, `src1` is placed into doubleword element 0 of VSR[XT]. Otherwise, `src2` is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

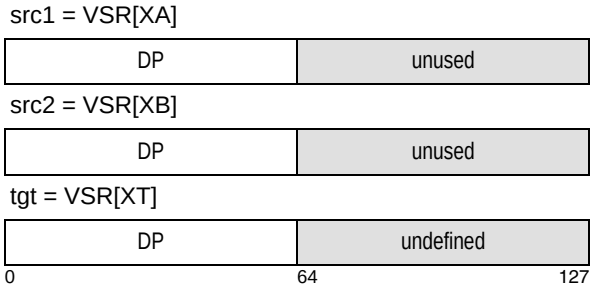
If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 64.

Special Registers Altered

FX VXSNAN

VSR Data Layout for xsmaxdp



src1 \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
–NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
–Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. If VE=1, update of VSR[XT] is suppressed.

Table 64.Actions for xsmaxdp

VSX Scalar Minimum Double-Precision XX3-form

xsmindp XT,XA,XB

60	T	A	B	168	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
result{0:63} ← MinimumDP(src1,src2)
if(vxsnan_flag) then SetFX(VXSNAN)
vex_flag     ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If `src1` is less than `src2`, `src1` is placed into doubleword element 0 of VSR[XT] in double-precision format. Otherwise, `src2` is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

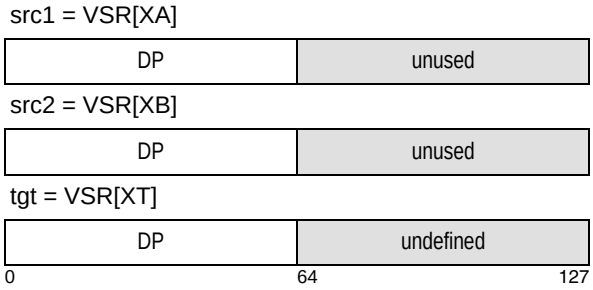
If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 65.

Special Registers Altered

FX VXSNAN

VSR Data Layout for xsmindp



		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element i ($i \in \{0,1\}$) of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. If VE=1, update of VSR[XT] is suppressed.

Table 65.Actions for xvmindp

VSX Scalar Multiply-Subtract Double-Precision XX3-form

xsmsubadp XT,XA,XB

60	T	A	B	49	AX	BX	TX
0	6	11	16	21	29	30	31

xsmsubmdp XT,XA,XB

60	T	A	B	57	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
reset_xflags()
src1        ← VSR[XA]{0:63}
src2        ← VSR[XT]{0:63}
src3        ← VSR[XB]{0:63}
src2        ← "xsmsubadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3        ← "xsmsubadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}    ← MultiplyAddDP(src1,src3,NegateDP(src2))
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For ***xsmsubadp***, do the following.

– Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].
– Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XT].
– Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmsubmdp***, do the following.

– Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].
– Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].
– Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 66.

`src2` is negated and added[2] to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized[3].

See part 2 of Table 66.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, "Scalar Floating-Point Final Result," on page 403.

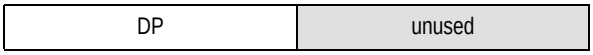
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

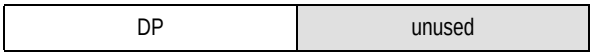
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsmsub(alm)dp

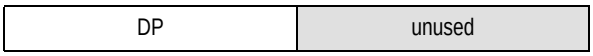
src1 = VSR[XA]



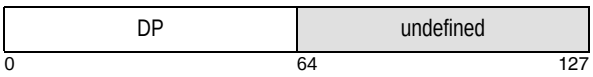
src2 = ***xsmsubadp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmsubadp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply		src3							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 66.Actions for xsmsub(a|m)dp

VSX Scalar Multiply-Subtract Single-Precision XX3-form

xsmsubasp XT,XA,XB

60	T	A	B	17	AX	BX	TX
0	6	11	16	21	29	30	31

xsmsubmsp XT,XA,XB

60	T	A	B	25	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if "xsmsubasp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×TX+T].doubleword[0]
   src3 ← VSR[32×BX+B].doubleword[0]
end
if "xsmsubmsp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×BX+B].doubleword[0]
   src3 ← VSR[32×TX+T].doubleword[0]
end

v      ← MultiplyAddDP(src1,src3,NegateDP(src2))
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For ***xsmsubasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmsubmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 67, “Actions for xsmsub(a|m)sp”.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized[3].

See part 2 of Table 67, “Actions for xsmsub(a|m)sp”.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

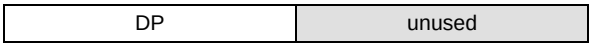
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Special Registers Altered

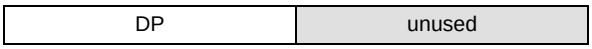
FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

VSR Data Layout for xsmsub(alm)sp

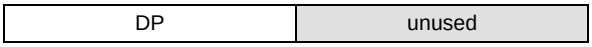
src1 = VSR[XA]



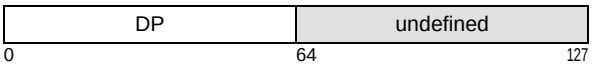
src2 = ***xsmsubasp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmsubasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmsubasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmsubmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmsubasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmsubmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 67.Actions for xsmsub(a|m)sp

VSX Scalar Multiply Double-Precision XX3-form

xsmuldp XT,XA,XB

60	T	A	B	48	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
v{0:inf}     ← MultiplyFP(src1,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vximz_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src1` is multiplied[1] by `src2`, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 68.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

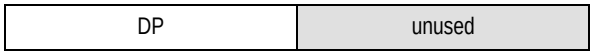
See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

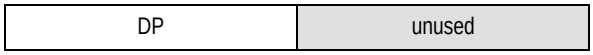
FPRF FR FI FX OX UX XX
VXSNAN VXIMZ

VSR Data Layout for xsmuldp

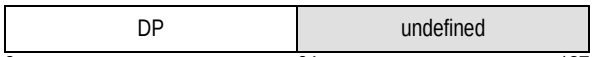
src1 = VSR[XA]



src2 = VSR[XB]



tgt = VSR[XT]



0 64 127

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← −Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← −Zero	v ← −Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← −Zero	v ← −Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← M(src1,src2)	v ← −Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← −Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 68.Actions for xsmuldp

VSX Scalar Multiply Single-Precision XX3-form

xsmulsp XT,XA,XB

60	T	A	B	16	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

src1 ← VSR[32×AX+A].doubleword[0]
src2 ← VSR[32×BX+B].doubleword[0]

v      ← MultiplyDP(src1,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied[1] by src2, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 69, “Actions for xsmulsp,” on page 448.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXIMZ

VSR Data Layout for xsmulsp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← –Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← –Zero	v ← –Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← –Zero	v ← –Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← M(src1,src2)	v ← –Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← –Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

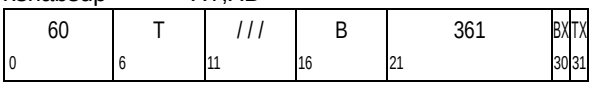
Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 69.Actions for xsmulsp

VSX Scalar Negative Absolute Value Double-Precision XX2-form

xsnabsdp XT,XB




```
XT           ← TX || T
XB           ← BX || B
result{0:63} ← 0b1 || VSR[XB]{1:63}
VSR[XT]      ← result || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

The contents of doubleword element 0 of VSR[XB], with bit 0 set to 1, is placed into doubleword element 0 of VSR[XT].

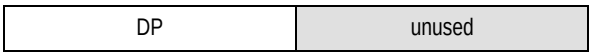
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

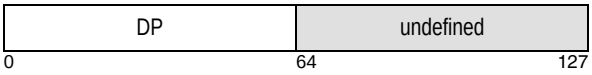
None

VSR Data Layout for xsnabsdp

src = VSR[XB]

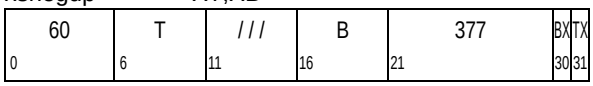


tgt = VSR[XT]



VSX Scalar Negate Double-Precision XX2-form

xsnegdp XT,XB




```
XT           ← TX || T
XB           ← BX || B
result{0:63} ← ~VSR[XB]{0} || VSR[XB]{1:63}
VSR[XT]      ← result || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

The contents of doubleword element 0 of VSR[XB], with bit 0 complemented, is placed into doubleword element 0 of VSR[XT].

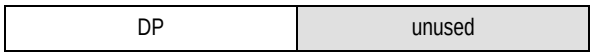
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

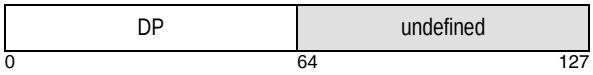
None

VSR Data Layout for xsnegdp

src = VSR[XB]



tgt = VSR[XT]



VSX Scalar Negative Multiply-Add Double-Precision XX3-form

xsnmaddadp XT,XA,XB

60	T	A	B	161	AX	BX	TX
0	6	11	16	21	29	30	31

xsnmaddmdp XT,XA,XB

60	T	A	B	169	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← "xsnmaddadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3         ← "xsnmaddadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}     ← MultiplyAddDP(src1,src3,src2)
result{0:63} ← NegateDP(RoundToDP(RN,v))
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0
   FI      ← 0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsnmaddadp***, do the following.
- Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmaddmdp***, do the following.
- Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 70.

`src2` is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 70.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

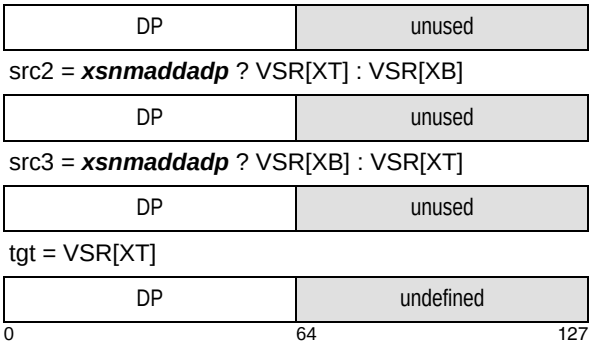
See Table 71, "Scalar Floating-Point Final Result with Negation," on page 453.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsnmadd(alm)dp



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsnmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsnmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsnmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 70.Actions for xsnmadd(a|m)dp

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	–	–	–	–	T(N(r)), FPRF←ClassFP(r), FI←0, FR←0
	0	–	–	–	–	–	–	1	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	1	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	no	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←0, FR←0
	–	–	–	–	0	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(XX)
	–	–	–	–	0	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(XX)
	–	–	–	–	1	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(XX), error()
	–	–	–	–	1	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←?, fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←?, fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	no	–	T(N(q)÷β), FPRF←ClassFP(N(q)÷β), FI←0, FR←0, fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	yes	no	T(N(q)÷β), FPRF←ClassFP(N(q)÷β), FI←1, FR←0, fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	yes	yes	T(N(q)÷β), FPRF←ClassFP(N(q)÷β), FI←1, FR←1, fx(OX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
ClassFP(x)	Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.
fx(x)	FX is set to 1 if x=0. x is set to 1.
β	Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{192}$ for single-precision.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element 0 of VSR[XT] in the target precision format. The contents of the remaining element(s) of VSR[XT] are undefined.
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Table 71.Scalar Floating-Point Final Result with Negation

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Tiny	–	–	0	–	–	–	–	–	no	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←0, FR←0
	–	–	0	–	0	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	no	–	T(N(q)×β), FPRF←ClassFP(N(q)×β), FI←0, FR←0, fx(UX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	no	T(N(q)×β), FPRF←ClassFP(N(q)×β), FI←1, FR←0, fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	yes	T(N(q)×β), FPRF←ClassFP(N(q)×β), FI←1, FR←1, fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
ClassFP(x)	Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 327.
fx(x)	FX is set to 1 if x=0. x is set to 1.
β	Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{192}$ for single-precision.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element 0 of VSR[XT] in the target precision format. The contents of the remaining element(s) of VSR[XT] are undefined.
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Table 71.Scalar Floating-Point Final Result with Negation (Continued)

VSX Scalar Negative Multiply-Add Single-Precision XX3-form

xsnmaddasp XT,XA,XB

60	T	A	B	129	AX	BX	TX
0	6	11	16	21	29	30	31

xsnmaddmsp XT,XA,XB

60	T	A	B	137	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if "xsnmaddasp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×TX+T].doubleword[0]
   src3 ← VSR[32×BX+B].doubleword[0]
end
if "xsnmaddmsp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×BX+B].doubleword[0]
   src3 ← VSR[32×TX+T].doubleword[0]
end

v      ← MultiplyAddDP(src1,src3,src2)
result ← NegateSP(RoundToSP(RN,v))

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertToSP(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For ***xsnmaddasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmaddmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 72, “Actions for xsnmadd(a|m)sp,” on page 457.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 72, “Actions for xsnmadd(a|m)sp,” on page 457.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 71, “Scalar Floating-Point Final Result with Negation,” on page 453.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

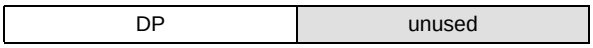
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

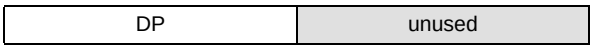
VSR Data Layout for xsnmadd(a|m)sp

src1 = VSR[XA]

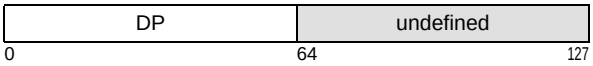
src2 = ***xsnmadda***(***dp***|***sp***) ? VSR[XT] : VSR[XB]



src3 = ***xsnmadda***(***dp***|***sp***) ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsnmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsnmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsnmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 72.Actions for xsnmadd(a|m)sp

VSX Scalar Negative Multiply-Subtract Double-Precision XX3-form

xsnmsubadp XT,XA,XB

60	T	A	B	177	AX	BX	TX
0	6	11	16	21	29	30	31

xsnmsubmdp XT,XA,XB

60	T	A	B	185	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XT]{0:63}
src3         ← VSR[XB]{0:63}
src2         ← "xsnmsubadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3         ← "xsnmsubadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}     ← MultiplyAddDP(src1,src3,NegateDP(src2))
result{0:63} ← NegateDP(RoundToDP(RN,v))
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsnmsubadp***, do the following.
- Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmsubmdp***, do the following.
- Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let `src3` be the double-precision floating-point value in doubleword element 0 of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 73.

`src2` is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 73.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 71, "Scalar Floating-Point Final Result with Negation," on page 453.

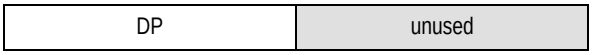
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

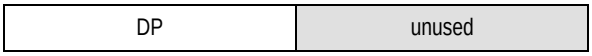
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsnmsub(alm)dp

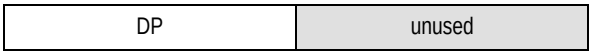
src1 = VSR[XA]



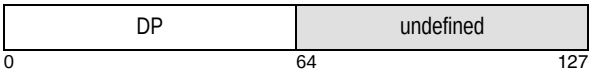
src2 = ***xsnmsubadp*** ? VSR[XT] : VSR[XB]



src3 = ***xsnmsubadp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsnmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsnmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsnmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 73.Actions for xsnmsub(a|m)dp

VSX Scalar Negative Multiply-Subtract Single-Precision XX3-form

xsnmsubasp XT,XA,XB

60	T	A	B	145	AX	BX	TX
0	6	11	16	21	29	30	31

xsnmsubmsp XT,XA,XB

60	T	A	B	153	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if "xsnmsubasp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×TX+T].doubleword[0]
   src3 ← VSR[32×BX+B].doubleword[0]
end
if "xsnmsubmsp" then do
   src1 ← VSR[32×AX+A].doubleword[0]
   src2 ← VSR[32×BX+B].doubleword[0]
   src3 ← VSR[32×TX+T].doubleword[0]
end

v      ← MultiplyAddDP(src1,src3,NegateDP(src2)))
result ← NegateSP(RoundToSP(RN,v))

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For ***xsnmsubasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmsubmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 74, "Actions for xsnmsub(a|m)sp," on page 463.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 74, "Actions for xsnmsub(a|m)sp," on page 463.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 71, "Scalar Floating-Point Final Result with Negation," on page 453.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

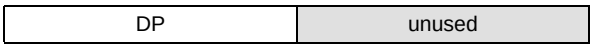
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

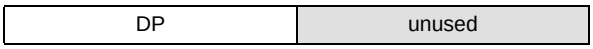
VSR Data Layout for xsnmsub(a|m)sp

src1 = VSR[XA]

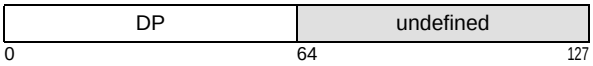
src2 = ***xsnmsubasp*** ? VSR[XT] : VSR[XB]



src3 = ***xsnmsubasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

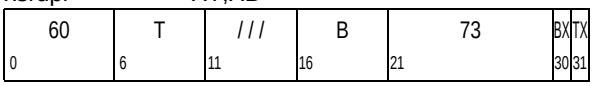
Explanation:

src1	The double-precision floating-point value in VSR[XA].doubleword[0].
src2	For ***xsnmsubasp***, the double-precision floating-point value in VSR[XT].doubleword[0]. For ***xsnmsubmsp***, the double-precision floating-point value in VSR[XB].doubleword[0].
src3	For ***xsnmsubasp***, the double-precision floating-point value in VSR[XB].doubleword[0]. For ***xsnmsubmsp***, the double-precision floating-point value in VSR[XT].doubleword[0].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 74.Actions for xsnmsub(a|m)sp

VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form

xsrdpi XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerNearAway(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassFP(result)
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src` is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

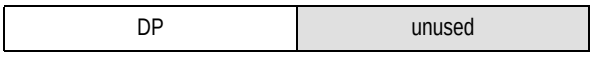
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

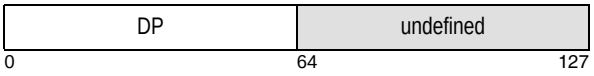
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpi

src = VSR[XB]



tgt = VSR[XT]



VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form

xsrdpic XT,XB

60	T	///	B	107	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
src         ← VSR[XB]{0:63}
if(RN=0b00) then result{0:63} ← RoundToDPIntegerNearEven(src)
if(RN=0b01) then result{0:63} ← RoundToDPIntegerTrunc(src)
if(RN=0b10) then result{0:63} ← RoundToDPIntegerCeil(src)
if(RN=0b11) then result{0:63} ← RoundToDPIntegerFloor(src)
if(vxsnan_flag) then SetFX(VXSNAN)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src` is rounded to an integer using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

FPRF FR FI FX XX VXSNAN

VSR Data Layout for xsrdpic

src = VSR[XB]

DP	unused

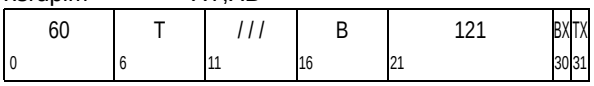
tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form

xsrdpim XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerFloor(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src` is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

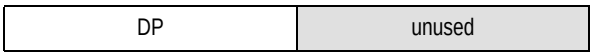
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

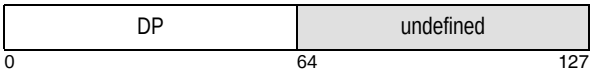
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpim

src = VSR[XB]

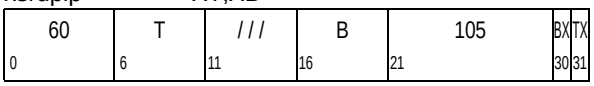


tgt = VSR[XT]



VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form

xsrdpip XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerCeil(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src` is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

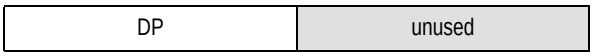
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

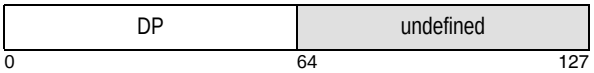
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpip

src = VSR[XB]

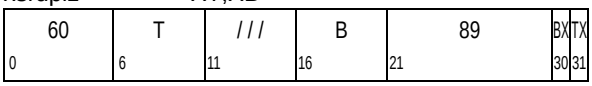


tgt = VSR[XT]



VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form

xsrdpiz XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerTrunc(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src` is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

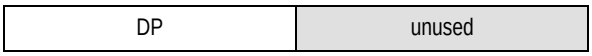
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

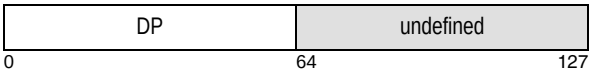
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpiz

src = VSR[XB]



tgt = VSR[XT]



VSX Scalar Reciprocal Estimate Double-Precision XX2-form

xsredp XT,XB

60	T	///	B	90	BX	TX
0	6	11	16	21	30	31

```
XT           ← TX || T
XB           ← BX || B
reset_xflags()
v{0:inf}     ← ReciprocalEstimateDP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(zx_flag)     then SetFX(ZX)
vex_flag     ← VE & vxsnan_flag
zex_flag     ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← 0bU
   FI      ← 0bU
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal of `src` is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of `src` would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of `src`. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\texttt{src}}}{\frac{1}{\texttt{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU FX OX UX XX=0bU VXSNAN

VSR Data Layout for xsredp

src = VSR[XB]

DP	unused

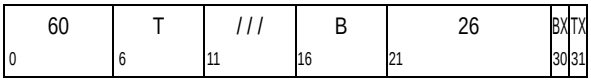
tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Reciprocal Estimate Single-Precision XX2-form

xsresp XT,XB




```
reset_xflags()

src    ← VSR[32×BX+B].doubleword[0]
v      ← ReciprocalEstimateDP(src)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(0bU)         then SetFX(XX)
if(zx_flag)     then SetFX(ZX)

vex_flag ← VE & vxsnan_flag
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← 0bU
   FI   ← 0bU
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

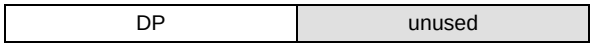
The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

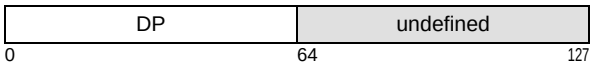
FPRF FR=0bU FI=0bU FX OX UX ZX XX=0bU
VXSNAN

VSR Data Layout for xsresp

src = VSR[XB]



tgt = VSR[XT]



VSX Scalar Round to Single-Precision XX2-form

xsrsp XT,XB

60	T	///	B	281	BX	TX
0	6	11	16	21	30	31

```
reset_xflags()

src    ← VSR[32×BX+B].doubleword[0]
result ← RoundToSP(RN,src)

if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format.

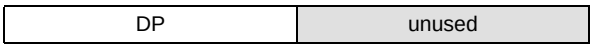
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified.

Special Registers Altered

FPRF FR FI FX OX UX XX VXSNAN

VSR Data Layout for xsrsp

src = VSR[XB]



tgt = VSR[XT]

0 64 127

VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form

xsrsqrtedp XT,XB

60	T	///	B	74	BX	TX
0	6	11	16	21	30	31

```
XT           ← TX || T
XB           ← BX || B
reset_xflags()
v{0:inf}     ← ReciprocalSquareRootEstimateDP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(zx_flag)     then SetFX(ZX)
vex_flag     ← VE & (vxsnan_flag | vxsqrt_flag)
zex_flag     ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← 0bU
   FI      ← 0bU
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\texttt{src}}}}{\frac{1}{\sqrt{\texttt{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
+Infinity	+Zero	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU FX OX UX
XX=0bU VXSNAN VXSQRT

VSR Data Layout for xsrsqrtedp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form

xsrsqrtesp XT,XB

60	T	/ / /	B	10	BX	TX
0	6	11	16	21	30	31

```
reset_xflags()

src    ← VSR[32×BX+B].doubleword[0]
v      ← ReciprocalSquareRootEstimateDP(src)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(0bU)         then SetFX(XX)
if(zx_flag)     then SetFX(ZX)

vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← 0bU
   FI   ← 0bU
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
+Infinity	+Zero	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU FX ZX XX=0bU VXSNAN VXSQRT

VSR Data Layout for xsrsqrtesp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Square Root Double-Precision XX2-form

xssqrtdp XT,XB

60	T	///	B	75	BX	TX
0	6	11	16	21	30	31

```
XT           ← TX || T
XB           ← BX || B
reset_xflags()
v{0:inf}     ← SquareRootFP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxsqrt_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The unbounded-precision square root of `src` is produced.

See Table 75.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX XX VXSNAN VXSQRT

VSR Data Layout for xssqrtdp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 75.Actions for xssqrtdp

VSX Scalar Square Root Single-Precision XX-form

xssqrtsp XT,XB

60	T	/ / /	B	11	BX	TX
0	6	11	16	21	30	31

```
reset_xflags()

src    ← VSR[32×BX+B].doubleword[0]
v      ← SquareRootDP(src)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertToDP(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 75.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXSQRT

VSR Data Layout for xssqrtsp

src = VSR[XB]

tgt = VSR[XT]

0 64 127

src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (0x7FF8_0000_0000_0000).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision and exponent range square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 76.Actions for xssqrtsp

VSX Scalar Subtract Double-Precision XX3-form

xssubdp XT,XA,XB

60	T	A	B	40	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
v{0:inf}     ← AddDP(src1,NegateDP(src2))
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

`src2` is negated and added[1] to `src1`, producing a sum having unbounded range and precision.

See Table 77.

The sum is normalized[2].

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

VSR Data Layout for xssubdp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

		src2							
		-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**-Infinity**	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	-Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	The floating-point value y is negated and then added to the floating-point value x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 77.Actions for xssubdp

VSX Scalar Subtract Single-Precision XX3-form

xssubsp XT,XA,XB

60	T	A	B	8	AX	BX	TX
0	6	11	16	21	30	30	31

```
reset_xflags()

src1   ← VSR[32×AX+A].doubleword[0]
src2   ← VSR[32×BX+B].doubleword[0]
v      ← AddDP(src1,NegateDP(src2))
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].doubleword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].doubleword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is negated and added[1] to src1, producing the sum, v, having unbounded range and precision.

See Table 78, “Actions for xssubsp,” on page 478.

v is normalized[2] and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 50, “Scalar Floating-Point Final Result,” on page 403.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

VSR Data Layout for xssubsp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

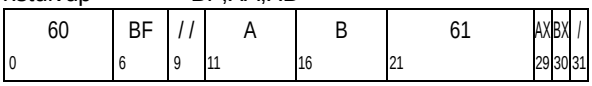
Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	The floating-point value y is negated and then added to the floating-point value x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 78.Actions for xssubsp

VSX Scalar Test for software Divide Double-Precision XX3-form

xstdivdp BF,XA,XB




```
XA          ← AX || A
XB          ← BX || B
src1        ← VSR[XA]{0:63}
src2        ← VSR[XB]{0:63}
e_a         ← VSR[XA]{1:11} - 1023
e_b         ← VSR[XB]{1:11} - 1023
fe_flag     ← IsNaN(src1) | IsInf(src1) |
              IsNaN(src2) | IsInf(src2) | IsZero(src2) |
              ( e_b <= -1022 ) |
              ( e_b >=  1021 ) |
              ( !IsZero(src1) & ( (e_a - e_b) >=  1023 ) ) |
              ( !IsZero(src1) & ( (e_a - e_b) <= -1021 ) ) |
              ( !IsZero(src1) & ( e_a <= -970 ) )
fg_flag     ← IsInf(src1) | IsInf(src2) |
              IsZero(src2) | IsDen(src2)
fl_flag     ← xsredp_error() <= 2^-14
CR[BF]      ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value AX concatenated with A.

Let XB be the value BX concatenated with B.

Let `src1` be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let `src2` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let `e_a` be the unbiased exponent of `src1`.

Let `e_b` be the unbiased exponent of `src2`.

`fe_flag` is set to 1 for any of the following conditions.
- `src1` is a NaN or an infinity.
- `src2` is a zero, a NaN, or an infinity.
- `e_b` is less than or equal to -1022.
- `e_b` is greater than or equal to 1021.
- `src1` is not a zero and the difference, `e_a - e_b`, is greater than or equal to 1023.
- `src1` is not a zero and the difference, `e_a - e_b`, is less than or equal to -1021.
- `src1` is not a zero and `e_a` is less than or equal to -970

Otherwise `fe_flag` is set to 0.

`fg_flag` is set to 1 for any of the following conditions.
- `src1` is an infinity.
- `src2` is a zero, an infinity, or a denormalized value.

Otherwise `fg_flag` is set to 0.

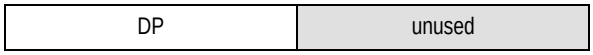
CR field BF is set to the value `0b1 || fg_flag || fe_flag || 0b0`.

Special Registers Altered

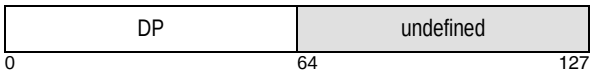
CR[BF]

VSR Data Layout for xstdivdp

src1 = VSR[XA]



src2 = VSR[XB]



VSX Scalar Test for software Square Root Double-Precision XX2-form

xstsqrtdp BF,XB

60	BF	/ /	/ / /	B	106	BX	/
0	6	9	11	16	21	30	31

```
XB       ← BX || B
src      ← VSR[XB]{0:63}
e_b      ← VSR[XB]{1:11} - 1023
fe_flag  ← IsNaN(src) | IsInf(src) | IsZero(src) |
             IsNeg(src) | ( e_b <= -970 )
fg_flag  ← IsInf(src) | IsZero(src) | IsDen(src)
fl_flag  ← xsrsqrtedp_error() <= 2^-14
CR[BF]   ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XB be the value BX concatenated with B.

Let `src` be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let e_b be the unbiased exponent of `src`.

`fe_flag` is set to 1 for any of the following conditions.
- `src` is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -970

Otherwise `fe_flag` is set to 0.

`fg_flag` is set to 1 for any of the following conditions.
- `src` is a zero, an infinity, or a denormalized value.

Otherwise `fg_flag` is set to 0.

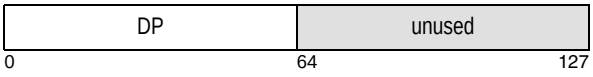
CR field BF is set to the value `0b1` || `fg_flag` || `fe_flag` || `0b0`.

Special Registers Altered

CR[BF]

VSR Data Layout for xstsqrtdp

src = VSR[XB]



VSX Vector Absolute Value Double-Precision XX2-form

xvabsdp XT,XB

60	T	/ / /	B	473	BX	TX
0	6	11	16	21	30	31

```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← 0b0 || VSR[XB]{i+1:i+63}
end
```

Let XT be the value TX concatenated with T.

Let XB be the value BX concatenated with B.

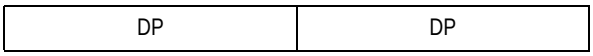
For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 set to 0, is placed into doubleword element i of VSR[XT].

Special Registers Altered

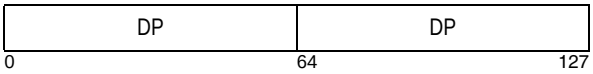
None

VSR Data Layout for xvabsdp

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Absolute Value Single-Precision XX2-form

xvabssp XT,XB

60	T	///	B	409	BX	TX
0	6	11	16	21	30	31

```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← 0b0 || VSR[XB]{i+1:i+31}
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 set to 0, is placed into word element i of VSR[XT].

Special Registers Altered

None

VSR Data Layout for xvabssp

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Add Double-Precision XX3-form

xvadddp XT,XA,XB

60	T	A	B	96	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   v{0:inf}       ← AddDP(src1,src2)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src2` is added[1] to `src1`, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 79.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvadddp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 79.Actions for xvadddp (element i)

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	0	0	0	0	–	–	–	–	T(r)
	–	–	–	0	–	–	–	–	–	–	–	1	–	–	–	–	T(r), fx(ZX)
	–	–	–	1	–	–	–	–	–	–	–	1	–	–	–	–	fx(ZX), error()
	0	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), fx(VXSQRT)
	0	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	T(r), fx(VXZDZ)
	0	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	T(r), fx(VXIDI)
	0	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	T(r), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	T(r), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	T(r), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), fx(VXSQRT)
	1	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	fx(VXZDZ), error()
	1	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	fx(VXIDI), error()
	1	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r)
	–	–	–	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(XX)
	–	–	–	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(XX)
	–	–	–	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(XX), error()
	–	–	–	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXSQRT	Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, $FPSCR_{VXSQRT}$.
VXIDI	Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, $FPSCR_{VXIDI}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
VXZDZ	Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, $FPSCR_{VXZDZ}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
ZX	Floating-Point Zero Divide Exception status flag, $FPSCR_{ZX}$.

Table 80.Vector Floating-Point Final Result

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	–	–	–	–	T(r), fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	–	–	–	–	T(r), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	no	–	fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	no	fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	yes	fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r)
	–	–	0	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	no	–	fx(UX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	no	fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	yes	fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXSQRT	Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, $FPSCR_{VXSQRT}$.
VXIDI	Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, $FPSCR_{VXIDI}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
VXZDZ	Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, $FPSCR_{VXZDZ}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
ZX	Floating-Point Zero Divide Exception status flag, $FPSCR_{ZX}$.

Table 80.Vector Floating-Point Final Result (Continued)

VSX Vector Add Single-Precision XX3-form

xvaddsp XT,XA,XB

60	T	A	B	64	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   v{0:inf}       ← AddSP(src1,src2)
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

`src2` is added[1] to `src1`, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 81.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvaddsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

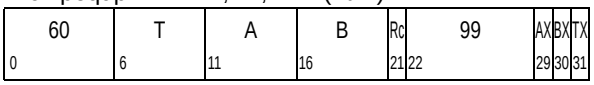
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 81.Actions for xvaddsp (element i)

VSX Vector Compare Equal To Double-Precision [& Record] XX3-form

xvcmpeqdp XT,XA,XB (Rc=0)
xvcmpeqdp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i←0 to 127 by 64
   reset_xflags()
   src1        ← VSR[XA]{i:i+63}
   src2        ← VSR[XB]{i:i+63}
   vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)

   if( CompareEQDP(src1,src2) ) then
      result{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+63} ← 0x0000_0000_0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src1` is compared to `src2`.

The contents of doubleword element i of VSR[XT] are set to all 1s if `src1` is equal to `src2`, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

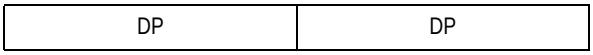
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

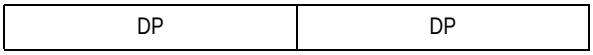
CR[6] . . . . . (if Rc=1)
FX VXSNAN

VSR Data Layout for xvcmpeqdp[.]

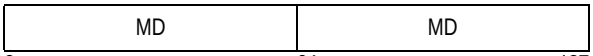
src1 = VSR[XA]

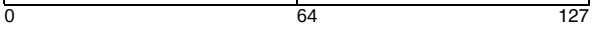


src2 = VSR[XB]



tgt = VSR[XT]





VSX Vector Compare Equal To Single-Precision [& Record] XX3-form

xvcmpeqsp XT,XA,XB (Rc=0)
xvcmpeqsp. XT,XA,XB (Rc=1)

60	T	A	B	Rc	67	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 32
   reset_xflags()
   src1        ← VSR[XA]{i:i+31}
   src2        ← VSR[XB]{i:i+31}
   vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)

   if( CompareEQSP(src1,src2) ) then
      result{i:i+31} ← 0xFFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+31} ← 0x0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

`src1` is compared to `src2`.

The contents of word element i of VSR[XT] are set to all 1s if `src1` is equal to `src2`, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

CR[6] . . . . . (if Rc=1)
FX VXSNAN

VSR Data Layout for xvcmpeqsp[.]

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

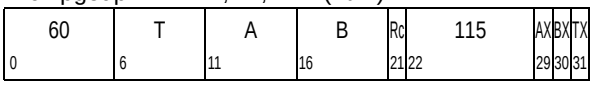
SP	SP	SP	SP

tgt = VSR[XT]

MW	MW	MW	MW
0	32	64	96	127

VSX Vector Compare Greater Than or Equal To Double-Precision [& Record] XX3-form

xvcmpgedp XT,XA,XB (Rc=0)
xvcmpgedp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 64
   reset_xflags()
   src1        ← VSR[XA]{i:i+63}
   src2        ← VSR[XB]{i:i+63}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGEDP(src1,src2) ) then
      result{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+63} ← 0x0000_0000_0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src1` is compared to `src2`.

The contents of doubleword element i of VSR[XT] are set to all 1s if `src1` is greater than or equal to the double-precision floating-point operand in doubleword element i of VSR[XB]`src2`, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

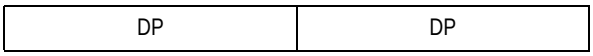
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

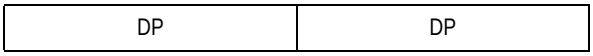
CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgedp[.]

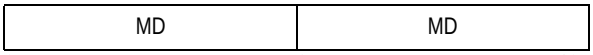
src1 = VSR[XA]



src2 = VSR[XB]



tgt = VSR[XT]



VSX Vector Compare Greater Than or Equal To Single-Precision [& record CR6] XX3-form

xvcmpgesp XT,XA,XB (Rc=0)
xvcmpgesp. XT,XA,XB (Rc=1)

60	T	A	B	Rc	83	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 32
   reset_xflags()
   src1        ← VSR[XA]{i:i+31}
   src2        ← VSR[XB]{i:i+31}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGESP(src1,src2) ) then
      result{i:i+31} ← 0xFFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+31} ← 0x0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

`src1` is compared to `src2`.

The contents of word element i of VSR[XT] are set to all 1s if `src1` is greater than or equal to `src2`, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgesp[.]

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

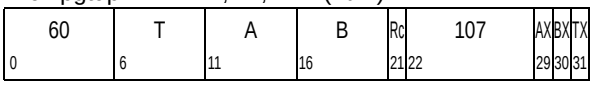
SP	SP	SP	SP

tgt = VSR[XT]

MW	MW	MW	MW
0	32	64	96	127

VSX Vector Compare Greater Than Double-Precision [& record CR6] XX3-form

xvcmpgtdp XT,XA,XB (Rc=0)
xvcmpgtdp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 64
   reset_xflags()
   src1        ← VSR[XA]{i:i+63}
   src2        ← VSR[XB]{i:i+63}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGTDP(src1,src2) ) then do
      result{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+63} ← 0x0000_0000_0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src1` is compared to `src2`.

The contents of doubleword element i of VSR[XT] are set to all 1s if `src1` is greater than `src2`, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

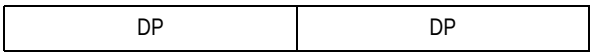
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

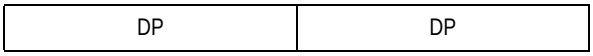
CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgtdp[.]

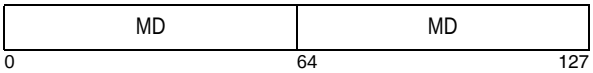
src1 = VSR[XA]

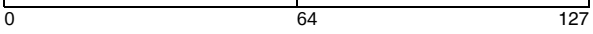


src2 = VSR[XB]



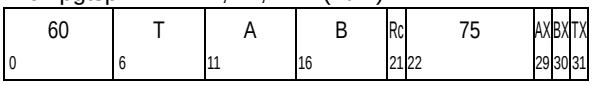
tgt = VSR[XT]





VSX Vector Compare Greater Than Single-Precision [& record CR6] XX3-form

xvcmpgtsp XT,XA,XB (Rc=0)
xvcmpgtsp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 32
   reset_xflags()
   src1        ← VSR[XA]{i:i+31}
   src2        ← VSR[XB]{i:i+31}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGTSP(src1,src2) ) then do
      result{i:i+31} ← 0xFFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+31} ← 0x0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

`src1` is compared to `src2`.

The contents of word element i of VSR[XT] are set to all 1s if `src1` is greater than `src2`, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

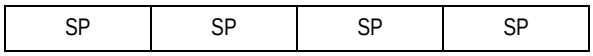
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

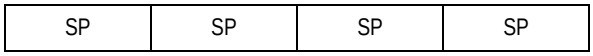
CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgtsp[.]

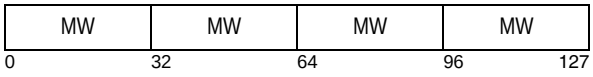
src1 = VSR[XA]



src2 = VSR[XB]

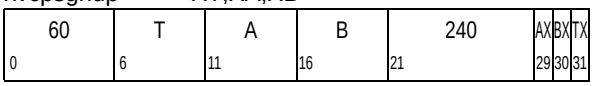


tgt = VSR[XT]



VSX Vector Copy Sign Double-Precision XX3-form

xvcpsgndp XT,XA,XB




```
XT ← TX || T
XA ← AX || A
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← VSR[XA]{i} || VSR[XB]{i+1:i+63}
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.
The contents of bit 0 of doubleword element i of VSR[XA] are concatenated with the contents of bits 1:63 of doubleword element i of VSR[XB] and placed into doubleword element i of VSR[XT].

Special Registers Altered

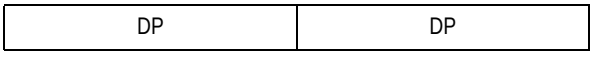
None

Extended Mnemonic		Equivalent To	
xvmovdp	XT,XB	xvcpsgndp	XT,XB,XB

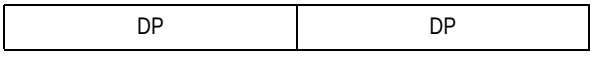
Table 82:

VSR Data Layout for xvcpsgndp

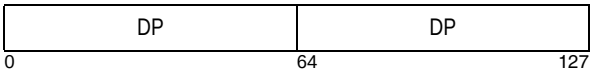
src1 = VSR[XA]



src2 = VSR[XB]

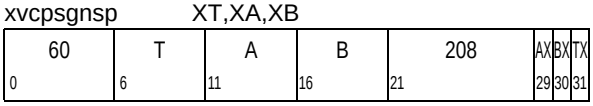


tgt = VSR[XT]



VSX Vector Copy Sign Single-Precision XX3-form

xvcpsgnsp XT,XA,XB




```
XT ← TX || T
XA ← AX || A
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← VSR[XA]{i} || VSR[XB]{i+1:i+31}
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.
The contents of bit 0 of word element i of VSR[XA] are concatenated with the contents of bits 1:31 of word element i of VSR[XB] and placed into word element i of VSR[XT].

Special Registers Altered

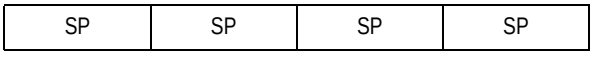
None

Extended Mnemonic		Equivalent To	
xvmovsp	XT,XB	xvcpsgnsp	XT,XB,XB

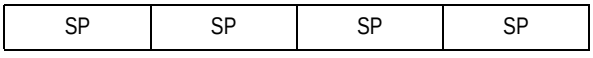
Table 83:

VSR Data Layout for xvcpsgnsp

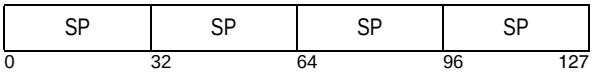
src1 = VSR[XA]



src2 = VSR[XB]



tgt = VSR[XT]



VSX Vector round Double-Precision to single-precision and Convert to Single-Precision format XX2-form

xvcvdpsp XT,XB

60	T	///	B	393	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src              ← VSR[XB]{i:i+63}
   result{i:i+31}   ← RoundToSP(RN,src)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag          ← ex_flag | (VE & vxsnan_flag)
   ex_flag          ← ex_flag | (OE & ox_flag)
   ex_flag          ← ex_flag | (UE & ux_flag)
   ex_flag          ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src` is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN

VSR Data Layout for xvcvdpsp

src = VSR[XB]

DP	DP

tgt = VSR[XT]

SP	undefined	SP	undefined
0	32	64	96	127

VSX Vector truncate Double-Precision to integer and Convert to Signed Integer Doubleword format with Saturate XX2-form

xvcvdpsxds XT,XB

60	T	///	B	472	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   rnd{0:63}      ← RoundToDPIntegerTrunc(VSR[XB]{i:i+63})
   result{i:i+63} ← ConvertDPtoSD(rnd)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

If `src` is a NaN, the result is the value `0x8000_0000_0000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{63}-1, the result is `0x7FFF_FFFF_FFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is `0x8000_0000_0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

The result is placed into doubleword element i of VSR[XT].

See Table 84.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpsxds

src = VSR[XB]

DP	DP

tgt = VSR[XT]

SD	SD

0 64 127

Programming Note

xvcvdpsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSD(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

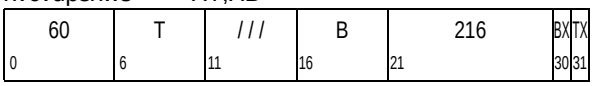
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
T(x)	The signed integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ {0,1}).

Table 84.Actions for xvcvdpsxds

VSX Vector truncate Double-Precision to integer and Convert to Signed Integer Word format with Saturate XX2-form

xvcvdpsxws XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   rnd{0:63}         ← RoundToDPIntegerTrunc(VSR[XB]{i:i+63})
   result{i:i+31}    ← ConvertDPtoSW(rnd)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

If `src` is a NaN, the result is the value `0x8000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{31}-1, the result is `0x7FFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, the result is `0x8000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

The result is placed into bits 0:31 of doubleword element i of VSR[XT].

The contents of bits 32:63 of doubleword element 1 of VSR[XT] are undefined.

See Table 85.

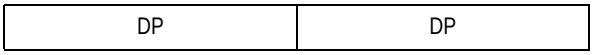
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

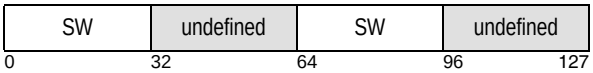
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpsxws

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvdpsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSW(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	T(Nmax), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

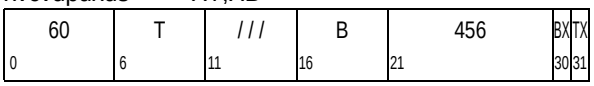
Explanation:

ConvertDPtoSW(x)	The double-precision floating-point integer value x converted to signed integer word format.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer word value, -2^{31} (0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x7FFF_FFFF).
RoundToDPintegerTrunc(x)	The double-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
T(x)	The signed integer word value x is placed in word element i of VSR[XT] (where i ∈ {0,2}).

Table 85.Actions for xvcvdpsxws

VSX Vector truncate Double-Precision to integer and Convert to Unsigned Integer Doubleword format with Saturate XX2-form

xvcvdpuxds XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   rnd{0:63}      ← RoundToDPIntegerTrunc(VSR[XB]{i:i+63})
   result{i:i+63} ← ConvertDPtoUD(rnd)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

If `src` is a NaN, the result is the value `0x0000_0000_0000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is `0xFFFF_FFFF_FFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is `0x0000_0000_0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

The result is placed into doubleword element i of VSR[XT].

See Table 86.

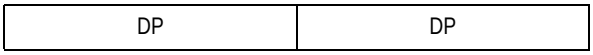
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

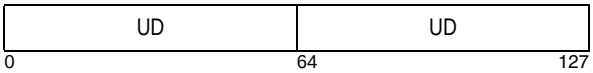
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpuxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvdpuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUD(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	T(Nmax), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

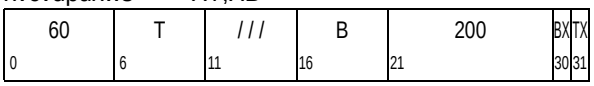
Explanation:

ConvertDPtoUD(x)	The double-precision floating-point integer value x converted to unsigned integer doubleword format.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF).
RoundToDPintegerTrunc(x)	The double-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
src	The double-precision floating-point value in doubleword element i VSR[XB] (where i ∈ {0,1}).
T(x)	The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ {0,1}).

Table 86.Actions for xvcvdpuxds

VSX Vector truncate Double-Precision to integer and Convert to Unsigned Integer Word format with Saturate XX2-form

xvcvdpuxws XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   rnd{0:63}         ← RoundToDIntegerTrunc(VSR[XB]{i:i+63})
   result{i:i+31}    ← ConvertDPtoUW(rnd)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

If `src` is a NaN, the result is the value `0x8000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{32}-1, the result is `0xFFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is `0x0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

The result is placed into bits 0:31 of doubleword element i of VSR[XT].

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

See Table 87.

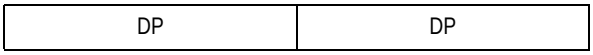
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

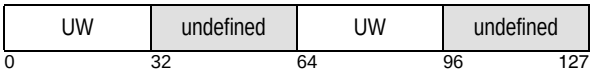
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpuxws

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvdpuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUW(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

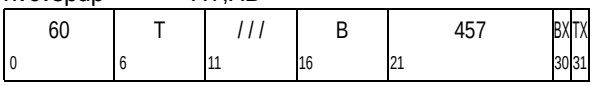
Explanation:

ConvertDPtoUW(x)	The double-precision floating-point integer value x converted to unsigned integer word format.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, 2^{32}-1 (0xFFFF_FFFF).
RoundToDPintegerTrunc(x)	The double-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
T(x)	The unsigned integer word value x is placed in word element i of VSR[XT] (where i ∈ {0,2}).

Table 87.Actions for xvcvdpuxws

VSX Vector Convert Single-Precision to Double-Precision format XX2-form

xvcvspdp XT,XB




```
XT              ← TX || T
XB              ← BX || B
ex_flag         ← 0b0
do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← ConvertSPtoDP(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the single-precision floating-point operand in bits 0:31 of doubleword element i of VSR[XB].

`src` is placed into doubleword element i of VSR[XT] in double-precison format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

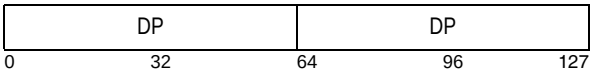
Special Registers Altered

FX VXSNAN

VSR Data Layout for xvcvspdp

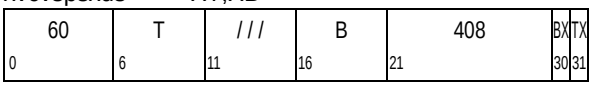
src = VSR[XB]

tgt = VSR[XT]



VSX Vector truncate Single-Precision to integer and Convert to Signed Integer Doubleword format with Saturate XX2-form

xvcvspsxds XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   rnd{0:31}      ← RoundToSPIntegerTrunc(VSR[XB]{i:i+31})
   result{i:i+63} ← ConvertSPtoSD(rnd)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the single-precision floating-point operand in word element i×2 of VSR[XB].

If `src` is a NaN, the result is the value `0x8000_0000_0000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{63}-1, the result is `0x7FFF_FFFF_FFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is `0x8000_0000_0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format.

The result is placed into doubleword element i of VSR[XT].

See Table 87.

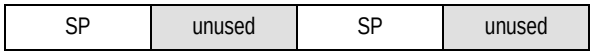
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

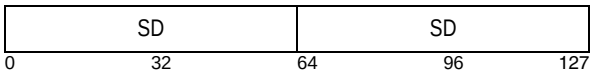
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspsxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvspsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoSD(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoSD(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

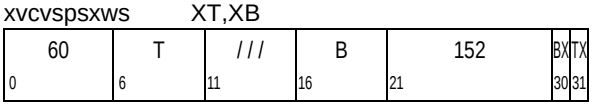
Explanation:

ConvertSPtoSD(x)	The single-precision floating-point integer value x converted to signed integer doubleword format.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
RoundToSPintegerTrunc(x)	The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
src	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,2}).
T(x)	The signed integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ {0,1}).

Table 88.Actions for xvcvspsxds

VSX Vector truncate Single-Precision to integer and Convert to Signed Integer Word format with Saturate XX2-form

xvcvspsxws XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   rnd{0:31}      ← RoundToSPIntegerTrunc(VSR[XB]{i:i+31})
   result{i:i+31} ← ConvertSPtoSW(rnd)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src` be the single-precision floating-point operand in word element i of VSR[XB].

If `src` is a NaN, the result is the value `0x8000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{31}-1, the result is `0x7FFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, the result is `0x8000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format.

The result is placed into word element i of VSR[XT].

See Table 87.

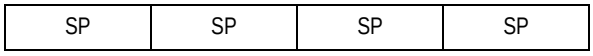
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

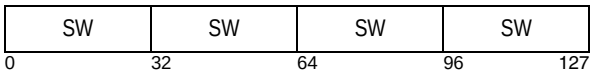
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspsxws

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvspsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoSW(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoSW(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

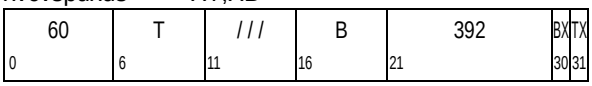
Explanation:

ConvertSPtoSW(x)	The single-precision floating-point integer value x converted to signed integer word format.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer word value, -2^{31} (0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x7FFF_FFFF).
RoundToSPintegerTrunc(x)	The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
src	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
T(x)	The signed integer word value x is placed in word element i of VSR[XT] (where i ∈ {0,1,2,3}).

Table 89.Actions for xvcvspsxws

VSX Vector truncate Single-Precision to integer and Convert to Unsigned Integer Doubleword format with Saturate XX2-form

xvcvspuxds XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   rnd{0:inf}     ← RoundToSPIntegerTrunc(src)
   result{i:i+63} ← ConvertSPtoUD(rnd)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the single-precision floating-point operand in word element i×2 of VSR[XB].

If `src` is a NaN, the result is the value `0x0000_0000_0000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is `0xFFFF_FFFF_FFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is `0x0000_0000_0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format.

The result is placed into doubleword element i of VSR[XT].

See Table 87.

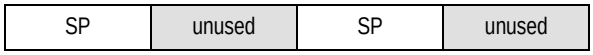
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

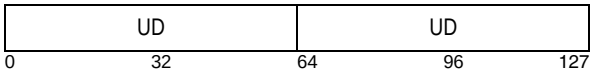
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspuxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvspuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoUD(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoUD(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

Explanation:

ConvertSPtoUD(x)	The single-precision floating-point integral value x converted to unsigned integer doubleword format.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF).
RoundToSPintegerTrunc(x)	The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
src	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,2}).
T(x)	The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ {0,1}).

Table 90.Actions for xvcvspuxds

VSX Vector truncate Single-Precision to integer and Convert to Unsigned Integer Word format with Saturate XX2-form

xvcvspuxws XT,XB

60	T	///	B	136	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   rnd{0:31}      ← RoundToSPIntegerTrunc(src)
   result{i:i+31} ← ConvertSPtoUW(rnd)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxcvi_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src` be the single-precision floating-point operand in word element i of VSR[XB].

If `src` is a NaN, the result is the value `0x0000_0000` and VXCVI is set to 1. If `src` is an SNaN, VXSNAN is also set to 1.

Otherwise, `src` is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{32}-1, the result is `0xFFFF_FFFF` and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is `0x0000_0000` and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format.

The result is placed into word element i of VSR[XT].

See Table 87.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspuxws

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

UW	UW	UW	UW
0	32	64	96	127

Programming Note

xvcvspuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by the RN field in the FPSCR.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoUW(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoUW(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

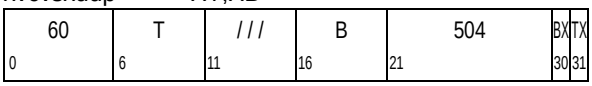
Explanation:

ConvertSPtoUW(x)	The single-precision floating-point integer value x converted to unsigned integer word format.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, 2^{32}-1 (0xFFFF_FFFF).
RoundToSPintegerTrunc(x)	The single-precision floating-point value x rounded to an integer using the rounding mode Round towards Zero.
src	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
T(x)	The unsigned integer word value x is placed in word element i of VSR[XT] (where i ∈ {0,1,2,3}).

Table 91.Actions for xvcvspuxws

VSX Vector Convert and round Signed Integer Doubleword to Double-Precision format XX2-form

xvcvsxddp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ConvertSDtoFP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

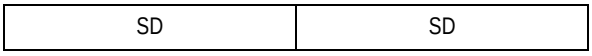
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

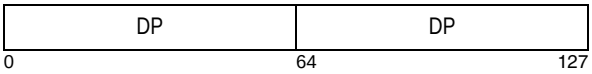
FX XX

VSR Data Layout for xvcvsxddp

src = VSR[XB]

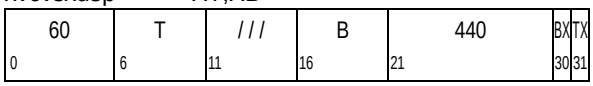


tgt = VSR[XT]



VSX Vector Convert and round Signed Integer Doubleword to Single-Precision format XX2-form

xvcvsxdsp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}          ← ConvertSDtoFP(VSR[XB]{i:i+63})
   result{i:i+31}    ← RoundToSP(RN,v)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(xx_flag) then SetFX(XX)
   ex_flag           ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

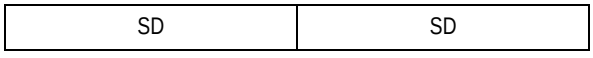
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

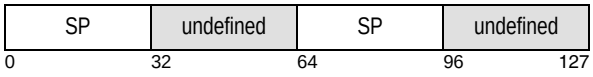
FX XX

VSR Data Layout for xvcvsxdsp

src = VSR[XB]

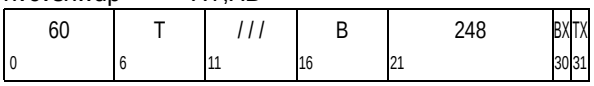


tgt = VSR[XT]



VSX Vector Convert Signed Integer Word to Double-Precision format XX2-form

xvcvsxwdp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ConvertSWtoFP(VSR[XB]{i:i+31})
   result{i:i+63} ← RoundToDP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the signed integer in bits 0:31 of doubleword element i of VSR[XB].

`src` is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

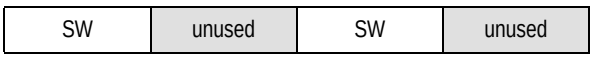
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

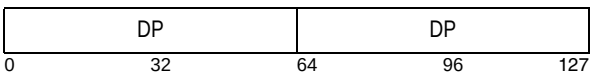
FX XX

VSR Data Layout for xvcvsxwdp

src = VSR[XB]

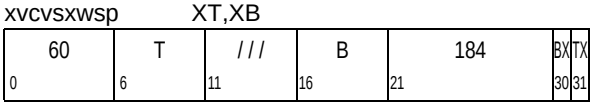


tgt = VSR[XT]



VSX Vector Convert and round Signed Integer Word to Single-Precision format XX2-form

xvcvsxwsp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← ConvertSWtoFP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToSP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src` be the signed integer in word element i of VSR[XB].

`src` is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into word element i of VSR[XT] in single-precision format.

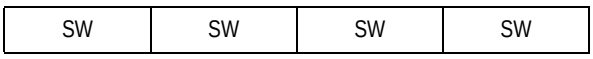
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

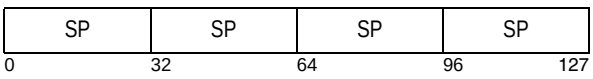
FX XX

VSR Data Layout for xvcvsxwsp

src = VSR[XB]

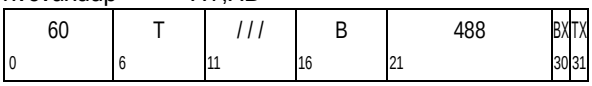


tgt = VSR[XT]



VSX Vector Convert and round Unsigned Integer Doubleword to Double-Precision format XX2-form

xvcvuxddp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ConvertUDtoFP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the unsigned integer in doubleword element i of VSR[XB].

`src` is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

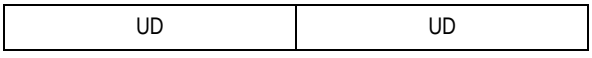
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

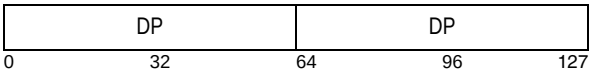
FX XX

VSR Data Layout for xvcvuxddp

src = VSR[XB]

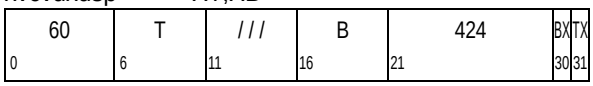


tgt = VSR[XT]



VSX Vector Convert and round Unsigned Integer Doubleword to Single-Precision format XX2-form

xvcvuxdsp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ConvertUDtoFP(VSR[XB]{i:i+63})
   result{i:i+31}    ← RoundToSP(RN,v)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the unsigned integer in doubleword element i of VSR[XB].

`src` is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

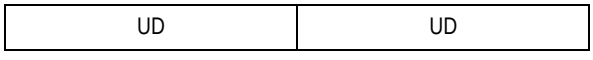
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

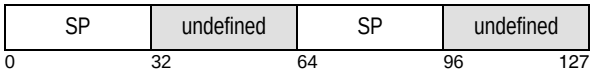
FX XX

VSR Data Layout for xvcvuxdsp

src = VSR[XB]

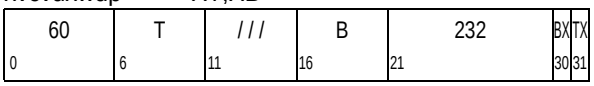


tgt = VSR[XT]



VSX Vector Convert and round Unsigned Integer Word to Double-Precision format XX2-form

xvcvuxwdp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ConvertUWtoFP(VSR[XB]{i:i+31})
   result{i:i+63} ← RoundToDP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the unsigned integer in bits 0:31 of doubleword element i of VSR[XB].

`src` is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

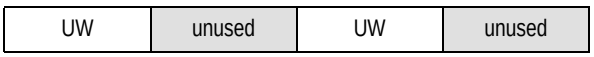
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

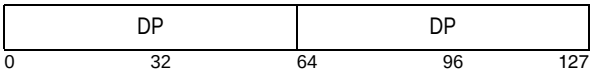
FX XX

VSR Data Layout for xvcvuxwdp

src = VSR[XB]

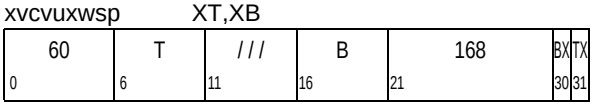


tgt = VSR[XT]



VSX Vector Convert and round Unsigned Integer Word to Single-Precision format XX2-form

xvcvuxwsp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← ConvertUWtoFP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToSP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src` be the unsigned integer in word element i of VSR[XB].

`src` is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into word element i of VSR[XT] in single-precision format.

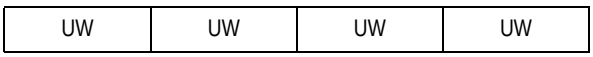
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

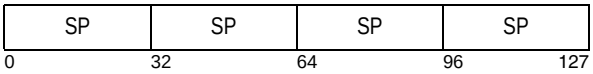
FX XX

VSR Data Layout for xvcvuxwsp

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Divide Double-Precision XX3-form

xvdivdp XT,XA,XB

60	T	A	B	120	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   v{0:inf}       ← DivideDP(src1,src2)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxidi_flag)  then SetFX(VXIDI)
   if(vxisi_flag)  then SetFX(VXZDZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxidi_flag)
   ex_flag        ← ex_flag | (VE & vxzdz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src1` is divided[1] by `src2`, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 92.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xvdivdp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← –Infinity zx_flag ← 1	v ← D(src1,src2)	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← –Zero	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Zero	v ← –Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Zero	v ← D(src1,src2)	v ← –Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 92.Actions for xvdivdp (element i)

VSX Vector Divide Single-Precision XX3-form

xvdivsp XT,XA,XB

60	T	A	B	88	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   v{0:inf}       ← DivideSP(src1,src2)
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxidi_flag)  then SetFX(VXIDI)
   if(vxisi_flag)  then SetFX(VXZDZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxidi_flag)
   ex_flag        ← ex_flag | (VE & vxzdz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

`src1` is divided[1] by `src2`, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 93.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xvdivsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← –Infinity zx_flag ← 1	v ← D(src1,src2)	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← –Zero	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Zero	v ← –Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Zero	v ← D(src1,src2)	v ← –Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 93.Actions for xvdivsp (element i)

VSX Vector Multiply-Add Double-Precision XX3-form

xvmaddadp XT,XA,XB

60	T	A	B	97	AX	BX	TX
0	6	11	16	21	29	30	31

xvmaddmdp XT,XA,XB

60	T	A	B	105	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvmaddadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvmaddadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,src2)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

For ***xvmaddadp***, do the following.

- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmaddmdp***, do the following.

- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 94.

`src2` is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 94.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, "Vector Floating-Point Final Result," on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

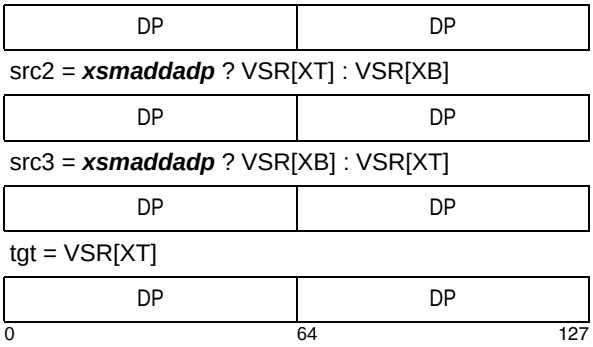
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmadd(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 94.Actions for xvmadd(a|m)dp

VSX Vector Multiply-Add Single-Precision XX3-form

xvmaddasp XT,XA,XB

60	T	A	B	65	AX	BX	TX
0	6	11	16	21	29	30	31

xvmaddmsp XT,XA,XB

60	T	A	B	73	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1 ← VSR[XA]{i:i+31}
   src2 ← "xvmaddasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvmaddasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}       ← MultiplyAddSP(src1,src3,src2)
   result{i:i+63} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

For ***xvmaddasp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XT].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvmaddmsp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 95.

`src2` is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 95.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, "Vector Floating-Point Final Result," on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

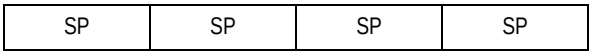
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

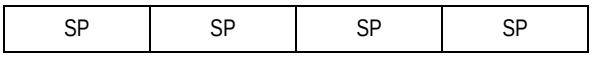
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmadd(a|m)sp

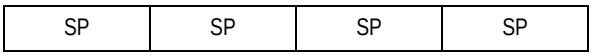
src1 = VSR[XA]



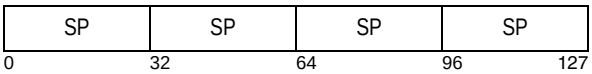
src2 = ***xsmaddasp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmaddasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	For ***xvmaddasp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}). For ***xvmaddmsp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
src3	For ***xvmaddasp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}). For ***xvmaddmsp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 95.Actions for xvmadd(a|m)sp

VSX Vector Maximum Double-Precision XX3-form

xvmaxdp XT,XA,XB

60	T	A	B	224	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   result{i:i+63} ← MaximumDP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

If `src1` is greater than `src2`, `src1` is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, `src2` is placed into doubleword element i of VSR[XT] in double-precision format.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

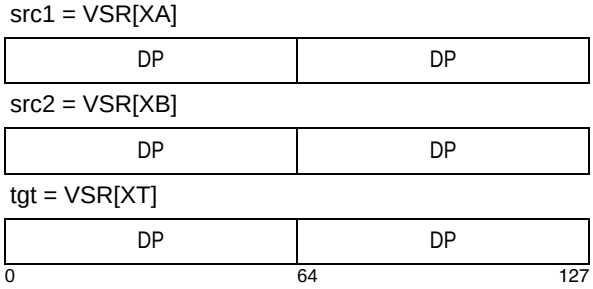
See Table 96.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvmaxdp



		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element i (i∈{0,1}) of VSR[XT] in double-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 96.Actions for xvmaxdp

VSX Vector Maximum Single-Precision XX3-form

xvmaxsp XT,XA,XB

60	T	A	B	192	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   result{i:i+63} ← MaximumSP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

If `src1` is greater than `src2`, `src1` is placed into word element i of VSR[XT] in single-precision format. Otherwise, `src2` is placed into word element i of VSR[XT] in single-precision format.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 97.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvmaxsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in word element i (i∈{0,1,2,3}) of VSR[XT] in single-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 97.Actions for xvmaxsp

VSX Vector Minimum Double-Precision XX3-form

xvmindp XT,XA,XB

60	T	A	B	232	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   result{i:i+63} ← MinimumDP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

If `src1` is less than `src2`, `src1` is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, `src2` is placed into doubleword element i of VSR[XT] in double-precision format.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

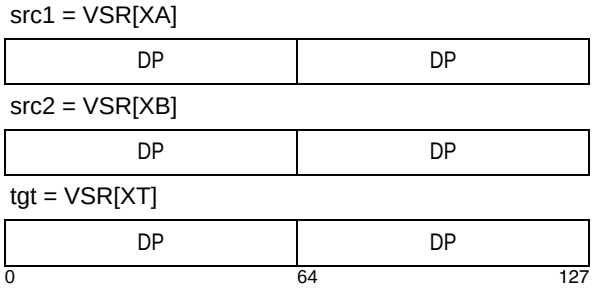
See Table 98.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvmindp



		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element i (i∈{0,1}) of VSR[XT] in double-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 98.Actions for xvmindp

VSX Vector Minimum Single-Precision XX3-form

xvminsp XT,XA,XB

60	T	A	B	200	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   result{i:i+31} ← MinimumSP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

If `src1` is less than `src2`, `src1` is placed into word element i of VSR[XT] in single-precision format. Otherwise, `src2` is placed into word element i of VSR[XT] in single-precision format.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 99.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvminsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in word element i (i∈{0,1,2,3}) of VSR[XT] in single-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 99.Actions for xvminsp

VSX Vector Multiply-Subtract Double-Precision XX3-form

xvmsubadp XT,XA,XB

60	T	A	B	113	AX	BX	TX
0	6	11	16	21	29	30	31

xvmsubmdp XT,XA,XB

60	T	A	B	121	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvmsubadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvmsubadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,NegateDP(src2))
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

For ***xvmsubadp***, do the following.
- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmsubmdp***, do the following.
- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 100.

`src2` is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 100.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, "Vector Floating-Point Final Result," on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

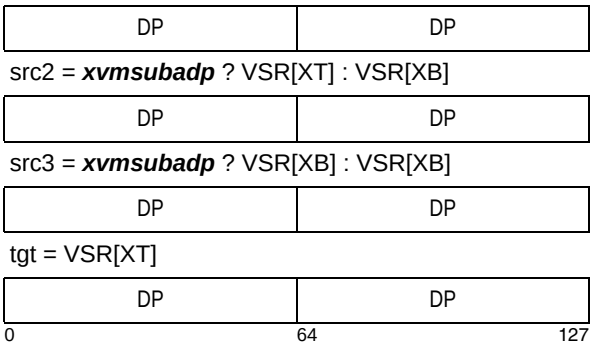
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmsub(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 100.Actions for xvmsub(a|m)dp

VSX Vector Multiply-Subtract Single-Precision XX3-form

xvmsubasp XT,XA,XB

60	T	A	B	81	AX	BX	TX
0	6	11	16	21	29	30	31

xvmsubmsp XT,XA,XB

60	T	A	B	89	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2 ← "xvmsubasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvmsubasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}       ← MultiplyAddSP(src1,src3,NegateSP(src2))
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

For ***xvmsubasp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XT].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvmsubmsp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 101.

`src2` is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 101.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, "Vector Floating-Point Final Result," on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

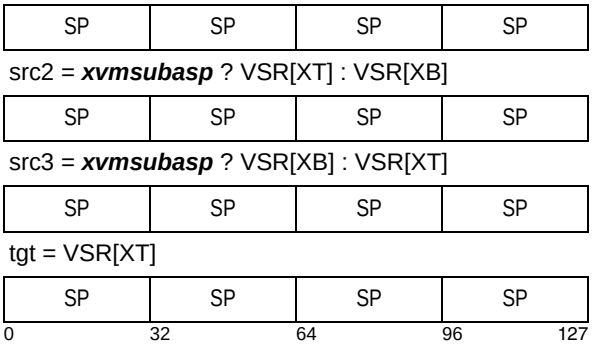
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmsub(alm)sp

src1 = VSR[XA]



Part 1: Multiply	src3							
src1	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	src2							
p	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	For ***xvmsubasp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}). For ***xvmsubmsp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
src3	For ***xvmsubasp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}). For ***xvmsubmsp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 101.Actions for xvmsub(a|m)sp

VSX Vector Multiply Double-Precision XX3-form

xvmuldp XT,XA,XB

60	T	A	B	112	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src3           ← VSR[XB]{i:i+63}
   v{0:inf}       ← MultiplyDP(src1,src3)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src1` is multiplied[1] by `src2`, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 102.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXIMZ

VSR Data Layout for xvmuldp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← −Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← −Zero	v ← −Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← −Zero	v ← −Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← M(src1,src2)	v ← −Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← −Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

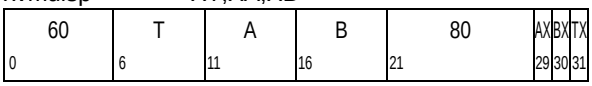
Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 102.Actions for xvmuldp

VSX Vector Multiply Single-Precision XX3-form

xvmulsp XT,XA,XB




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src3           ← VSR[XB]{i:i+31}
   v{0:inf}       ← MultiplySP(src1,src3)
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

`src1` is multiplied[1] by `src2`, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 103.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

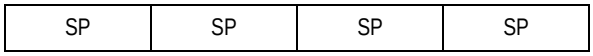
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

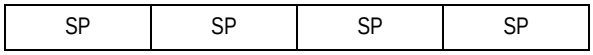
FX OX UX XX VXSNAN VXIMZ

VSR Data Layout for xvmulsp

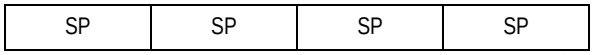
src1 = VSR[XA]

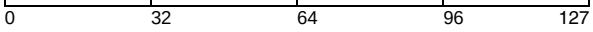


src2 = VSR[XB]



tgt = VSR[XT]





1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← −Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← −Zero	v ← −Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← −Zero	v ← −Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← −Infinity	v ← M(src1,src2)	v ← −Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← −Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

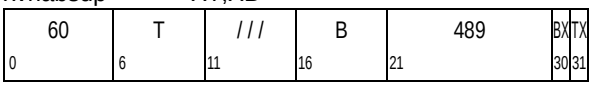
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 103.Actions for xvmulsp

VSX Vector Negative Absolute Value Double-Precision XX2-form

xvnabsdp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← 0b1 || VSR[XB]{i+1:i+63}
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

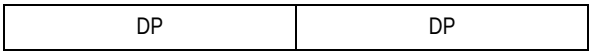
For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 set to 1, is placed into doubleword element i of VSR[XT].

Special Registers Altered

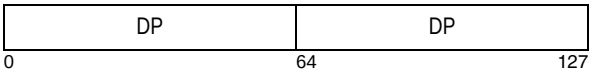
None

VSR Data Layout for xvnabsdp

src = VSR[XB]

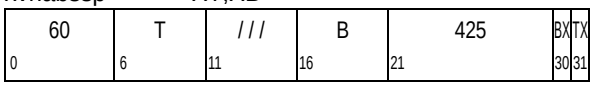


tgt = VSR[XT]



VSX Vector Negative Absolute Value Single-Precision XX2-form

xvnabssp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← 0b1 || VSR[XB]{i+1:i+31}
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

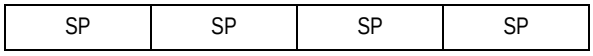
For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 set to 1, is placed into word element i of VSR[XT].

Special Registers Altered

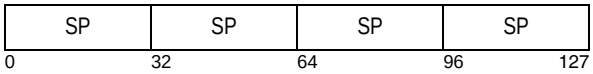
None

VSR Data Layout for xvnabssp

src = VSR[XB]

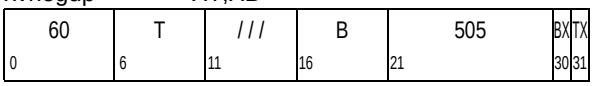


tgt = VSR[XT]



VSX Vector Negate Double-Precision XX2-form

xvnegdp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← ~VSR[XB]{i} || VSR[XB]{i+1:i+63}
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

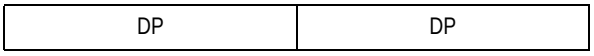
For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 complemented, is placed into doubleword element i of VSR[XT].

Special Registers Altered

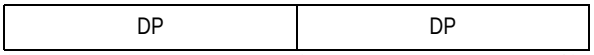
None

VSR Data Layout for xvnegdp

src = VSR[XB]

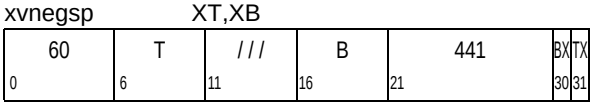


tgt = VSR[XT]



VSX Vector Negate Single-Precision XX2-form

xvnegsp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← ~VSR[XB]{i} || VSR[XB]{i+1:i+31}
end
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

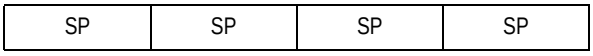
For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 complemented, is placed into word element i of VSR[XT].

Special Registers Altered

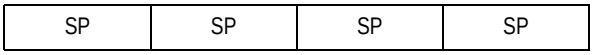
None

VSR Data Layout for xvnegsp

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Negative Multiply-Add Double-Precision XX3-form

xvnmaddadp XT,XA,XB

60	T	A	B	225	AX	BX	TX
0	6	11	16	21	29	30	31

xvnmaddmdp XT,XA,XB

60	T	A	B	233	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvnmaddadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvnmaddadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,src2)
   result{i:i+63} ← NegateDP(RoundToDP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

For ***xvnmaddadp***, do the following.

- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvnmaddmdp***, do the following.

- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 104.

`src2` is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 104.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 105, "Vector Floating-Point Final Result with Negation," on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

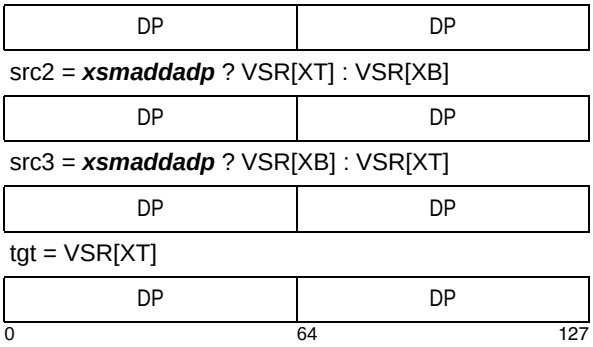
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmadd(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvnmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvnmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvnmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvnmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 104.Actions for xvnmadd(a|m)dp

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	–	–	–	–	T(N(r))
	0	–	–	–	–	–	–	1	–	–	–	–	T(r), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	T(r), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	T(r), fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	T(r), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	1	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	no	–	–	–	T(N(r))
	–	–	–	–	0	–	–	–	yes	no	–	–	T(N(r)), fx(XX)
	–	–	–	–	0	–	–	–	yes	yes	–	–	T(N(r)), fx(XX)
	–	–	–	–	1	–	–	–	yes	no	–	–	T(N(r)), fx(XX), error()
	–	–	–	–	1	–	–	–	yes	yes	–	–	T(N(r)), fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	T(N(r)), fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	T(N(r)), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	no	–	fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	yes	no	fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	yes	yes	fx(OX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Table 105.Vector Floating-Point Final Result with Negation

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Tiny	–	–	0	–	–	–	–	–	no	–	–	–	T(N(r))
	–	–	0	–	0	–	–	–	yes	no	–	–	T(N(r)), fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	yes	yes	–	–	T(N(r)), fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	yes	no	–	–	T(N(r)), fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	yes	yes	–	–	T(N(r)), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	no	–	fx(UX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	no	fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	yes	fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 49, “Floating-Point Intermediate Result Handling,” on page 402, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Table 105.Vector Floating-Point Final Result with Negation (Continued)

VSX Vector Negative Multiply-Add Single-Precision XX3-form

xvnmaddasp XT,XA,XB

60	T	A	B	193	AX	BX	TX
0	6	11	16	21	29	30	31

xvnmaddmsp XT,XA,XB

60	T	A	B	201	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2 ← "xvnmaddasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvnmaddasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}       ← MultiplyAddSP(src1,src3,src2)
   result{i:i+31} ← NegateSP(RoundToSP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

For ***xvnmaddasp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XT].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvnmaddmsp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 106.

`src2` is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 106.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 105, "Vector Floating-Point Final Result with Negation," on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

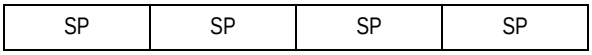
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

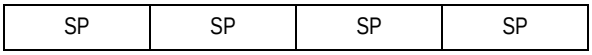
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmadd(alm)sp

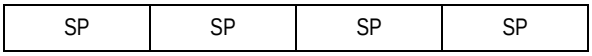
src1 = VSR[XA]



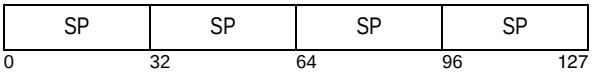
src2 = ***xsmaddadp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmaddadp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	For ***xvnmaddasp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}). For ***xvnmaddmsp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
src3	For ***xvnmaddasp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}). For ***xvnmaddmsp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 106.Actions for xvnmadd(a|m)sp

VSX Vector Negative Multiply-Subtract Double-Precision XX3-form

xvnmsubadp XT,XA,XB

60	T	A	B	241	AX	BX	TX
0	6	11	16	21	29	30	31

xvnmsubmdp XT,XA,XB

60	T	A	B	249	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvmsubadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvmsubadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,NegateDP(src2))
   result{i:i+63} ← NegateDP(RoundToDP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

For ***xvmsubadp***, do the following.

- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmsubmdp***, do the following.

- Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let `src3` be the double-precision floating-point operand in doubleword element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 107.

`src2` is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 107.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 105, "Vector Floating-Point Final Result with Negation," on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

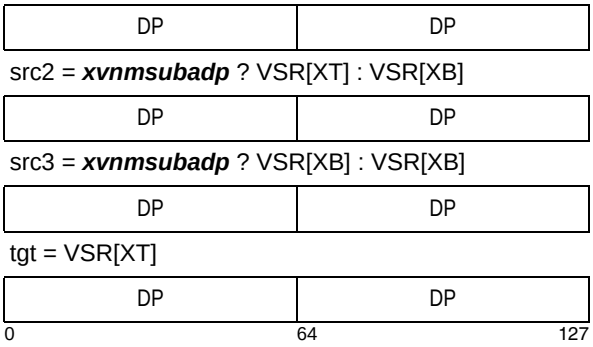
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmsub(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvnmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvnmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvnmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvnmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 107.Actions for xvnmsub(a|m)dp

VSX Vector Negative Multiply-Subtract Single-Precision XX3-form

xvnmsubasp XT,XA,XB

60	T	A	B	209	AX	BX	TX
0	6	11	16	21	29	30	31

xvnmsubmsp XT,XA,XB

60	T	A	B	217	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1 ← VSR[XA]{i:i+31}
   src2 ← "xvnmsubasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvnmsubasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}       ← MultiplyAddSP(src1,src3,NegateSP(src2))
   result{i:i+31} ← NegateSP(RoundToSP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

For ***xvnmsubasp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XT].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvnmsubmsp***, do the following.
- Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].
- Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].
- Let `src3` be the single-precision floating-point operand in word element i of VSR[XT].

`src1` is multiplied[1] by `src3`, producing a product having unbounded range and precision.

See part 1 of Table 108.

`src2` is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 108.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 105, "Vector Floating-Point Final Result with Negation," on page 550.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

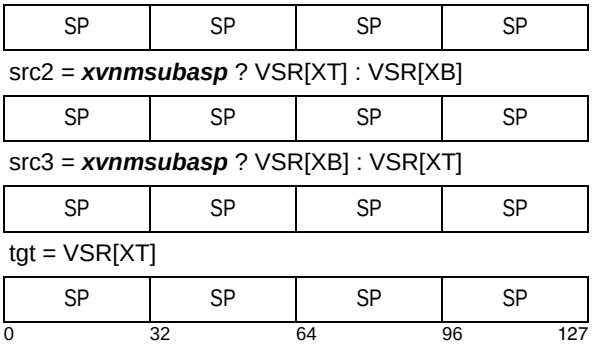
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmsub(alm)sp

src1 = VSR[XA]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
src3	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 108.Actions for xvnmsub(a|m)sp

VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form

xvrdpi XT,XB

60	T	///	B	201	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerNearAway(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvrdpi

src = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form

xvrdpic XT,XB

60	T	///	B	235	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src{0:63} ← VSR[XB]{i:i+63}
   if(RN=0b00) then
      result{i:i+63} ← RoundToDPIntegerNearEven(src)
   if(RN=0b01) then
      result{i:i+63} ← RoundToDPIntegerTrunc(src)
   if(RN=0b10) then
      result{i:i+63} ← RoundToDPIntegerCeil(src)
   if(RN=0b11) then
      result{i:i+63} ← RoundToDPIntegerFloor(src)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN

VSR Data Layout for xvrdpic

src = VSR[XB]

DP	DP

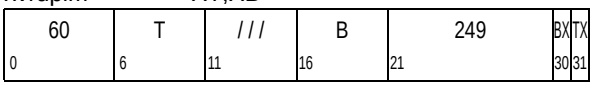
tgt = VSR[XT]

DP	DP

0 64 127

VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form

xvrdpim XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerFloor(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

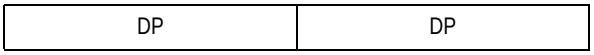
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

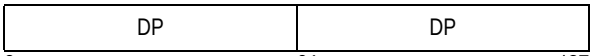
FX VXSNAN

VSR Data Layout for xvrdpim

src = VSR[XB]

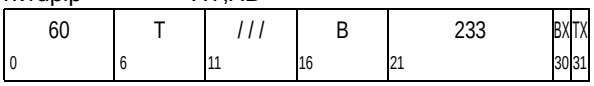


tgt = VSR[XT]



VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form

xvrdpip XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerCeil(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

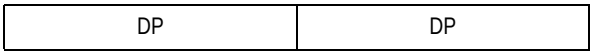
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

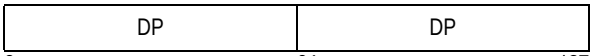
FX VXSNAN

VSR Data Layout for xvrdpip

src = VSR[XB]

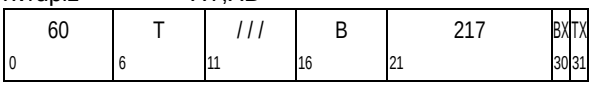


tgt = VSR[XT]



VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form

xvrdpiz XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerTrunc(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src` is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

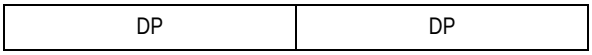
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

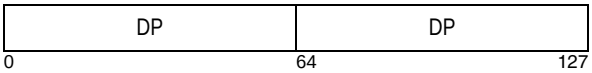
FX VXSNAN

VSR Data Layout for xvrdpiz

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Reciprocal Estimate Double-Precision XX2-form

xvredp XT,XB

60	T	///	B	218	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ReciprocalEstimateDP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

A double-precision floating-point estimate of the reciprocal of `src` is placed into doubleword element i of VSR[XT] in double-precision format.

Unless the reciprocal of `src` would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of `src`. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\texttt{src}}}{\frac{1}{\texttt{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX OX UX ZX VXSNAN

VSR Data Layout for xvredp

src = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

VSX Vector Reciprocal Estimate Single-Precision XX2-form

xvresp XT,XB

60	T	///	B	154	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← ReciprocalEstimateSP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src` be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal of `src` is placed into word element i of VSR[XT] in single-precision format.

Unless the reciprocal of `src` would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of `src`. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\texttt{src}}}{\frac{1}{\texttt{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX OX UX ZX VXSNAN

VSR Data Layout for xvresp

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form

xvrspi XT,XB

60	T	///	B	137	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} ← RoundToSPIntegerNearAway(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvrspi

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form

xvrspic XT,XB

60	T	///	B	171	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src{0:31} ← VSR[XB]{i:i+31}
   if(RN=0b00) then
      result{i:i+31} ← RoundToSPIntegerNearEven(src)
   if(RN=0b01) then
      result{i:i+31} ← RoundToSPIntegerTrunc(src)
   if(RN=0b10) then
      result{i:i+31} ← RoundToSPIntegerCeil(src)
   if(RN=0b11) then
      result{i:i+31} ← RoundToSPIntegerFloor(src)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer value using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN

VSR Data Layout for xvrspic

src = VSR[XB]

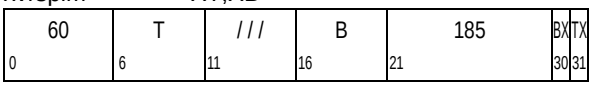
SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form

xvrspim XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} = RoundToSPIntegerFloor(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into word element i of VSR[XT] in single-precision format.

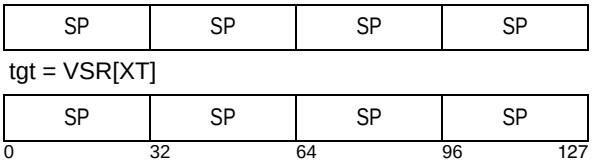
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

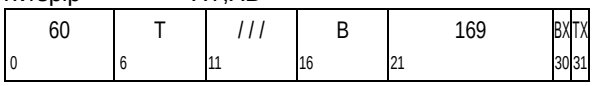
VSR Data Layout for xvrspim

src = VSR[XB]



VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form

xvrspip XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} = RoundToSPIntegerCeil(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into word element i of VSR[XT] in single-precision format.

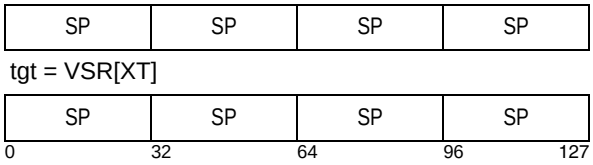
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvrspip

src = VSR[XB]



VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form

xvrspiz XT,XB

60	T	///	B	153	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} = RoundToSPIntegerTrunc(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into word element i of VSR[XT] in single-precision format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvrspiz

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form

xvrsqrtedp XT,XB

60	T	///	B	202	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i←0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← RecipSquareRootEstimateDP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

A double-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element i of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
+Infinity	+Zero	None
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

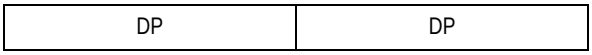
The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

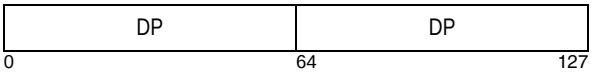
FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtedp

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form

xvrsqrtesp XT,XB

60	T	///	B	138	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← RecipSquareRootEstimateSP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src` be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of `src` is placed into word element i of VSR[XT] in single-precision format.

Unless the reciprocal of the square root of `src` would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of `src`. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\texttt{src}}}}{\frac{1}{\sqrt{\texttt{src}}}} \right| \leq \frac{1}{16384}$$

Architecture Note

Increasing the required accuracy of ***xvrsqrtesp*** to 13 bits is currently under consideration.

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
+Infinity	+Zero	None
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtesp

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Square Root Double-Precision XX2-form

xvsqrtdp XT,XB

60	T	///	B	203	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i←0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← SquareRootDP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (XE & xx_flag
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src` be the double-precision floating-point operand in doubleword element i of VSR[XB].

The unbounded-precision square root of `src` is produced.

See Table 109.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, "Floating-Point Intermediate Result Handling," on page 402.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, "Vector Floating-Point Final Result," on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXSQRT

VSR Data Layout for xvsqrtdp

src = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 109.Actions for xvsqrtdp

VSX Vector Square Root Single-Precision XX2-form

xvsqrtsp XT,XB

60	T	///	B	139	BX	TX
0	6	11	16	21	30	31

```
XT        ← TX || T
XB        ← BX || B
ex_flag   ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← SquareRootSP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (XE & xx_flag
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src` be the single-precision floating-point operand in word element i of VSR[XB].

The unbounded-precision square root of `src` is produced.

See Table 110.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXSQRT

VSR Data Layout for xvsqrtsp

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 110.Actions for xvsqrtsp

VSX Vector Subtract Double-Precision XX3-form

xvsubdp XT,XA,XB

60	T	A	B	104	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   v{0:inf}       ← AddDP(src1,NegateDP(src2))
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 1, do the following.

Let `src1` be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let `src2` be the double-precision floating-point operand in doubleword element i of VSR[XB].

`src2` is negated and added[1] to `src1`, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 111.

The intermediate result is rounded to double-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvsubdp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Infinity	v ← −src2	v ← −Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← +Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 111.Actions for xvsubdp

VSX Vector Subtract Single-Precision XX3-form

xvsubsp XT,XA,XB

60	T	A	B	72	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   v{0:inf}       ← AddSP(src1,NegateSP(src2))
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

For each vector element i from 0 to 3, do the following.

Let `src1` be the single-precision floating-point operand in word element i of VSR[XA].

Let `src2` be the single-precision floating-point operand in word element i of VSR[XB].

`src2` is negated and added[1] to `src1`, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 112.

The intermediate result is rounded to single-precision using the rounding mode specified by the Floating-Point Rounding Control field RN of the FPSCR.

See Table 49, “Floating-Point Intermediate Result Handling,” on page 402.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 80, “Vector Floating-Point Final Result,” on page 484.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvsubsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxisi_flag ← 1	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Infinity	v ← −src2	v ← −Zero	v ← Rezd	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← −src2	v ← Rezd	v ← +Zero	v ← −src2	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← −Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

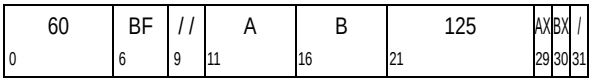
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 112.Actions for xvsubsp

VSX Vector Test for software Divide Double-Precision XX3-form

xvtdivdp BF,XA,XB




```
XA      ← AX || A
XB      ← BX || B
eq_flag ← 0b0
gt_flag ← 0b0

do i=0 to 127 by 64
   src1       ← VSR[XA]{i:i+63}
   src2       ← VSR[XB]{i:i+63}
   e_a        ← src1{1:11} - 1023
   e_b        ← src2{1:11} - 1023
   fe_flag    ← fe_flag | IsNaN(src1) | IsInf(src1) |
                IsNaN(src2) | IsInf(src2) | IsZero(src2) |
                ( e_b <= -1022 ) |
                ( e_b >=  1021 ) |
                ( !IsZero(src1) & ( (e_a - e_b) >=  1023 ) ) |
                ( !IsZero(src1) & ( (e_a - e_b) <= -1021 ) ) |
                ( !IsZero(src1) & ( e_a <= -970 ) )
   fg_flag    ← fg_flag | IsInf(src1) | IsInf(src2) |
                IsZero(src2) | IsDen(src2)
end

fl_flag ← xvredp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_a be the unbiased exponent of src1.

Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.

- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to -1022.
- e_b is greater than or equal to 1021.
- src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
- src1 is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
- src1 is not a zero and e_a is less than or equal to -970

fg_flag is set to 1 for any of the following conditions.

- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

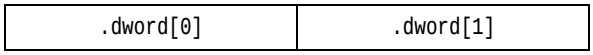
CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

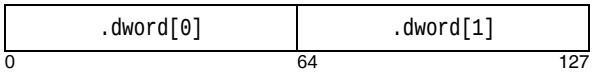
CR[BF]

VSR Data Layout for xvtdivdp

src1 = VSR[XA]

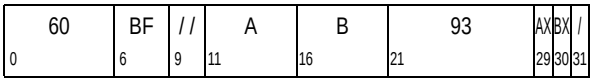


src2 = VSR[XB]



VSX Vector Test for software Divide Single-Precision XX3-form

xvtdivsp BF,XA,XB




```
XA      ← AX || A
XB      ← BX || B
eq_flag ← 0b0
gt_flag ← 0b0

do i=0 to 127 by 32
   src1       ← VSR[XA]{i:i+31}
   src2       ← VSR[XB]{i:i+31}
   e_a        ← src1{1:8} - 127
   e_b        ← src2{1:8} - 127
   fe_flag    ← fe_flag | IsNaN(src1) | IsInf(src1) |
                IsNaN(src2) | IsInf(src2) | IsZero(src2) |
                ( e_b <= -126 ) |
                ( e_b >=  125 ) |
                ( !IsZero(src1) & ( (e_a - e_b) >=  127 ) ) |
                ( !IsZero(src1) & ( (e_a - e_b) <= -125 ) ) |
                ( !IsZero(src1) & ( e_a <= -103 ) )
   fg_flag    ← fg_flag | IsInf(src1) | IsInf(src2) |
                IsZero(src2) | IsDen(src2)
end

fl_flag ← xvredp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

Let e_a be the unbiased exponent of src1.

Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.

- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to -126.
- e_b is greater than or equal to 125.
- src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 127.
- src1 is not a zero and the difference, e_a - e_b, is less than or equal to -125.
- src1 is not a zero and e_a is less than or equal to -103.

fg_flag is set to 1 for any of the following conditions.

- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

VSR Data Layout for xvtdivsp

src1 = VSR[XA]

.word[0]	.word[1]	.word[2]	.word[3]

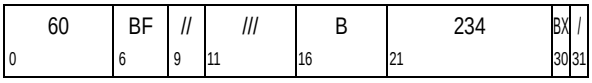
src2 = VSR[XB]

.word[0]	.word[1]	.word[2]	.word[3]

0 32 64 96 127

VSX Vector Test for software Square Root Double-Precision XX2-form

xvtsqrtdp BF,XB




```
XB      ← BX || B
fe_flag ← 0b0
fg_flag ← 0b0

do i=0 to 127 by 64
   src        ← VSR[XB]{i:i+63}
   e_b        ← src2{1:11} - 1023
   fe_flag    ← fe_flag | IsNaN(src) | IsInf(src)  |
                IsZero(src) | IsNeg(src) | ( e_a <= -970 )
   fg_flag    ← fg_flag | IsInf(src) | IsZero(src) |
                IsDen(src)
end

fl_flag ← xvrsqrtedp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XB be the value BX concatenated with B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.
- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -970.

fg_flag is set to 1 for the following condition.
- src is a zero, an infinity, or a denormalized value.

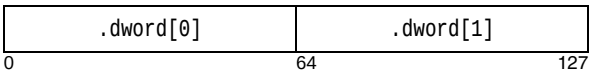
CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

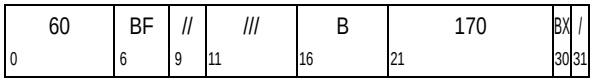
VSR Data Layout for xvtsqrtdp

src = VSR[XB]



VSX Vector Test for software Square Root Single-Precision XX2-form

xvtsqrtsp BF,XB




```
XB      ← BX || B
fe_flag ← 0b0
fg_flag ← 0b0

do i=0 to 127 by 32
   src      ← VSR[XB]{i:i+31}
   e_b      ← src2{1:8} - 127
   fe_flag  ← fe_flag | IsNaN(src) | IsInf(src) |
              IsZero(src) | IsNeg(src) | ( e_a <= -103 )
   fg_flag  ← fg_flag | IsInf(src) | IsZero(src) |
              IsDen(src)
end

fl_flag = xvrsqrtesp_error() <= 2^-14
CR[BF]  = 0b1 || fg_flag || fe_flag || 0b0
```

Let XB be the value BX concatenated with B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.
- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -103.

fg_flag is set to 1 for the following condition.
- src is a zero, an infinity, or a denormalized value.

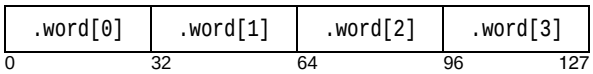
CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

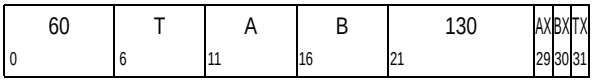
VSR Data Layout for xvtsqrtsp

src = VSR[XB]



VSX Logical AND XX3-form

xxland XT,XA,XB




```
XT      ← TX || T
XA      ← AX || A
XB      ← BX || B
VSR[XT] ← VSR[XA] & VSR[XB]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

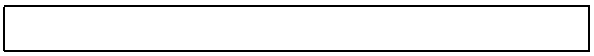
The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

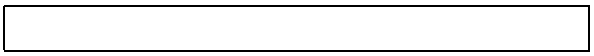
None

VSR Data Layout for xxland

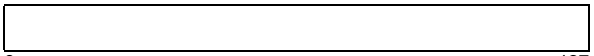
src1 = VSR[XA]



src2 = VSR[XB]



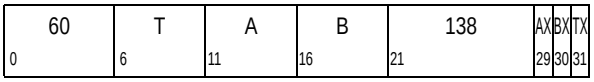
tgt = VSR[XT]



0 127

VSX Logical AND with Complement XX3-form

xxlandc XT,XA,XB




```
XT      ← TX || T
XA      ← AX || A
XB      ← BX || B
VSR[XT] ← VSR[XA] & ~VSR[XB]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

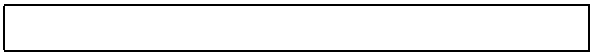
The contents of VSR[XA] are ANDed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

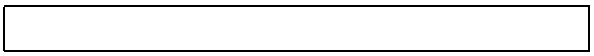
None

VSR Data Layout for xxland

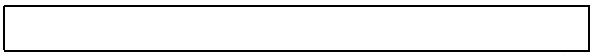
src1 = VSR[XA]



src2 = VSR[XB]



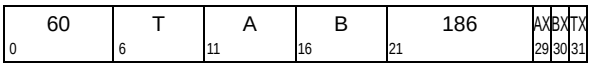
tgt = VSR[XT]



0 127

VSX Logical Equivalence XX3-form

xxleqv XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] ≡ VSR[32×BX+B]

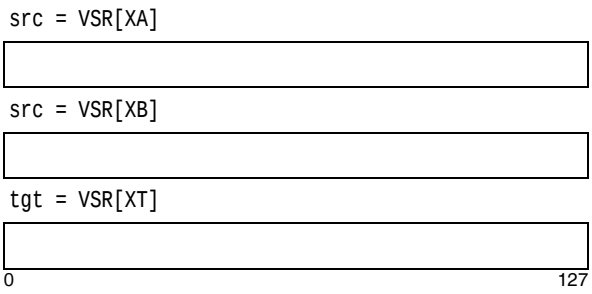
Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered:

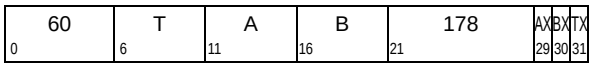
None

VSR Data Layout for xxleqv



VSX Logical NAND XX3-form

xxlnand XT,XA,XB



VSR[32×TX+T] ← ¬(VSR[32×AX+A] & VSR[32×BX+B])

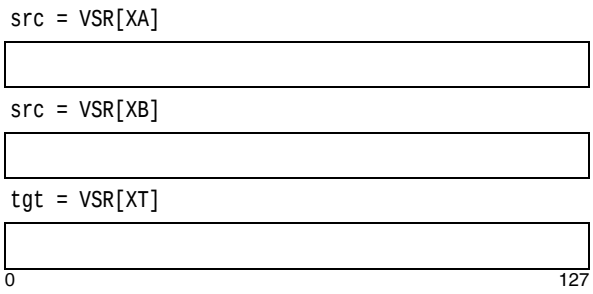
Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered:

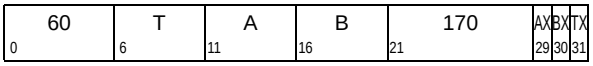
None

VSR Data Layout for xxlnand



VSX Logical OR with Complement XX3-form

xxlorc XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] | ¬VSR[32×BX+B]

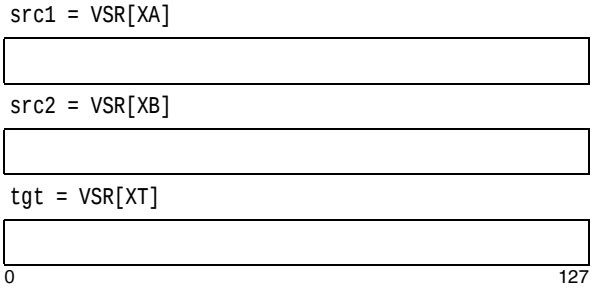
Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of VSR[XA] are ORed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered:

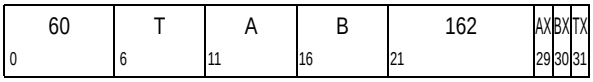
None

VSR Data Layout for xxlorc



VSX Logical NOR XX3-form

xxlnor XT,XA,XB



VSR[32×TX+T] ← ~(VSR[32×AX+A] | VSR[32×BX+B])

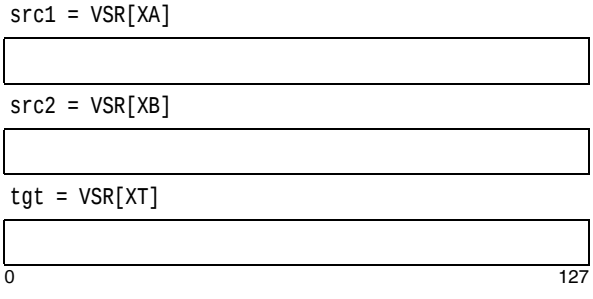
Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of VSR[XA] are ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered

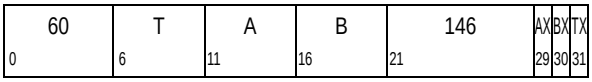
None

VSR Data Layout for xxlnor



VSX Logical OR XX3-form

xxlor XT,XA,XB




```
VSR[32×TX+T] ← VSR[32×AX+A] | VSR[32×BX+B]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

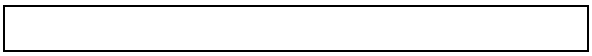
The contents of VSR[XA] are ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

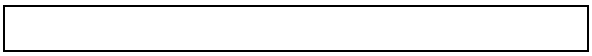
None

VSR Data Layout for xxlor

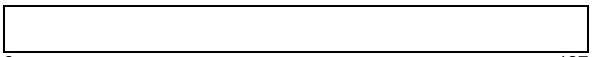
src1 = VSR[XA]



src2 = VSR[XB]



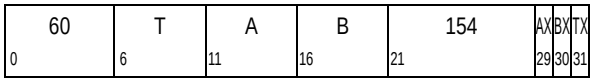
tgt = VSR[XT]



0 127

VSX Logical XOR XX3-form

xxlxor XT,XA,XB




```
VSR[32×TX+T] ← VSR[32×AX+A] ^ VSR[32×BX+B]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

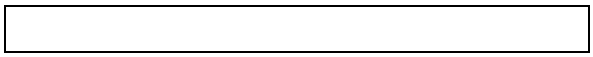
The contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

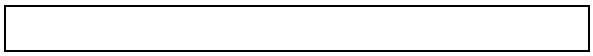
None

VSR Data Layout for xxlxor

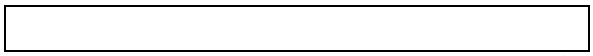
src1 = VSR[XA]



src2 = VSR[XB]



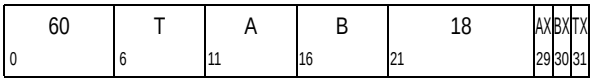
tgt = VSR[XT]



0 127

VSX Merge High Word XX3-form

xxmrghw XT,XA,XB




```
VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[0]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[0]
VSR[32×TX+T].word[2] ← VSR[32×AX+A].word[1]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[1]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of word element 0 of VSR[XA] are placed into word element 0 of VSR[XT].

The contents of word element 0 of VSR[XB] are placed into word element 1 of VSR[XT].

The contents of word element 1 of VSR[XA] are placed into word element 2 of VSR[XT].

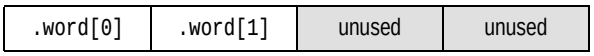
The contents of word element 1 of VSR[XB] are placed into word element 3 of VSR[XT].

Special Registers Altered

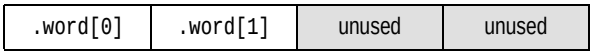
None

VSR Data Layout for xxmrghw

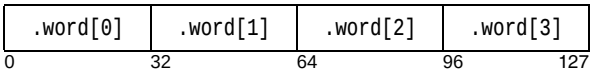
src1 = VSR[XA]



src2 = VSR[XB]

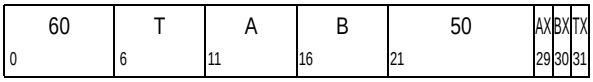


tgt = VSR[XT]



VSX Merge Low Word XX3-form

xxmrglw XT,XA,XB




```
VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[2]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[2]
VSR[32×TX+T].word[2] ← VSR[32×AX+A].word[3]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[3]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

The contents of word element 2 of VSR[XA] are placed into word element 0 of VSR[XT].

The contents of word element 2 of VSR[XB] are placed into word element 1 of VSR[XT].

The contents of word element 3 of VSR[XA] are placed into word element 2 of VSR[XT].

The contents of word element 3 of VSR[XB] are placed into word element 3 of VSR[XT].

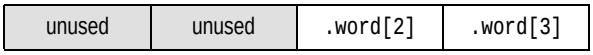
Special Registers Altered

None

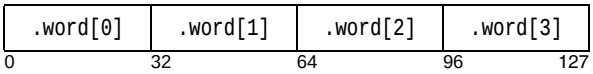
VSR Data Layout for xxmrglw

src1 = VSR[XA]

src2 = VSR[XB]

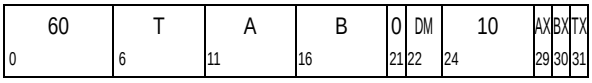


tgt = VSR[XT]



VSX Permute Doubleword Immediate XX3-form

xxpermdi XT,XA,XB,DM




```
VSR[32×TX+T].dword[0] ← VSR[32×AX+A].dword[DM.bit[0]]
VSR[32×TX+T].dword[1] ← VSR[32×BX+B].dword[DM.bit[1]]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

If DM.bit[0]=0, the contents of doubleword element 0 of VSR[XA] are placed into doubleword element 0 of VSR[XT]. Otherwise the contents of doubleword element 1 of VSR[XA] are placed into doubleword element 0 of VSR[XT].

If DM.bit[1]=0, the contents of doubleword element 0 of VSR[XB] are placed into doubleword element 1 of VSR[XT]. Otherwise the contents of doubleword element 1 of VSR[XB] are placed into doubleword element 1 of VSR[XT].

Special Registers Altered

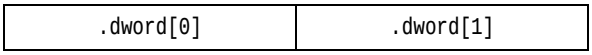
None

Extended Mnemonic		Equivalent To	
xxspltd	T,A,0	xxpermdi	T,A,A,0b00
xxspltd	T,A,1	xxpermdi	T,A,A,0b11
xxmrghd	T,A,B	xxpermdi	T,A,B,0b00
xxmrgld	T,A,B	xxpermdi	T,A,B,0b11
xxswapd	T,A	xxpermdi	T,A,A,0b10

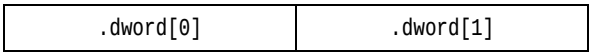
Table 113:

VSR Data Layout for xxpermdi

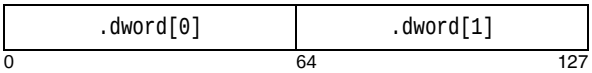
src1 = VSR[XA]



src2 = VSR[XB]



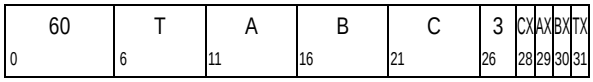
tgt = VSR[XT]



0 64 127

VSX Select XX4-form

xxsel XT,XA,XB,XC




```
do i=0 to 127
   if (VSR[32×CX+C].bit[i]=0) then
      VSR[32×TX+T].bit[i] ← VSR[32×AX+A].bit[i]
   else
      VSR[32×TX+T].bit[i] ← VSR[32×BX+B].bit[i]
end
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.
Let XC be the value CX concatenated with C.

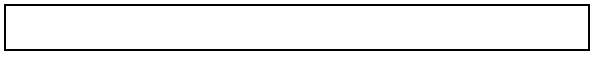
For each bit of VSR[XC] that contains the value 0, the corresponding bit of VSR[XA] is placed into the corresponding bit of VSR[XT]. Otherwise, the corresponding bit of VSR[XB] is placed into the corresponding bit of VSR[XT].

Special Registers Altered

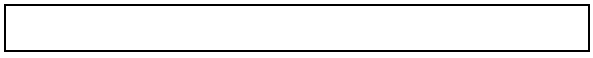
None

VSR Data Layout for xxsel

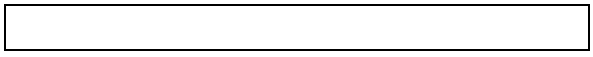
src1 = VSR[XA]



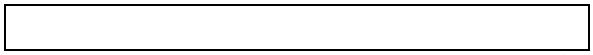
src2 = VSR[XB]



src3 = VSR[XC]



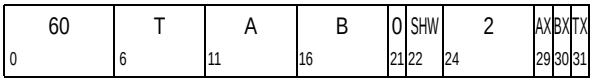
tgt = VSR[XT]



0 127

VSX Shift Left Double by Word Immediate XX3-form

xxsldwi XT,XA,XB,SHW




```
source.qword[0] ← VSR[32×AX+A]
source.qword[1] ← VSR[32×BX+B]
VSR[32×TX+T] ← source.word[SHW:SHW+3]
```

Let XT be the value TX concatenated with T.
Let XA be the value AX concatenated with A.
Let XB be the value BX concatenated with B.

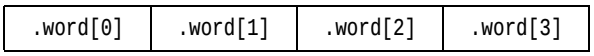
Let the source vector be the concatenation of the contents of VSR[XA] followed by the contents of VSR[XB]. Words SHW:SHW+3 of the source vector are placed into VSR[XT].

Special Registers Altered

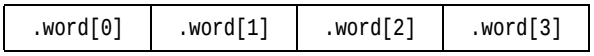
None

VSR Data Layout for xxsldwi

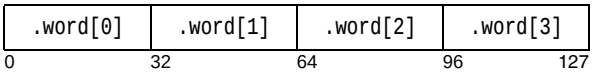
src1 = VSR[XA]



src2 = VSR[XB]

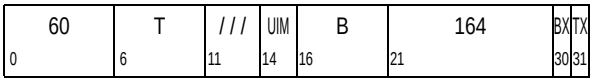


tgt = VSR[XT]



VSX Splat Word XX2-form

xxspltw XT,XB,UIM




```
VSR[32×TX+T].word[0] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[2] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[UIM]
```

Let XT be the value TX concatenated with T.
Let XB be the value BX concatenated with B.

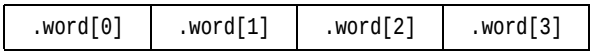
The contents of word element UIM of VSR[XB] are replicated in each word element of VSR[XT].

Special Registers Altered

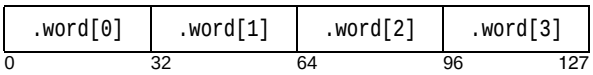
None

VSR Data Layout for xxspltw

src = VSR[XB]



tgt = VSR[XT]



Chapter 8. Signal Processing Engine (SPE) [Category: Signal Processing Engine]

8.1 Overview

The Signal Processing Engine (SPE) accelerates signal processing applications normally suited to DSP operation. This is accomplished using short vectors (two element) within 64-bit GPRs and using single instruction multiple data (SIMD) operations to perform the requisite computations. SPE also architects an Accumulator register to allow for back to back operations without loop unrolling.

8.2 Nomenclature and Conventions

Several conventions regarding nomenclature are used for SPE:

- The Signal Processing Engine category is abbreviated as SPE.
- Bits 0 to 31 of a 64-bit register are referenced as upper word, even word or high word element of the register. Bits 32:63 are referred to as lower word, odd word or low word element of the register. Each half is an element of a 64-bit GPR.
- Bits 0 to 15 and bits 32 to 47 are referenced as even halfwords. Bits 16 to 31 and bits 48 to 63 are referenced as odd halfwords.
- Mnemonics for SPE instructions generally begin with the letters 'ev' (embedded vector).

The RTL conventions in described below are used in addition to those described in Section 1.3:Additional RTL functions are described in Appendix D.

Notation **Meaning**

$\times_{sf}$ Signed fractional multiplication. Result of multiplying 2 signed fractional quantities having bit length n taking the least significant 2n-1 bits of the sign extended product and concatenating a 0 to the least significant bit forming a signed fractional result of 2n bits. Two 16-bit signed fractional quantities, a and b are multiplied, as shown below:

$ea_{0:31}$ = EXTS(a)
$eb_{0:31}$ = EXTS(b)
$prod_{0:63}$ = ea X eb
$eprod_{0:63}$ = EXTS($prod_{32:63}$)
$result_{0:31}$ = $eprod_{33:63}$ || 0b0

$\times_{gsf}$ Guarded signed fractional multiplication. Result of multiplying 2 signed fractional quantities having bit length 16 taking the least significant 31 bits of the sign extended product and concatenating a 0 to the least significant bit forming a guarded signed fractional result of 64 bits. Since guarded signed fractional multiplication produces a 64-bit result, fractional input quantities of -1 and -1 can produce +1 in the intermediate product. Two 16-bit fractional quantities, a and b are multiplied, as shown below:

$ea_{0:31}$ = EXTS(a)
$eb_{0:31}$ = EXTS(b)
$prod_{0:63}$ = ea X eb
$eprod_{0:63}$ = EXTS($prod_{32:63}$)
$result_{0:63}$ = $eprod_{1:63}$ || 0b0

<< Logical shift left. x << y shifts value x left by y bits, leaving zeros in the vacated bits.

>> Logical shift right. x >> y shifts value x right by y bits, leaving zeros in the vacated bits.

8.3 Programming Model

8.3.1 General Operation

SPE instructions generally take elements from one source register and operate on them with the corresponding elements of a second source register (and/or the accumulator) to produce results. Results are placed in the destination register and/or the accumulator. Instructions that are vector in nature (i.e. produce results of more than one element) provide results for each element that are independent of the computation of the other elements. These instructions can also be used to perform scalar DSP operations by ignoring the results of the upper 32-bit half of the register file.

There are no record forms of *SPE* instructions. As a result, the meaning of bits in the CR is different than for other categories. *SPE Compare* instructions specify a CR field, two source registers, and the type of compare: greater than, less than, or equal. Two bits of the CR field are written with the result of the vector compare, one for each element. The remaining two bits reflect the ANDing and ORing of the vector compare results.

8.3.2 GPR Registers

The SPE requires a GPR register file with thirty-two 64-bit registers. For 32-bit implementations, instructions that normally operate on a 32-bit register file access and change only the least significant 32-bits of the GPRs leaving the most significant 32-bits unchanged. For 64-bit implementations, operation of these instructions is unchanged, i.e. those instructions continue to operate on the 64-bit registers as they would if the SPE was not implemented. Most *SPE* instructions view the 64-bit register as being composed of a vector of two elements, each of which is 32 bits wide (some instructions read or write 16-bit elements). The most significant 32-bits are called the upper word, high word or even word. The least significant 32-bits are called the lower word, low word or odd word. Unless otherwise specified, *SPE* instructions write all 64-bits of the destination register.

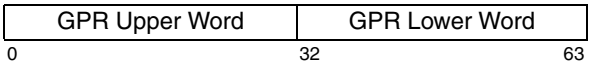


Figure 127.GPR

8.3.3 Accumulator Register

A partially visible accumulator register (ACC) is provided for some SPE instructions. The accumulator is a 64-bit register that holds the results of the *Multiply Accumulate* (*MAC*) forms of *SPE Fixed-Point* instructions. The accumulator allows the back-to-back execution of dependent *MAC* instructions, something that is found in the inner loops of DSP code such as FIR and FFT filters. The accumulator is partially visible to the programmer in the sense that its results do not have to be explicitly read to use them. Instead they are always copied into a 64-bit destination GPR which is specified as part of the instruction. Based upon the type of instruction, the accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

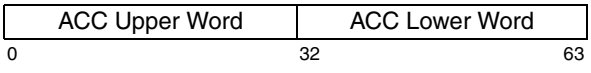


Figure 128.Accumulator

8.3.4 Signal Processing Embedded Floating-Point Status and Control Register (SPEFSCR)

Status and control for SPE uses the SPEFSCR register. This register is also used by the SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, and SPE.Embedded Float Vector categories. Status and control bits are shared with these categories. The SPEFSCR register is implemented as special purpose register (SPR) number 512 and is read and written by the ***mfspr*** and ***mtspr*** instructions. *SPE* instructions affect both the high element (bits 32:33) and low element status flags (bits 48:49) of the SPEFSCR.

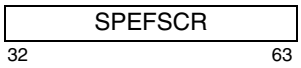


Figure 129. Signal Processing and Embedded Floating-Point Status and Control Register

The SPEFSCR bits are defined as shown below.

Bit	Description
32	***Summary Integer Overflow High*** (SOVH) SOVH is set to 1 when an *SPE* instruction sets OVH. This is a sticky bit.
33	***Integer Overflow High*** (OVH) OVH is set to 1 to indicate that an overflow has occurred in the upper element during execution of an *SPE* instruction. The bit is set to 1 if a result of an operation performed by the instruction cannot be represented in the number of bits into which the result is to be placed, and is set to 0 otherwise. The OVH bit is not altered by *Modulo* instructions, or by other instructions that cannot overflow.
34	***Embedded Floating-Point Guard Bit High*** (FGH) [Category: SP.FV] FGH is supplied for use by the Embedded Floating-Point Round interrupt handler. FGH is an extension of the low-order bits of the fractional result produced from an *SPE.Embedded Float Vector* instruction on the high word. FGH is zeroed if an overflow, underflow, or invalid input error is detected on the high element of an *SPE.Embedded Float Vector* instruction. Execution of an *SPE.Embedded Float Scalar* instruction leaves FGH undefined.
35	***Embedded Floating-Point Inexact Bit High*** (FXH) [Category: SP.FV] FXH is supplied for use by the Embedded Floating-Point Round interrupt handler. FXH is an extension of the low-order bits of the fractional result produced from an *SPE.Embedded Float Vector* instruction on the high word.

FXH represents the logical ‘or’ of all the bits shifted right from the Guard bit when the fractional result is normalized. FXH is zeroed if an overflow, underflow, or invalid input error is detected on the high element of an *SPE.Embedded Float Vector* instruction.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FXH undefined.

36 ***Embedded Floating-Point Invalid Operation/Input Error High*** (FINVH) [Category: SP.FV]
The FINVH bit is set to 1 if any high word operand of an *SPE.Embedded Float Vector* instruction is infinity, NaN, or a denormalized value, or if the instruction is a divide and the dividend and divisor are both 0, or if a conversion to integer or fractional value overflows.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FINVH undefined.

37 ***Embedded Floating-Point Divide By Zero High*** (FDBZH) [Category: SP.FV]
The FDBZH bit is set to 1 when an *SPE.Embedded Vector Floating-Point Divide* instruction is executed with a divisor of 0 in the high word operand, and the dividend is a finite nonzero number.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FDBZH undefined.

38 ***Embedded Floating-Point Underflow High*** (FUNFH) [Category: SP.FV]
The FUNFH bit is set to 1 when the execution of an *SPE.Embedded Float Vector* instruction results in an underflow on the high word operation.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FUNFH undefined.

39 ***Embedded Floating-Point Overflow High*** (FOVFH) [Category: SP.FV]
The FOVFH bit is set to 1 when the execution of an *SPE.Embedded Float Vector* instruction results in an overflow on the high word operation.

Execution of an *SPE.Embedded Float Scalar* instruction leaves FOVFH undefined.

40:41 Reserved

42 ***Embedded Floating-Point Inexact Sticky Flag*** (FINXS) [Categories: SP.FV, SP.FD, SP.FS]
The FINXS bit is set to 1 whenever the execution of an *Embedded Floating-Point* instruction delivers an inexact result for either the low or high element and no Embedded Floating-Point Data interrupt is taken for either element, or if an *Embedded Floating-Point* instruction results in overflow (FOVF=1 or FOVFH=1), but Embedded Floating-Point Overflow exceptions are disabled (FOVFE=0), or if an *Embedded Floating-Point* instruction results in underflow (FUNF=1 or FUNFH=1), but Embedded Floating-Point Underflow exceptions are disabled (FUNFE=0), and no Embedded Floating-Point Data interrupt occurs. This is a sticky bit.

43 ***Embedded Floating-Point Invalid Operation/Input Sticky Flag*** (FINVS) [Categories: SP.FV, SP.FD, SP.FS]
The FINVS bit is defined to be the sticky result of any *Embedded Floating-Point* instruction that causes FINVH or FINV to be set to 1. That is, FINVS ← FINVS | FINV | FINVH. This is a sticky bit.

44 ***Embedded Floating-Point Divide By Zero Sticky Flag*** (FDBZS) [Categories: SP.FV, SP.FD, SP.FS]
The FDBZS bit is set to 1 when an *Embedded Floating-Point Divide* instruction sets FDBZH or FDBZ to 1. That is, FDBZS ← FDBZS | FDBZ | FDBZH. This is a sticky bit.

45 ***Embedded Floating-Point Underflow Sticky Flag*** (FUNFS) [Categories: SP.FV, SP.FD, SP.FS]
The FUNFS bit is defined to be the sticky result of any *Embedded Floating-Point* instruction that causes FUNFH or FUNF to be set to 1. That is, FUNFS ← FUNFS | FUNF | FUNFH. This is a sticky bit.

46 ***Embedded Floating-Point Overflow Sticky Flag*** (FOVFS) [Categories: SP.FV, SP.FD, SP.FS]
The FOVFS bit is defined to be the sticky result of any *Embedded Floating-Point* instruction that causes FOVH or FOVF to be set to 1. That is, FOVFS ← FOVFS | FOVF | FOVFH. This is a sticky bit.

47 Reserved

48 ***Summary Integer Overflow*** (SOV)
SOV is set to 1 when an *SPE* instruction sets OV to 1. This is a sticky bit.

49 ***Integer Overflow*** (OV)
OV is set to 1 to indicate that an overflow has occurred in the lower element during execution of an *SPE* instruction. The bit is set to 1 if a result of an operation performed by the instruction cannot be represented in the number of bits into which the result is to be placed, and is set to 0 otherwise. The OV bit is not altered by *Modulo* instructions, or by other instructions that cannot overflow.

50 ***Embedded Floating-Point Guard Bit (Low/scalar)*** (FG) [Categories: SP.FV, SP.FD, SP.FS]

FG is supplied for use by the Embedded Floating-Point Round interrupt handler. FG is an extension of the low-order bits of the fractional result produced from an *Embedded Floating-Point* instruction on the low word. FG is zeroed if an overflow, underflow, or invalid input error is detected on the low element of an *Embedded Floating-Point* instruction.

51 ***Embedded Floating-Point Inexact Bit (Low/scalar)*** (FX) [Categories: SP.FV, SP.FD, SP.FS]
FX is supplied for use by the Embedded Floating-Point Round interrupt handler. FX is an extension of the low-order bits of the fractional result produced from an *Embedded Floating-Point* instruction on the low word. FX represents the logical 'or' of all the bits shifted right from the Guard bit when the fractional result is normalized. FX is zeroed if an overflow, underflow, or invalid input error is detected on *Embedded Floating-Point* instruction

52 ***Embedded Floating-Point Invalid Operation/Input Error (Low/scalar)*** (FINV) [Categories: SP.FV, SP.FD, SP.FS]
The FINV bit is set to 1 if any low word operand of an *Embedded Floating-Point* instruction is infinity, NaN, or a denormalized value, or if the operation is a divide and the dividend and divisor are both 0, or if a conversion to integer or fractional value overflows.

53 ***Embedded Floating-Point Divide By Zero (Low/scalar)*** (FDBZ) [Categories: SP.FV, SP.FD, SP.FS]
The FDBZ bit is set to 1 when an *Embedded Floating-Point Divide* instruction is executed with a divisor of 0 in the low word operand, and the dividend is a finite nonzero number.

54 ***Embedded Floating-Point Underflow (Low/scalar)*** (FUNF) [Categories: SP.FV, SP.FD, SP.FS]
The FUNF bit is set to 1 when the execution of an *Embedded Floating-Point* instruction results in an underflow on the low word operation.

55 ***Embedded Floating-Point Overflow (Low/scalar)*** (FOVF) [Categories: SP.FV, SP.FD, SP.FS]
The FOVF bit is set to 1 when the execution of an *Embedded Floating-Point* instruction results in an overflow on the low word operation.

56 Reserved

57 ***Embedded Floating-Point Round (Inexact) Exception Enable*** (FINXE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

The Embedded Floating-Point Round interrupt is taken if the exception is enabled and if FG | FGH | FX | FXH (signifying an inexact result) is set to 1 as a result of an *Embedded Floating-Point* instruction.

If an *Embedded Floating-Point* instruction results in overflow or underflow and the corresponding Embedded Floating-Point Underflow or Embedded Floating-Point Overflow exception is disabled then the Embedded Floating-Point Round interrupt is taken.

58 ***Embedded Floating-Point Invalid Operation/Input Error Exception Enable*** (FINVE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FINV or FINVH bit is set to 1 by an *Embedded Floating-Point* instruction.

59 ***Embedded Floating-Point Divide By Zero Exception Enable*** (FDBZE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FDBZ or FDBZH bit is set to 1 by an *Embedded Floating-Point* instruction.

60 ***Embedded Floating-Point Underflow Exception Enable*** (FUNFE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FUNF or FUNFH bit is set to 1 by an *Embedded Floating-Point* instruction.

61 ***Embedded Floating-Point Overflow Exception Enable*** (FOVFE) [Categories: SP.FV, SP.FD, SP.FS]

0 Exception disabled
1 Exception enabled

If the exception is enabled, an Embedded Floating-Point Data interrupt is taken if the FOVF or FOVFH bit is set to 1 by an *Embedded Floating-Point* instruction.

62:63 ***Embedded Floating-Point Rounding Mode Control*** (FRMC) [Categories: SP.FV, SP.FD, SP.FS]

00 Round to Nearest

01 Round toward Zero
10 Round toward +Infinity
11 Round toward -Infinity

Programming Note

Rounding modes 0b10 (+Infinity) and 0b11 (-Infinity) may not be supported by some implementations. If an implementation does not support these, Embedded Floating-Point Round interrupts are generated for every *Embedded Floating-Point* instruction for which rounding is required when +Infinity or -Infinity modes are set and software is required to produce the correctly rounded result

8.3.5 Data Formats

The SPE provides two different data formats, integer and fractional. Both data formats can be treated as signed or unsigned quantities.

8.3.5.1 Integer Format

Unsigned integers consist of 16, 32, or 64-bit binary integer values. The largest representable value is 2^n-1 where n represents the number of bits in the value. The smallest representable value is 0. Computations that produce values larger than 2^n-1 or smaller than 0 may set OV or OVH in the SPEFSCR.

Signed integers consist of 16, 32, or 64-bit binary values in two's complement form. The largest representable value is 2^{n-1}-1 where n represents the number of bits in the value. The smallest representable value is -2^{n-1}. Computations that produce values larger than 2^{n-1}-1 or smaller than -2^{n-1} may set OV or OVH in the SPEFSCR.

8.3.5.2 Fractional Format

Fractional data format is conventionally used for DSP fractional arithmetic. Fractional data is useful for representing data converted from analog devices.

Unsigned fractions consist of 16, 32, or 64-bit binary fractional values that range from 0 to less than 1. Unsigned fractions place the radix point immediately to the left of the most significant bit. The most significant bit of the value represents the value 2^{-1}, the next most significant bit represents the value 2^{-2} and so on. The largest representable value is $1-2^{-n}$ where n represents the number of bits in the value. The smallest representable value is 0. Computations that produce values larger than $1-2^{-n}$ or smaller than 0 may set OV or OVH in the SPEFSCR. The SPE category does not define unsigned fractional forms of instructions to manipulate unsigned fractional data since the unsigned integer forms of the instructions produce the same results as would the unsigned fractional forms.

Guarded unsigned fractions are 64-bit binary fractional values. Guarded unsigned fractions place the decimal point immediately to the left of bit 32. The largest representable value is $2^{32}-2^{-32}$. The smallest representable value is 0. Guarded unsigned fractional computations are always modulo and do not set OV or OVH in the SPEFSCR.

Signed fractions consist of 16, 32, or 64-bit binary fractional values in two's-complement form that range from -1 to less than 1. Signed fractions place the decimal point immediately to the right of the most significant bit. The largest representable value is $1-2^{-(n-1)}$ where n represents the number of bits in the value. The smallest representable value is -1. Computations that produce values larger than $1-2^{-(n-1)}$or smaller than -1 may set OV or OVH in the SPEFSCR. Multiplication of two signed fractional values causes the result to be shifted left one bit to remove the resultant redundant sign bit in the product. In this case, a 0 bit is concatenated as the least significant bit of the shifted result.

Guarded signed fractions are 64-bit binary fractional values. Guarded signed fractions place the decimal point immediately to the left of bit 33. The largest representable value is $2^{32}-2^{-31}$. The smallest representable value is $-2^{32}-1+2^{-31}$. Guarded signed fractional computations are always modulo and do not set OV or OVH in the SPEFSCR.

8.3.6 Computational Operations

The SPE category supports several different computational capabilities. Both modulo and saturation results can be performed. Modulo results produce truncation of the overflow bits in a calculation, therefore overflow does not occur and no saturation is performed. For instructions for which overflow occurs, saturation provides a maximum or minimum representable value (for the data type) in the case of overflow. Instructions are provided for a wide range of computational capability. The operation types can be divided into 4 basic categories:

- *Simple Vector* instructions. These instructions use the corresponding low and high word elements of the operands to produce a vector result that is placed in the destination register, the accumulator, or both.
- *Multiply and Accumulate* instructions. These instructions perform multiply operations, optionally add the result to the accumulator, and place the result into the destination register and optionally into the accumulator. These instructions are composed of different multiply forms, data formats and data accumulate options. The mnemonics for these instructions indicate their various characteristics. These are shown in Table 114.
- *Load* and *Store* instructions. These instructions provide load and store capabilities for moving data

to and from memory. A variety of forms are provided that position data for efficient computation.

- *Compare* and miscellaneous instructions. These instructions perform miscellaneous functions such as field manipulation, bit reversed incrementing, and vector compares.

Table 114:Mnemonic Extensions for Multiply Accumulate Instructions

Extension	Meaning	Comments
Multiply Form		
he	halfword even	16 X 16 → 32
heg	halfword even guarded	16 X 16 → 32, 64-bit final accumulate result
ho	halfword odd	16 X 16 → 32
hog	halfword odd guarded	16 X 16 → 32, 64-bit final accumulate result
w	word	32 X 32 → 64
wh	word high	32 X 32 → 32 (high-order 32 bits of product)
wl	word low	32 X 32 → 32 (low-order 32 bits of product)
Data Format		
smf	signed modulo fractional	modulo, no saturation or overflow
smi	signed modulo integer	modulo, no saturation or overflow
ssf	signed saturate fractional	saturation on product and accumulate
ssi	signed saturate integer	saturation on product and accumulate
umi	unsigned modulo integer	modulo, no saturation or overflow
usi	unsigned saturate integer	saturation on product and accumulate
Accumulate Option		
a	place in accumulator	result → accumulator
aa	add to accumulator	accumulator + result → accumulator
aaw	add to accumulator as word elements	$accumulator_{0:31}$ + $result_{0:31}$ → $accumulator_{0:31}$ $accumulator_{32:63}$ + $result_{32:63}$ → $accumulator_{32:63}$
an	add negated to accumulator	accumulator - result → accumulator
anw	add negated to accumulator as word elements	$accumulator_{0:31}$ - $result_{0:31}$ → $accumulator_{0:31}$ $accumulator_{32:63}$ - $result_{32:63}$ → $accumulator_{32:63}$

8.3.7 SPE Instructions

8.3.8 Saturation, Shift, and Bit Reverse Models

For saturation, left shifts, and bit reversal, the pseudo RTL is provided here to more accurately describe those functions that are referenced in the instruction pseudo RTL.

8.3.8.1 Saturation

```
SATURATE(ov, carry, sat_ovn, sat_ov, val)
if ov then
    if carry then
        return sat_ovn
    else
        return sat_ov
else
    return val
```

8.3.8.2 Shift Left

```
SL(value, cnt)
if cnt > 31 then
    return 0
else
    return (value << cnt)
```

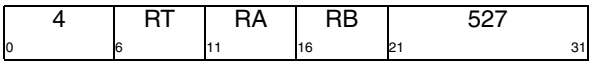
8.3.8.3 Bit Reverse

```
BITREVERSE(value)
result ← 0
mask ← 1
shift ← 31
cnt ← 32
while cnt > 0 then do
    t ← value & mask
    if shift >= 0 then
        result ← (t << shift) | result
    else
        result ← (t >> -shift) | result
    cnt ← cnt - 1
    shift ← shift - 2
    mask ← mask << 1
return result
```

8.3.9 SPE Instruction Set

Bit Reversed Increment *EVX-form*

brinc RT,RA,RB




```
n ← implementation-dependent number of mask bits
mask ← (RB)64-n:63
a ← (RA)64-n:63
d ← BITREVERSE(1 + BITREVERSE(a | (¬mask)))
RT ← (RA)0:63-n || (d & mask)
```

brinc computes a bit-reverse index based on the contents of RA and a mask specified in RB. The new index is written to RT.

The number of bits in the mask is implementation-dependent but may not exceed 32.

Special Registers Altered:

None

Programming Note

brinc provides a way for software to access FFT data in a bit-reversed manner. RA contains the index into a buffer that contains data on which FFT is to be performed. RB contains a mask that allows the index to be updated with bit-reversed addressing. Typically this instruction precedes a load with index instruction; for example,

```
brinc r2, r3, r4
lhax r8, r5, r2
```

RB contains a bit-mask that is based on the number of points in an FFT. To access a buffer containing n byte sized data that is to be accessed with bit-reversed addressing, the mask has $\log_2 n$ 1s in the least significant bit positions and 0s in the remaining most significant bit positions. If, however, the data size is a multiple of a halfword or a word, the mask is constructed so that the 1s are shifted left by $\log_2$ (size of the data) and 0s are placed in the least significant bit positions.

Programming Note

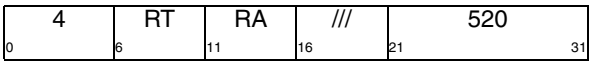
This instruction only modifies the lower 32 bits of the destination register in 32-bit implementations. For 64-bit implementations in 32-bit mode, the contents of the upper 32-bits of the destination register are undefined.

Programming Note

Execution of ***brinc*** does not cause SPE Unavailable exceptions regardless of MSR_{SPV}.

Vector Absolute Value *EVX-form*

evabs RT,RA




```
RT0:31 ← ABS((RA)0:31)
RT32:63 ← ABS((RA)32:63)
```

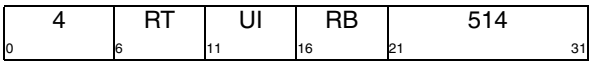
The absolute value of each element of RA is placed in the corresponding elements of RT. An absolute value of 0x8000_0000 (most negative number) returns 0x8000_0000.

Special Registers Altered:

None

Vector Add Immediate Word *EVX-form*

evaddiw RT,RB,UI




```
RT0:31 ← (RB)0:31 + EXTZ(UI)
RT32:63 ← (RB)32:63 + EXTZ(UI)
```

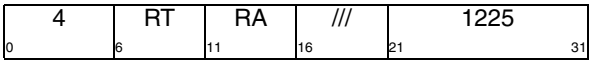
UI is zero-extended and added to both the high and low elements of RB and the results are placed in RT. Note that the same value is added to both elements of the register.

Special Registers Altered:

None

Vector Add Signed, Modulo, Integer to Accumulator Word *EVX-form*

evaddsmiaaw RT,RA




```
RT0:31 ← (ACC)0:31 + (RA)0:31
RT32:63 ← (ACC)32:63 + (RA)32:63
ACC0:63 ← (RT)0:63
```

Each word element in RA is added to the corresponding element in the accumulator and the results are placed in RT and into the accumulator.

Special Registers Altered:

ACC

Vector Add Signed, Saturate, Integer to Accumulator Word *EVX-form*

evaddssiaaw RT,RA

4	RT	RA	///	1217
0	6	11	16	21 31

$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) + EXTS((RA)_{0:31})$
$ovh \leftarrow temp_{31} \oplus temp_{32}$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$

$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) + EXTS((RA)_{32:63})$
$ovl \leftarrow temp_{31} \oplus temp_{32}$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$

$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

Each signed-integer word element in RA is sign-extended and added to the corresponding sign-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Add Unsigned, Modulo, Integer to Accumulator Word *EVX-form*

evaddumiaaw RT,RA

4	RT	RA	///	1224
0	6	11	16	21 31

$RT_{0:31} \leftarrow (ACC)_{0:31} + (RA)_{0:31}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + (RA)_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

Each unsigned-integer word element in RA is added to the corresponding element in the accumulator and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Add Unsigned, Saturate, Integer to Accumulator Word *EVX-form*

evaddusiaaw RT,RA

4	RT	RA	///	1216
0	6	11	16	21 31

$temp_{0:63} \leftarrow EXTZ((ACC)_{0:31}) + EXTZ((RA)_{0:31})$
$ovh \leftarrow temp_{31}$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

$temp_{0:63} \leftarrow EXTZ((ACC)_{32:63}) + EXTZ((RA)_{32:63})$
$ovl \leftarrow temp_{31}$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$

$ACC_{0:63} \leftarrow (RT)_{0:63}$

$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

Each unsigned-integer word element in RA is zero-extended and added to the corresponding zero-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Add Word *EVX-form*

evaddw RT,RA,RB

4	RT	RA	RB	512
0	6	11	16	21 31

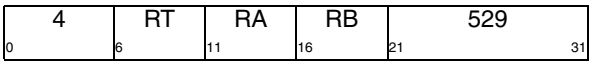
$RT_{0:31} \leftarrow (RA)_{0:31} + (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} + (RB)_{32:63}$

The corresponding elements of RA and RB are added and the results are placed in RT. The sum is a modulo sum.

Special Registers Altered:
None

Vector AND *EVX-form*

evand RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31}$ & $(RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63}$ & $(RB)_{32:63}$

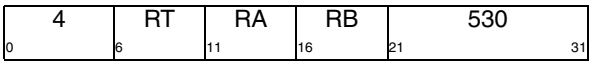
The corresponding elements of RA and RB are ANDed bitwise and the results are placed in the corresponding element of RT.

Special Registers Altered:

None

Vector AND with Complement *EVX-form*

evandc RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31}$ & $(\neg(RB)_{0:31})$
$RT_{32:63} \leftarrow (RA)_{32:63}$ & $(\neg(RB)_{32:63})$

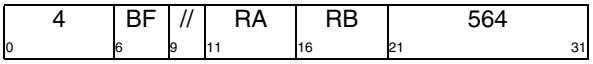
The word elements of RA are ANDed bitwise with the complement of the corresponding elements of RB. The results are placed in the corresponding element of RT.

Special Registers Altered:

None

Vector Compare Equal *EVX-form*

evcmpeq BF,RA,RB




```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

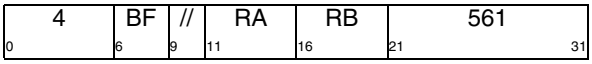
The most significant bit in BF is set if the high-order element of RA is equal to the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is equal to the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Compare Greater Than Signed *EVX-form*

evcmpgts BF,RA,RB




```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is greater than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is greater than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Compare Greater Than Unsigned EVX-form

evcmpgtu BF,RA,RB

4	BF	//	RA	RB	560
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah >u bh) then ch ← 1
else ch ← 0
if (al >u bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is greater than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is greater than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Compare Less Than Unsigned EVX-form

evcmpltu BF,RA,RB

4	BF	//	RA	RB	562
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah <u bh) then ch ← 1
else ch ← 0
if (al <u bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

The most significant bit in BF is set if the high-order element of RA is less than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is less than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Compare Less Than Signed EVX-form

evcmplts BF,RA,RB

4	BF	//	RA	RB	563
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF+32:4×BF+35 ← ch || cl || (ch | cl) || (ch & cl)
```

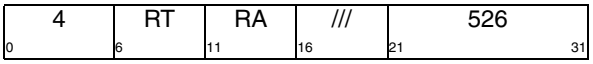
The most significant bit in BF is set if the high-order element of RA is less than the high-order element of RB; it is cleared otherwise. The next bit in BF is set if the low-order element of RA is less than the low-order element of RB and cleared otherwise. The last two bits of BF are set to the OR and AND of the result of the compare of the high and low elements.

Special Registers Altered:

CR field BF

Vector Count Leading Signed Bits Word *EVX-form*

evcntlsw RT,RA




```
n ← 0
s ← (RA)n
do while n < 32
   if (RA)n ≠ s then leave
   n ← n + 1
RT0:31 ← n
n ← 0
s ← (RA)n+32
do while n < 32
   if (RA)n+32 ≠ s then leave
   n ← n + 1
RT32:63 ← n
```

The leading sign bits in each element of RA are counted, and the respective count is placed into each element of RT.

Special Registers Altered:

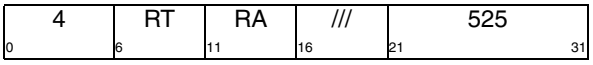
None

Programming Note

evcntlzw is used for unsigned operands; ***evcntlsw*** is used for signed operands.

Vector Count Leading Zeros Word *EVX-form*

evcntlzw RT,RA




```
n ← 0
do while n < 32
   if (RA)n = 1 then leave
   n ← n + 1
RT0:31 ← n
n ← 0
do while n < 32
   if (RA)n+32 = 1 then leave
   n ← n + 1
RT32:63 ← n
```

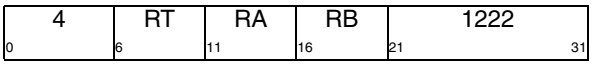
The leading zero bits in each element of RA are counted, and the respective count is placed into each element of RT.

Special Registers Altered:

None

Vector Divide Word Signed *EVX-form*

evdivws RT,RA,RB




```
ddh ← (RA)0:31
ddl ← (RA)32:63
dvh ← (RB)0:31
dvl ← (RB)32:63
RT0:31 ← ddh ÷ dvh
RT32:63 ← ddl ÷ dvl
ovh ← 0
ovl ← 0
if ((ddh < 0) & (dvh = 0)) then
   RT0:31 ← 0x8000_0000
   ovh ← 1
else if ((ddh >= 0) & (dvh = 0)) then
   RT0:31 ← 0x7FFFFFFF
   ovh ← 1
else if (ddh = 0x8000_0000)&(dvh = 0xFFFF_FFFF)
then
   RT0:31 ← 0x7FFFFFFF
   ovh ← 1
if ((ddl < 0) & (dvl = 0)) then
   RT32:63 ← 0x8000_0000
   ovl ← 1
else if ((ddl >= 0) & (dvl = 0)) then
   RT32:63 ← 0x7FFFFFFF
   ovl ← 1
else if (ddl = 0x8000_0000)&(dvl = 0xFFFF_FFFF)
then
   RT32:63 ← 0x7FFFFFFF
   ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of RA. The two divisors are the two elements of the contents of RB. The resulting two 32-bit quotients on each element are placed into RT. The remainders are not supplied. The operands and quotients are interpreted as signed integers.

Special Registers Altered:

OV OVH SOV SOVH

Programming Note

Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

Vector Divide Word Unsigned *EVX-form*

evdivwu RT,RA,RB

4	RT	RA	RB	1223
0	6	11	16	21 31

```
ddh ← (RA)0:31
ddl ←(RA)32:63
dvh ← (RB)0:31
dvl ← (RB)32:63
RT0:31 ← ddh ÷ dvh
RT32:63 ← ddl ÷ dvl
ovh ← 0
ovl ← 0
if (dvh = 0) then
   RT0:31 ← 0xFFFFFFFF
   ovh ← 1
if (dvl = 0) then
    RT32:63 ← 0xFFFFFFFF
    ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of RA. The two divisors are the two elements of the contents of RB. Two 32-bit quotients are formed as a result of the division on each of the high and low elements and the quotients are placed into RT. Remainders are not supplied. Operands and quotients are interpreted as unsigned integers.

Special Registers Altered:

OV OVH SOV SOVH

Programming Note

Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

Vector Equivalent *EVX-form*

eveqv RT,RA,RB

4	RT	RA	RB	537
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} \equiv (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \equiv (RB)_{32:63}$

The corresponding elements of RA and RB are XORed bitwise, and the complemented results are placed in RT.

Special Registers Altered:

None

Vector Extend Sign Byte *EVX-form*

evextsb RT,RA

4	RT	RA	///	522
0	6	11	16	21 31

$RT_{0:31} \leftarrow EXTS((RA)_{24:31})$
$RT_{32:63} \leftarrow EXTS((RA)_{56:63})$

The signs of the low-order byte in each of the elements in RA are extended, and the results are placed in RT.

Special Registers Altered:

None

Vector Extend Sign Halfword *EVX-form*

evextsh RT,RA

4	RT	RA	///	523
0	6	11	16	21 31

$RT_{0:31} \leftarrow EXTS((RA)_{16:31})$
$RT_{32:63} \leftarrow EXTS((RA)_{48:63})$

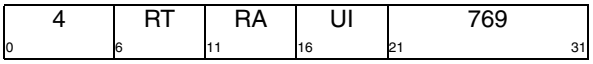
The signs of the odd halfwords in each of the elements in RA are extended, and the results are placed in RT.

Special Registers Altered:

None

Vector Load Double Word into Double Word EVX-form

evldd RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
RT ← MEM(EA, 8)
```

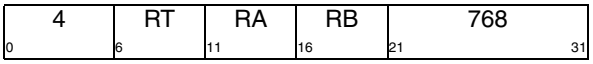
D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:

None

Vector Load Double Word into Double Word Indexed EVX-form

evlddx RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)
```

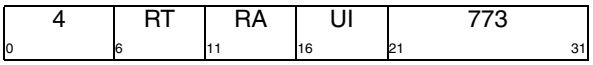
The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:

None

Vector Load Double into Four Halfwords EVX-form

evldh RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
RT0:15 ← MEM(EA, 2)
RT16:31 ← MEM(EA+2,2)
RT32:47 ← MEM(EA+4,2)
RT48:63 ← MEM(EA+6,2)
```

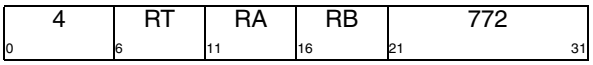
D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:

None

Vector Load Double into Four Halfwords Indexed EVX-form

evldhx RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA, 2)
RT16:31 ← MEM(EA+2,2)
RT32:47 ← MEM(EA+4,2)
RT48:63 ← MEM(EA+6,2)
```

The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:

None

Vector Load Double into Two Words *EVX-form*

evldw RT,D(RA)

4	RT	RA	UI	771
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
RT0:31 ← MEM(EA, 4)
RT32:63 ← MEM(EA+4, 4)
```

D in the instruction mnemonic is UI × 8. The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:

None

Vector Load Double into Two Words Indexed *EVX-form*

evldwx RT,RA,RB

4	RT	RA	RB	770
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← MEM(EA, 4)
RT32:63 ← MEM(EA+4, 4)
```

The doubleword addressed by EA is loaded from memory and placed in RT.

Special Registers Altered:

None

Vector Load Halfword into Halfwords Even and Splat *EVX-form*

evlhhesplat RT,D(RA)

4	RT	RA	UI	777
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×2)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA,2)
RT48:63 ← 0x0000
```

D in the instruction mnemonic is UI × 2. The halfword addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:

None

Vector Load Halfword into Halfwords Even and Splat Indexed *EVX-form*

evlhhesplatx RT,RA,RB

4	RT	RA	RB	776
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA,2)
RT48:63 ← 0x0000
```

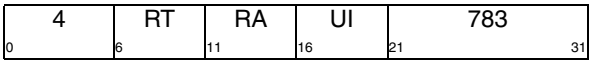
The halfword addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:

None

Vector Load Halfword into Halfword Odd Signed and Splat EVX-form

evlhhossplat RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×2)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA,2))
```

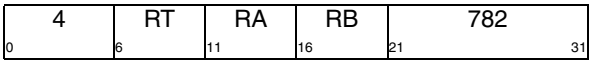
D in the instruction mnemonic is UI × 2. The halfword addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:

None

Vector Load Halfword into Halfword Odd Signed and Splat Indexed EVX-form

evlhhossplatx RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA,2))
```

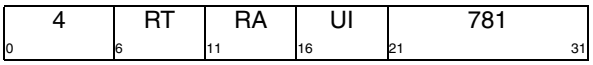
The halfword addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:

None

Vector Load Halfword into Halfword Odd Unsigned and Splat EVX-form

evlhhousplat RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×2)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA,2))
```

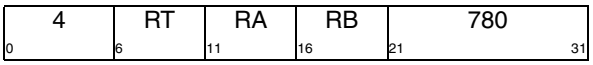
D in the instruction mnemonic is UI × 2. The halfword addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:

None

Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed EVX-form

evlhhousplatx RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA,2))
```

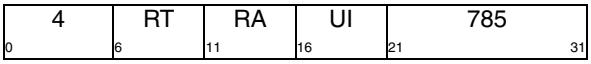
The halfword addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:

None

Vector Load Word into Two Halfwords Even *EVX-form*

evlwhe RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA+2,2)
RT48:63 ← 0x0000
```

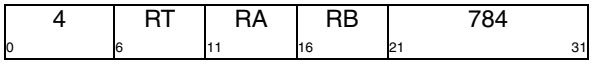
D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in the even halfwords of each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:

None

Vector Load Word into Two Halfwords Even Indexed *EVX-form*

evlwhex RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA,2)
RT16:31 ← 0x0000
RT32:47 ← MEM(EA+2,2)
RT48:63 ← 0x0000
```

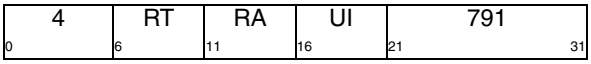
The word addressed by EA is loaded from memory and placed in the even halfwords in each element of RT. The odd halfwords of each element of RT are set to 0.

Special Registers Altered:

None

Vector Load Word into Two Halfwords Odd Signed (with sign extension) *EVX-form*

evlwhos RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA+2,2))
```

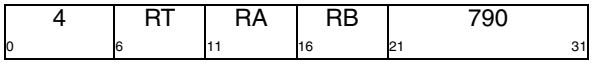
D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:

None

Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension) *EVX-form*

evlwhosx RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTS(MEM(EA,2))
RT32:63 ← EXTS(MEM(EA+2,2))
```

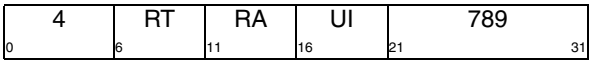
The word addressed by EA is loaded from memory and placed in the odd halfwords sign extended in each element of RT.

Special Registers Altered:

None

Vector Load Word into Two Halfwords Odd Unsigned (zero-extended) EVX-form

evlwhou RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA+2,2))
```

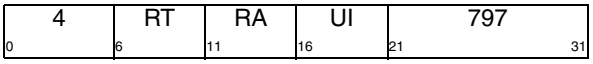
D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:

None

Vector Load Word into Two Halfwords and Splat EVX-form

evlwhsplat RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:15 ← MEM(EA,2)
RT16:31 ← MEM(EA,2)
RT32:47 ← MEM(EA+2,2)
RT48:63 ← MEM(EA+2,2)
```

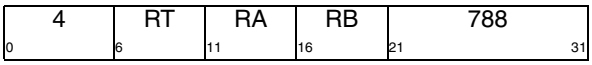
D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in both the even and odd halfwords in each element of RT.

Special Registers Altered:

None

Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended) EVX-form

evlwhoux RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← EXTZ(MEM(EA,2))
RT32:63 ← EXTZ(MEM(EA+2,2))
```

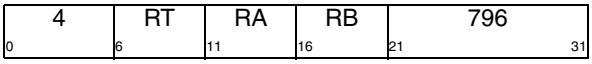
The word addressed by EA is loaded from memory and placed in the odd halfwords zero-extended in each element of RT.

Special Registers Altered:

None

Vector Load Word into Two Halfwords and Splat Indexed EVX-form

evlwhsplatx RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:15 ← MEM(EA,2)
RT16:31 ← MEM(EA,2)
RT32:47 ← MEM(EA+2,2)
RT48:63 ← MEM(EA+2,2)
```

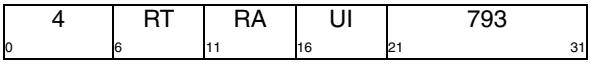
The word addressed by EA is loaded from memory and placed in both the even and odd halfwords in each element of RT.

Special Registers Altered:

None

Vector Load Word into Word and Splat *EVX-form*

evlwwsplat RT,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
RT0:31 ← MEM(EA,4)
RT32:63 ← MEM(EA,4)
```

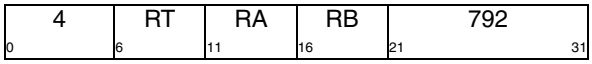
D in the instruction mnemonic is UI × 4. The word addressed by EA is loaded from memory and placed in both elements of RT.

Special Registers Altered:

None

Vector Load Word into Word and Splat Indexed *EVX-form*

evlwwsplatx RT,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
RT0:31 ← MEM(EA,4)
RT32:63 ← MEM(EA,4)
```

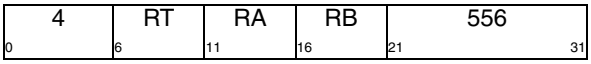
The word addressed by EA is loaded from memory and placed in both elements of RT.

Special Registers Altered:

None

Vector Merge High *EVX-form*

evmergehi RT,RA,RB




```
RT0:31 ← (RA)0:31
RT32:63 ← (RB)0:31
```

The high-order elements of RA and RB are merged and placed in RT.

Special Registers Altered:

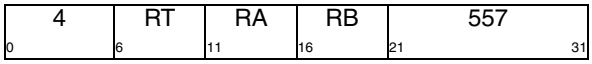
None

Programming Note

A vector splat high can be performed by specifying the same register in RA and RB.

Vector Merge Low *EVX-form*

evmergelo RT,RA,RB




```
RT0:31 ← (RA)32:63
RT32:63 ← (RB)32:63
```

The low-order elements of RA and RB are merged and placed in RT.

Special Registers Altered:

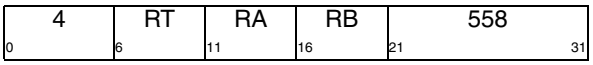
None

Programming Note

A vector splat low can be performed by specifying the same register in RA and RB.

Vector Merge High/Low *EVX-form*

evmergehilo RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31}$
$RT_{32:63} \leftarrow (RB)_{32:63}$

The high-order element of RA and the low-order element of RB are merged and placed in RT.

Special Registers Altered:

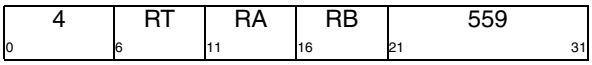
None

Programming Note

With appropriate specification of RA and RB, ***evmergehi***, ***evmergelo***, ***evmergehilo***, and ***evmergelohi*** provide a full 32-bit permute of two source operands.

Vector Merge Low/High *EVX-form*

evmergelohi RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{32:63}$
$RT_{32:63} \leftarrow (RB)_{0:31}$

The low-order element of RA and the high-order element of RB are merged and placed in RT.

Special Registers Altered:

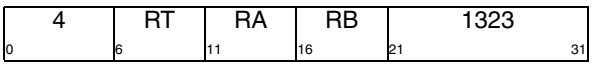
None

Programming Note

A vector swap can be performed by specifying the same register in RA and RB.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate *EVX-form*

evmhegsmfaa RT,RA,RB



$temp_{0:63} \leftarrow (RA)_{32:47} \times_{gsf} (RB)_{32:47}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator

Special Registers Altered:

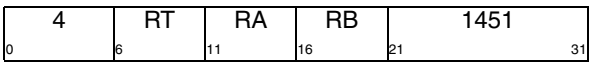
ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative *EVX-form*

evmhegsmfan RT,RA,RB



$temp_{0:63} \leftarrow (RA)_{32:47} \times_{gsf} (RB)_{32:47}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate EVX-form

evmhegsmiaa RT,RA,RB

4	RT	RA	RB	1321
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended and added to the contents of the 64-bit accumulator, and the resulting sum is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative EVX-form

evmhegsmian RT,RA,RB

4	RT	RA	RB	1449
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended and subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate EVX-form

evmhegumiaa RT,RA,RB

4	RT	RA	RB	1320
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative EVX-form

evmhegumian RT,RA,RB

4	RT	RA	RB	1448
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low even-numbered unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional *EVX-form*

evmhesmf RT,RA,RB

4	RT	RA	RB	1035
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied then placed into the corresponding words of RT.

Special Registers Altered:

None

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator *EVX-form*

evmhesmfa RT,RA,RB

4	RT	RA	RB	1067
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$

$RT_{32:63} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied then placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words *EVX-form*

evmhesmfaaw RT,RA,RB

4	RT	RA	RB	1291
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words *EVX-form*

evmhesmfanw RT,RA,RB

4	RT	RA	RB	1419
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{sf} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$

$temp_{0:31} \leftarrow (RA)_{32:47} \times_{sf} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32-bit intermediate products are subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer EVX-form

evmhesmi RT,RA,RB

4	RT	RA	RB	1033
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:

None

Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator EVX-form

evmhesmia RT,RA,RB

4	RT	RA	RB	1065
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words EVX-form

evmhesmiaaw RT,RA,RB

4	RT	RA	RB	1289
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words EVX-form

evmhesmianw RT,RA,RB

4	RT	RA	RB	1417
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{si} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

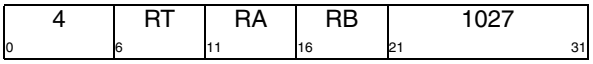
For each word element in the accumulator, the corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional EVX-form

evmhessf RT,RA,RB




```
temp0:31 ← (RA)0:15 ×sf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)32:47 ×sf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

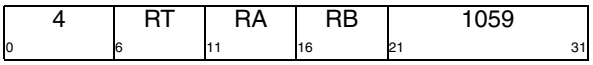
The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator EVX-form

evmhessfa RT,RA,RB




```
temp0:31 ← (RA)0:15 ×sf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)32:47 ×sf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
ACC0:63 ← (RT)0:63
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words *EVX-form*

evmhessfaaw RT,RA,RB

4	RT	RA	RB	1283
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 xsf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)

temp0:31 ← (RA)32:47 xsf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words *EVX-form*

evmhessfanw RT,RA,RB

4	RT	RA	RB	1411
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 xsf (RB)0:15
if ((RA)0:15 = 0x8000) & ((RB)0:15 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)32:47 xsf (RB)32:47
if ((RA)32:47 = 0x8000) & ((RB)32:47 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words *EVX-form*

evmhessiaaw RT,RA,RB

4	RT	RA	RB	1281
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 xsi (RB)0:15
temp0:63 ← EXTS((ACC)0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)

temp0:31 ← (RA)32:47 xsi (RB)32:47
temp0:63 ← EXTS((ACC)32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words *EVX-form*

evmhessianw RT,RA,RB

4	RT	RA	RB	1409
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 xsi (RB)0:15
temp0:63 ← EXTS((ACC)0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)32:47 xsi (RB)32:47
temp0:63 ← EXTS((ACC)32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← RT0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding even-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer EVX-form

evmheumi RT,RA,RB

4	RT	RA	RB	1032
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$

The corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:

None

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words EVX-form

evmheumiaaw RT,RA,RB

4	RT	RA	RB	1288
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words and the sums are placed into the corresponding RT and accumulator words.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator EVX-form

evmheumia RT,RA,RB

4	RT	RA	RB	1064
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{32:63} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words EVX-form

evmheumianw RT,RA,RB

4	RT	RA	RB	1416
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{0:15} \times_{ui} (RB)_{0:15}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding RT and accumulator words.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words *EVX-form*

evmheusiaaw RT,RA,RB

4	RT	RA	RB	1280
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 xui (RB)0:15
temp0:63 ← EXTZ((ACC)0:31) + EXTZ(temp0:31)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF,
                  temp32:63)
temp0:31 ← (RA)32:47 xui (RB)32:47
temp0:63 ← EXTZ((ACC)32:63) + EXTZ(temp0:31)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF,
                   0xFFFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words *EVX-form*

evmheusianw RT,RA,RB

4	RT	RA	RB	1408
0	6	11	16	21 31

```
temp0:31 ← (RA)0:15 xui (RB)0:15
temp0:63 ← EXTZ((ACC)0:31) - EXTZ(temp0:31)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000,
                  temp32:63)
temp0:31 ← (RA)32:47 xui (RB)32:47
temp0:63 ← EXTZ((ACC)32:63) - EXTZ(temp0:31)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0x0000_0000,
                   0x0000_0000, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate *EVX-form*

evmhogsmfaa RT,RA,RB

4	RT	RA	RB	1327
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{48:63} \times_{gsf} (RB)_{48:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative *EVX-form*

evmhogsmfan RT,RA,RB

4	RT	RA	RB	1455
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{48:63} \times_{gsf} (RB)_{48:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered, halfword signed fractional elements in RA and RB are multiplied using guarded signed fractional multiplication producing a sign extended 64-bit fractional product with the decimal between bits 32 and 33. The product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Note

If the two input operands are both -1.0, the intermediate product is represented as +1.0.

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate *EVX-form*

evmhogsmiaa RT,RA,RB

4	RT	RA	RB	1325
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low odd-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended to 64 bits then added to the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative *EVX-form*

evmhogsmian RT,RA,RB

4	RT	RA	RB	1453
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

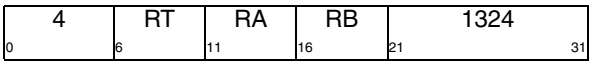
The corresponding low odd-numbered halfword signed-integer elements in RA and RB are multiplied. The intermediate product is sign-extended to 64 bits then subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate *EVX-form*

evmhogumiaa RT,RA,RB



$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

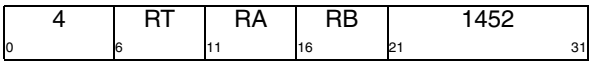
The corresponding low odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended to 64 bits then added to the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative *EVX-form*

evmhogumian RT,RA,RB



$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

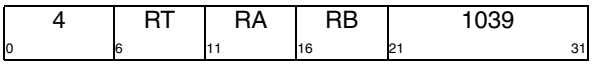
The corresponding low odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The intermediate product is zero-extended to 64 bits then subtracted from the contents of the 64-bit accumulator, and the result is placed in RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional *EVX-form*

evmhosmf RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$

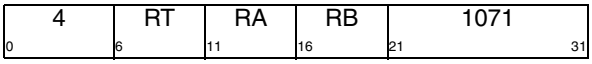
The corresponding odd-numbered, halfword signed fractional elements in RA and RB are multiplied. Each product is placed into the corresponding words of RT.

Special Registers Altered:

None

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator *EVX-form*

evmhosmfa RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding odd-numbered, halfword signed fractional elements in RA and RB are multiplied. Each product is placed into the corresponding words of RT. and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words *EVX-form*

evmhosmfaaw RT,RA,RB

4	RT	RA	RB	1295
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are added to the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words *EVX-form*

evmhosmfanw RT,RA,RB

4	RT	RA	RB	1423
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{sf} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{sf} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each intermediate product are subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer *EVX-form*

evmhosmi RT,RA,RB

4	RT	RA	RB	1037
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:

None

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator *EVX-form*

evmhosmia RT,RA,RB

4	RT	RA	RB	1069
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

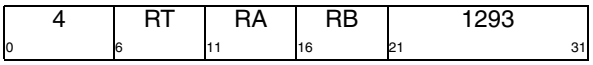
The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words *EVX-form*

evmhosmiaaw RT,RA,RB



$temp_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

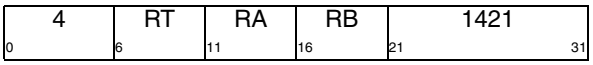
For each word element in the accumulator, the corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is added to the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words *EVX-form*

evmhosmianw RT,RA,RB



$temp_{0:31} \leftarrow (RA)_{16:31} \times_{si} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied. Each intermediate 32-bit product is subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding RT words and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional EVX-form

evmhossf RT,RA,RB

4	RT	RA	RB	1031
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator EVX-form

evmhossfa RT,RA,RB

4	RT	RA	RB	1063
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
ACC0:63 ← (RT)0:63
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied. The 32 bits of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words *EVX-form*

evmhossfaaw RT,RA,RB

4	RT	RA	RB	1287
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words *EVX-form*

evmhossfanw RT,RA,RB

4	RT	RA	RB	1415
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 ×sf (RB)16:31
if ((RA)16:31 = 0x8000) & ((RB)16:31 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   movh ← 0
temp0:63 ← EXTS((ACC)0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 ×sf (RB)48:63
if ((RA)48:63 = 0x8000) & ((RB)48:63 = 0x8000) then
   temp0:31 ← 0x7FFF_FFFF
   movl ← 1
else
   movl ← 0
temp0:63 ← EXTS((ACC)32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh | movh
SPEFSCROV ← ovl| movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered halfword signed fractional elements in RA and RB are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words *EVX-form*

evmhossiaaw RT,RA,RB

4	RT	RA	RB	1285
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 xsi (RB)16:31
temp0:63 ← EXTS((ACC)0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 xsi (RB)48:63
temp0:63 ← EXTS((ACC)32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer *EVX-form*

evmhoumi RT,RA,RB

4	RT	RA	RB	1036
0	6	11	16	21 31

```
RT0:31 ← (RA)16:31 xui (RB)16:31
RT32:63 ← (RA)48:63 xui (RB)48:63
```

The corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into the corresponding words of RT.

Special Registers Altered:

None

Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words *EVX-form*

evmhossianw RT,RA,RB

4	RT	RA	RB	1413
0	6	11	16	21 31

```
temp0:31 ← (RA)16:31 xsi (RB)16:31
temp0:63 ← EXTS((ACC)0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:31 ← (RA)48:63 xsi (RB)48:63
temp0:63 ← EXTS((ACC)32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered halfword signed-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator *EVX-form*

evmhoumia RT,RA,RB

4	RT	RA	RB	1068
0	6	11	16	21 31

```
RT0:31 ← (RA)16:31 xui (RB)16:31
RT32:63 ← (RA)48:63 xui (RB)48:63
ACC0:63 ← (RT)0:63
```

The corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. The two 32-bit products are placed into RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words *EVX-form*

evmhoumiaaw RT,RA,RB

4	RT	RA	RB	1292
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{ui} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding RT and accumulator words.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words *EVX-form*

evmhousiaaw RT,RA,RB

4	RT	RA	RB	1284
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{ui} (RB)_{16:31}$
$temp_{0:63} \leftarrow EXTZ((ACC)_{0:31}) + EXTZ(temp_{0:31})$
$ovh \leftarrow temp_{31}$
$RT_{0:31} \leftarrow SATURATE(ovh, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTZ((ACC)_{32:63}) + EXTZ(temp_{0:31})$
$ovl \leftarrow temp_{31}$
$RT_{32:63} \leftarrow SATURATE(ovl, 0, 0xFFFF\_FFFF, 0xFFFF\_FFFF, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each word element in the accumulator, corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words *EVX-form*

evmhoumianw RT,RA,RB

4	RT	RA	RB	1420
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{ui} (RB)_{16:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{0:31}$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator word. The results are placed into the corresponding RT and accumulator words.

Special Registers Altered:

ACC

Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words *EVX-form*

evmhousianw RT,RA,RB

4	RT	RA	RB	1412
0	6	11	16	21 31

$temp_{0:31} \leftarrow (RA)_{16:31} \times_{ui} (RB)_{16:31}$
$temp_{0:63} \leftarrow EXTZ((ACC)_{0:31}) - EXTZ(temp_{0:31})$
$ovh \leftarrow temp_{31}$
$RT_{0:31} \leftarrow SATURATE(ovh, 0, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$
$temp_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:63} \leftarrow EXTZ((ACC)_{32:63}) - EXTZ(temp_{0:31})$
$ovl \leftarrow temp_{31}$
$RT_{32:63} \leftarrow SATURATE(ovl, 0, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

For each word element in the accumulator, corresponding odd-numbered halfword unsigned-integer elements in RA and RB are multiplied producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Initialize Accumulator EVX-form

evmra RT,RA

4	RT	RA	///	1220
0	6	11	16	21 31

$ACC_{0:63} \leftarrow (RA)_{0:63}$
$RT_{0:63} \leftarrow (RA)_{0:63}$

The contents of RA are placed into the accumulator and RT. This is the method for initializing the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word High Signed, Modulo, Fractional EVX-form

evmwhsmf RT,RA,RB

4	RT	RA	RB	1103
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{sf} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$

The corresponding word signed fractional elements in RA and RB are multiplied and bits 0:31 of the two products are placed into the two corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Word High Signed, Modulo, Integer EVX-form

evmwhsmi RT,RA,RB

4	RT	RA	RB	1101
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$

The corresponding word signed-integer elements in RA and RB are multiplied. Bits 0:31 of the two 64-bit products are placed into the two corresponding words of RT.

Special Registers Altered:
None

Vector Multiply Word High Signed, Modulo, Fractional to Accumulator EVX-form

evmwhsmfa RT,RA,RB

4	RT	RA	RB	1135
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{sf} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding word signed fractional elements in RA and RB are multiplied and bits 0:31 of the two products are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word High Signed, Modulo, Integer to Accumulator EVX-form

evmwhsmia RT,RA,RB

4	RT	RA	RB	1133
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{0:31}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{0:31}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding word signed-integer elements in RA and RB are multiplied. Bits 0:31 of the two 64-bit products are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word High Signed, Saturate, Fractional EVX-form

evmwhssf RT,RA,RB

4	RT	RA	RB	1095
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 ×sf (RB)0:31
if ((RA)0:31 = 0x8000_0000)&((RB)0:31 = 0x8000_0000)
then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63 = 0x8000_0000 &(RB)32:63 = 0x8000_0000)
then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding word signed fractional elements in RA and RB are multiplied. Bits 0:31 of each product are placed into the corresponding words of RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

OV OVH SOV SOVH

Vector Multiply Word High Signed, Saturate, Fractional to Accumulator EVX-form

evmwhssfa RT,RA,RB

4	RT	RA	RB	1127
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 ×sf (RB)0:31
if ((RA)0:31 = 0x8000_0000) &((RB)0:31 =0x8000_0000)
then
   RT0:31 ← 0x7FFF_FFFF
   movh ← 1
else
   RT0:31 ← temp0:31
   movh ← 0
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63=0x8000_0000)&((RB)32:63=0x8000_0000)
then
   RT32:63 ← 0x7FFF_FFFF
   movl ← 1
else
   RT32:63 ← temp0:31
   movl ← 0
ACC0:63 ← (RT)0:63
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding word signed fractional elements in RA and RB are multiplied. Bits 0:31 of each product are placed into the corresponding words of RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Word High Unsigned, Modulo, Integer EVX-form

evmwhumi RT,RA,RB

4	RT	RA	RB	1100
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 ×ui (RB)0:31
RT0:31 ← temp0:31
temp0:63 ← (RA)32:63 ×ui (RB)32:63
RT32:63 ← temp0:31
```

The corresponding word unsigned-integer elements in RA and RB are multiplied. Bits 0:31 of the two products are placed into the two corresponding words of RT.

Special Registers Altered:

None

Vector Multiply Word High Unsigned, Modulo, Integer to AccumulatorEVX-form

evmwhumia RT,RA,RB

4	RT	RA	RB	1132
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 ×ui (RB)0:31
RT0:31 ← temp0:31
temp0:63 ← (RA)32:63 ×ui (RB)32:63
RT32:63 ← temp0:31
ACC0:63 ← (RT)0:63
```

The corresponding word unsigned-integer elements in RA and RB are multiplied. Bits 0:31 of the two products are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words *EVX-form*

evmwlsmiaaw RT,RA,RB

4	RT	RA	RB	1353
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word signed-integer elements in RA and RB are multiplied. The least significant 32 bits of each intermediate product are added to the contents of the corresponding accumulator words, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words *EVX-form*

evmwlsmianw RT,RA,RB

4	RT	RA	RB	1481
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word elements in RA and RB are multiplied. The least significant 32 bits of each intermediate product are subtracted from the contents of the corresponding accumulator words and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words *EVX-form*

evmwlssiaaw RT,RA,RB

4	RT	RA	RB	1345
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) + EXTS(temp_{32:63})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) + EXTS(temp_{32:63})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding word signed-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words *EVX-form*

evmwlssianw RT,RA,RB

4	RT	RA	RB	1473
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{si} (RB)_{0:31}$
$temp_{0:63} \leftarrow EXTS((ACC)_{0:31}) - EXTS(temp_{32:63})$
$ovh \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) - EXTS(temp_{32:63})$
$ovl \leftarrow (temp_{31} \oplus temp_{32})$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x8000\_0000, 0x7FFF\_FFFF, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$
$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

The corresponding word signed-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Word Low Unsigned, Modulo, Integer *EVX-form*

evmwlumi RT,RA,RB

4	RT	RA	RB	1096
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{32:63}$

The corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of RT.

Special Registers Altered:

None

Programming Note

The least significant 32 bits of the product are independent of whether the word elements in RA and RB are treated as signed or unsigned 32-bit integers.

Note that ***evmwlumi*** can be used for signed or unsigned integers.

Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator *EVX-form*

evmwlumia RT,RA,RB

4	RT	RA	RB	1128
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of RT and into the accumulator.

Special Registers Altered:

ACC

Programming Note

The least significant 32 bits of the product are independent of whether the word elements in RA and RB are treated as signed or unsigned 32-bit integers.

Note that ***evmwlumia*** can be used for signed or unsigned integers.

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words *EVX-form*

evmwlumiaaw RT,RA,RB

4	RT	RA	RB	1352
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} + temp_{32:63}$

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} + temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are added to the contents of the corresponding accumulator word and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words *EVX-form*

evmwlumianw RT,RA,RB

4	RT	RA	RB	1480
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{0:31} \times_{ui} (RB)_{0:31}$
$RT_{0:31} \leftarrow (ACC)_{0:31} - temp_{32:63}$
$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - temp_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

For each word element in the accumulator, the corresponding word unsigned-integer elements in RA and RB are multiplied. The least significant 32 bits of each product are subtracted from the contents of the corresponding accumulator word and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words *EVX-form*

evmwlusiaaw RT,RA,RB

4	RT	RA	RB	1344
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 xui (RB)0:31
temp0:63 ← EXTZ((ACC)0:31) + EXTZ(temp32:63)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF,
                  temp32:63)
temp0:63 ← (RA)32:63 xui (RB)32:63
temp0:63 ← EXTZ((ACC)32:63) + EXTZ(temp32:63)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF,
                   0xFFFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Word Signed, Modulo, Fractional *EVX-form*

evmwsmf RT,RA,RB

4	RT	RA	RB	1115
0	6	11	16	21 31

```
RT0:63 ← (RA)32:63 xsf (RB)32:63
```

The corresponding low word signed fractional elements in RA and RB are multiplied. The product is placed in RT.

Special Registers Altered:

None

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words *EVX-form*

evmwlusianw RT,RA,RB

4	RT	RA	RB	1472
0	6	11	16	21 31

```
temp0:63 ← (RA)0:31 xui (RB)0:31
temp0:63 ← EXTZ((ACC)0:31) - EXTZ(temp32:63)
ovh ← temp31
RT0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000,
                  temp32:63)
temp0:63 ← (RA)32:63 xui (RB)32:63
temp0:63 ← EXTZ((ACC)32:63) - EXTZ(temp32:63)
ovl ← temp31
RT32:63 ← SATURATE(ovl, 0, 0x0000_0000,
                   0x0000_0000, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned-integer elements in RA and RB are multiplied producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator saturating if overflow occurs, and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Multiply Word Signed, Modulo, Fractional to Accumulator *EVX-form*

evmwsmfa RT,RA,RB

4	RT	RA	RB	1147
0	6	11	16	21 31

```
RT0:63 ← (RA)32:63 xsf (RB)32:63
ACC0:63 ← (RT)0:63
```

The corresponding low word signed fractional elements in RA and RB are multiplied. The product is placed in RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Signed, Modulo, Fractional and Accumulate *EVX-form*

evmwsmfaa RT,RA,RB

4	RT	RA	RB	1371
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low word signed fractional elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Integer *EVX-form*

evmwsmi RT,RA,RB

4	RT	RA	RB	1113
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$

The low word signed-integer elements in RA and RB are multiplied. The product is placed in RT.

Special Registers Altered:
None

Vector Multiply Word Signed, Modulo, Integer and Accumulate *EVX-form*

evmwsmiaa RT,RA,RB

4	RT	RA	RB	1369
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word signed-integer elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative *EVX-form*

evmwsmfan RT,RA,RB

4	RT	RA	RB	1499
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{sf} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The corresponding low word signed fractional elements in RA and RB are multiplied. The intermediate product is subtracted from the contents of the accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Integer to Accumulator *EVX-form*

evmwsmia RT,RA,RB

4	RT	RA	RB	1145
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word signed-integer elements in RA and RB are multiplied. The product is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative *EVX-form*

evmwsmian RT,RA,RB

4	RT	RA	RB	1497
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{si} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word signed-integer elements in RA and RB are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:
ACC

Vector Multiply Word Signed, Saturate, Fractional *EVX-form*

evmwssf RT,RA,RB

4	RT	RA	RB	1107
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 xsf (RB)32:63
if ((RA)32:63 = 0x8000_0000) & (RB32:63 = 0x8000_0000)
then
   RT0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   RT0:63 ← temp0:63
   mov ← 0
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | mov
```

The low word signed fractional elements in RA and RB are multiplied. The 64-bit product is placed in RT. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

OV OVH SOV

Vector Multiply Word Signed, Saturate, Fractional to Accumulator *EVX-form*

evmwssfa RT,RA,RB

4	RT	RA	RB	1139
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 xsf (RB)32:63
if ((RA)32:63=0x8000_0000)&((RB)32:63=0x8000_0000)
then
   RT0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   RT0:63 ← temp0:63
   mov ← 0
ACC0:63 ← (RT)0:63
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | mov
```

The low word signed fractional elements in RA and RB are multiplied. The 64-bit product is placed in RT and into the accumulator. If both inputs are -1.0, the result saturates to the largest positive signed fraction.

Special Registers Altered:

ACC OV OVH SOV

Vector Multiply Word Signed, Saturate, Fractional and Accumulate EVX-form

evmwssfaa RT,RA,RB

4	RT	RA	RB	1363
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63=0x8000_0000)&((RB)32:63=0x8000_0000)
then
   temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   mov ← 0
temp0:64 ← EXTS((ACC)0:63) + EXTS(temp0:63)
ov ← (temp0 ⊕ temp1)
RT0:63 ← temp1:64

ACC0:63 ← (RT)0:63
SPEFSCROVH ← 0
SPEFSCROV ← ov | mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in RA and RB are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then added to the accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV

Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative EVX-form

evmwssfan RT,RA,RB

4	RT	RA	RB	1491
0	6	11	16	21 31

```
temp0:63 ← (RA)32:63 ×sf (RB)32:63
if ((RA)32:63=0x8000_0000)&((RB)32:63=0x8000_0000)
then
   temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF
   mov ← 1
else
   mov ← 0
temp0:64 ← EXTS((ACC)0:63) - EXTS(temp0:63)
ov ← (temp0 ⊕ temp1)
RT0:63 ← temp1:64
ACC0:63 ← (RT)0:63
SPEFSCROVH ← 0
SPEFSCROV ← ov | mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in RA and RB are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then subtracted from the accumulator and the result is placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV

Vector Multiply Word Unsigned, Modulo, Integer EVX-form

evmwumi RT,RA,RB

4	RT	RA	RB	1112
0	6	11	16	21 31

```
RT0:63 ← (RA)32:63 ×ui (RB)32:63
```

The low word unsigned-integer elements in RA and RB are multiplied to form a 64-bit product that is placed in RT.

Special Registers Altered:

None

Vector Multiply Word Unsigned, Modulo, Integer to Accumulator EVX-form

evmwumia RT,RA,RB

4	RT	RA	RB	1144
0	6	11	16	21 31

```
RT0:63 ← (RA)32:63 ×ui (RB)32:63
ACC0:63 ← (RT)0:63
```

The low word unsigned-integer elements in RA and RB are multiplied to form a 64-bit product that is placed in RT and into the accumulator.

Special Registers Altered:

ACC

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate *EVX-form*

evmwumiaa RT,RA,RB

4	RT	RA	RB	1368
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} + temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word unsigned-integer elements in RA and RB are multiplied. The intermediate product is added to the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and in RT.

Special Registers Altered:

ACC

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative *EVX-form*

evmwumian RT,RA,RB

4	RT	RA	RB	1496
0	6	11	16	21 31

$temp_{0:63} \leftarrow (RA)_{32:63} \times_{ui} (RB)_{32:63}$
$RT_{0:63} \leftarrow (ACC)_{0:63} - temp_{0:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

The low word unsigned-integer elements in RA and RB are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and in RT.

Special Registers Altered:

ACC

Vector NAND *EVX-form*

evnand RT,RA,RB

4	RT	RA	RB	542
0	6	11	16	21 31

$RT_{0:31} \leftarrow \neg((RA)_{0:31} \ \& \ (RB)_{0:31})$
$RT_{32:63} \leftarrow \neg((RA)_{32:63} \ \& \ (RB)_{32:63})$

Each element of RA and RB is bitwise NANDed. The result is placed in the corresponding element of RT.

Special Registers Altered:

None

Vector Negate *EVX-form*

evneg RT,RA

4	RT	RA	///	521
0	6	11	16	21 31

$RT_{0:31} \leftarrow NEG((RA)_{0:31})$
$RT_{32:63} \leftarrow NEG((RA)_{32:63})$

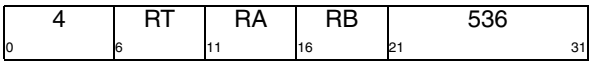
The negative of each element of RA is placed in RT. The negative of 0x8000_0000 (most negative number) returns 0x8000_0000.

Special Registers Altered:

None

Vector NOR *EVX-form*

evnor RT,RA,RB



$RT_{0:31} \leftarrow \neg((RA)_{0:31} \mid (RB)_{0:31})$
$RT_{32:63} \leftarrow \neg((RA)_{32:63} \mid (RB)_{32:63})$

Each element of RA and RB is bitwise NORed. The result is placed in the corresponding element of RT.

Special Registers Altered:

None

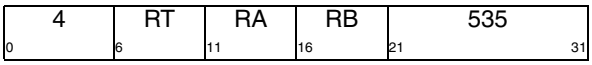
Extended Mnemonics:

Extended mnemonics are provided for the *Vector NOR* instruction to produce a vector bitwise complement operation.

Extended:	**Equivalent to:**
evnot RT,RA	***evnor*** RT,RA,RA

Vector OR *EVX-form*

evor RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31} \mid (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \mid (RB)_{32:63}$

Each element of RA and RB is bitwise ORed. The result is placed in the corresponding element of RT.

Special Registers Altered:

None

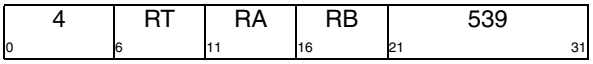
Extended Mnemonics:

Extended mnemonics are provided for the *Vector OR* instruction to provide a 64-bit vector move instruction.

Extended:	**Equivalent to:**
evmr RT,RA	***evor*** RT,RA,RA

Vector OR with Complement *EVX-form*

evorc RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31} \mid (\neg(RB)_{0:31})$
$RT_{32:63} \leftarrow (RA)_{32:63} \mid (\neg(RB)_{32:63})$

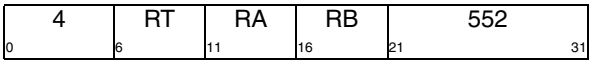
Each element of RA is bitwise ORed with the complement of RB. The result is placed in the corresponding element of RT.

Special Registers Altered:

None

Vector Rotate Left Word *EVX-form*

evrlw RT,RA,RB



$nh \leftarrow (RB)_{27:31}$
$nl \leftarrow (RB)_{59:63}$
$RT_{0:31} \leftarrow ROTL((RA)_{0:31}, nh)$
$RT_{32:63} \leftarrow ROTL((RA)_{32:63}, nl)$

Each of the high and low elements of RA is rotated left by an amount specified in RB. The result is placed in RT. Rotate values for each element of RA are found in bit positions $RB_{27:31}$ and $RB_{59:63}$.

Special Registers Altered:

None

Vector Rotate Left Word Immediate *EVX-form*

evrlwi RT,RA,UI

4	RT	RA	UI	554
0	6	11	16	21 31

```
n ← UI
RT0:31 ← ROTL((RA)0:31, n)
RT32:63 ← ROTL((RA)32:63, n)
```

Both the high and low elements of RA are rotated left by an amount specified by UI.

Special Registers Altered:

None

Vector Select *EVS-form*

evsel RT,RA,RB,BFA

4	RT	RA	RB	79	BFA
0	6	11	16	21	29 31

```
ch ← CRBFA×4
cl ← CRBFA×4+1
if (ch = 1) then RT0:31 ← (RA)0:31
else RT0:31 ← (RB)0:31
if (cl = 1) then RT32:63 ← (RA)32:63
else RT32:63 ← (RB)32:63
```

If the most significant bit in the BFA field of CR is set to 1, the high-order element of RA is placed in the high-order element of RT; otherwise, the high-order element of RB is placed into the high-order element of RT. If the next most significant bit in the BFA field of CR is set to 1, the low-order element of RA is placed in the low-order element of RT, otherwise, the low-order element of RB is placed into the low-order element of RT.

Special Registers Altered:

None

Vector Round Word *EVX-form*

evrndw RT,RA

4	RT	RA	///	524
0	6	11	16	21 31

```
RT0:31 ← ((RA)0:31+0x00008000) & 0xFFFF0000
RT32:63 ← ((RA)32:63+0x00008000) & 0xFFFF0000
```

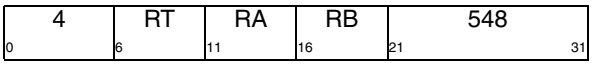
The 32-bit elements of RA are rounded into 16 bits. The result is placed in RT. The resulting 16 bits are placed in the most significant 16 bits of each element of RT, zeroing out the low-order 16 bits of each element.

Special Registers Altered:

None

Vector Shift Left Word *EVX-form*

evslw RT,RA,RB




```
nh ← (RB)26:31
nl ← (RB)58:63
RT0:31 ← SL((RA)0:31, nh)
RT32:63 ← SL((RA)32:63, nl)
```

Each of the high and low elements of RA is shifted left by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63.

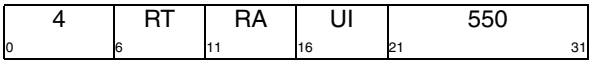
Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

None

Vector Shift Left Word Immediate *EVX-form*

evslwi RT,RA,UI




```
n ← UI
RT0:31 ← SL((RA)0:31, n)
RT32:63 ← SL((RA)32:63, n)
```

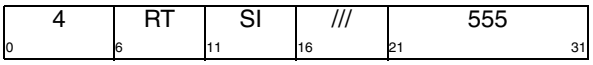
Both high and low elements of RA are shifted left by the 5-bit UI value and the results are placed in RT.

Special Registers Altered:

None

Vector Splat Fractional Immediate *EVX-form*

evsplatfi RT,SI




```
RT0:31 ← SI || 27 0
RT32:63 ← SI || 27 0
```

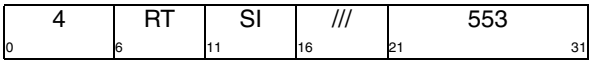
The value specified by SI is padded with trailing zeros and placed in both elements of RT. The SI ends up in bit positions $RT_{0:4}$ and $RT_{32:36}$.

Special Registers Altered:

None

Vector Splat Immediate *EVX-form*

evsplati RT,SI




```
RT0:31 ← EXTS(SI)
RT32:63 ← EXTS(SI)
```

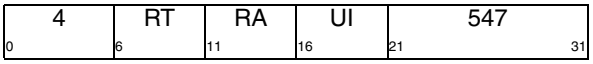
The value specified by SI is sign extended and placed in both elements of RT.

Special Registers Altered:

None

Vector Shift Right Word Immediate Signed *EVX-form*

evsrwis RT,RA,UI




```
n ← UI
RT0:31 ← EXTS((RA)0:31-n)
RT32:63 ← EXTS((RA)32:63-n)
```

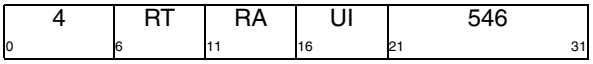
Both high and low elements of RA are shifted right by the 5-bit UI value. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

Special Registers Altered:

None

Vector Shift Right Word Immediate Unsigned *EVX-form*

evsrwiu RT,RA,UI




```
n ← UI
RT0:31 ← EXTZ((RA)0:31-n)
RT32:63 ← EXTZ((RA)32:63-n)
```

Both high and low elements of RA are shifted right by the 5-bit UI value; zeros are shifted into the most significant position.

Special Registers Altered:

None

Vector Shift Right Word Signed EVX-form

evsrws RT,RA,RB

4	RT	RA	RB	545
0	6	11	16	21 31

```
nh ← (RB)26:31
nl ← (RB)58:63
RT0:31 ← EXTS((RA)0:31-nh)
RT32:63 ← EXTS((RA)32:63-nl)
```

Both the high and low elements of RA are shifted right by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63. The sign bits are shifted into the most significant position.

Shift amounts from 32 to 63 give a result of 32 sign bits.

Special Registers Altered:

None

Vector Shift Right Word Unsigned EVX-form

evsrwu RT,RA,RB

4	RT	RA	RB	544
0	6	11	16	21 31

```
nh ← (RB)26:31
nl ← (RB)58:63
RT0:31 ← EXTZ((RA)0:31-nh)
RT32:63 ← EXTZ((RA)32:63-nl)
```

Both the high and low elements of RA are shifted right by an amount specified in RB. The result is placed in RT. The separate shift amounts for each element are specified by 6 bits in RB that lie in bit positions 26:31 and 58:63. Zeros are shifted into the most significant position.

Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

None

Vector Store Double of Double EVX-form

evstdd RS,D(RA)

4	RS	RA	UI	801
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ←b + EXTZ(UI×8)
MEM(EA,8) ← (RS)0:63
```

D in the instruction mnemonic is UI × 8. The contents of RS are stored as a doubleword in storage addressed by EA.

Special Registers Altered:

None

Vector Store Double of Double Indexed EVX-form

evstddx RS,RA,RB

4	RS	RA	RB	800
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ←b + (RB)
MEM(EA,8) ← (RS)0:63
```

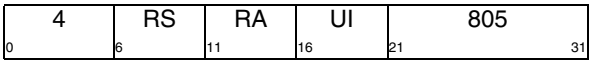
The contents of RS are stored as a doubleword in storage addressed by EA.

Special Registers Altered:

None

Vector Store Double of Four Halfwords *EVX-form*

evstdh RS,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)16:31
MEM(EA+4,2) ← (RS)32:47
MEM(EA+6,2) ← (RS)48:63
```

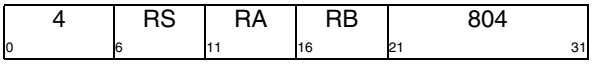
D in the instruction mnemonic is UI × 8. The contents of RS are stored as four halfwords in storage addressed by EA.

Special Registers Altered:

None

Vector Store Double of Four Halfwords Indexed *EVX-form*

evstdhx RS,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)16:31
MEM(EA+4,2) ← (RS)32:47
MEM(EA+6,2) ← (RS)48:63
```

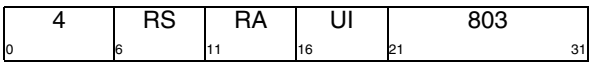
The contents of RS are stored as four halfwords in storage addressed by EA.

Special Registers Altered:

None

Vector Store Double of Two Words *EVX-form*

evstdw RS,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×8)
MEM(EA,4) ← (RS)0:31
MEM(EA+4,4) ← (RS)32:63
```

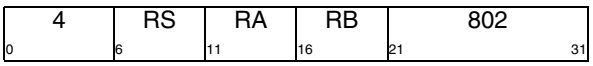
D in the instruction mnemonic is UI × 8. The contents of RS are stored as two words in storage addressed by EA.

Special Registers Altered:

None

Vector Store Double of Two Words Indexed *EVX-form*

evstdwx RS,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,4) ← (RS)0:31
MEM(EA+4,4) ← (RS)32:63
```

The contents of RS are stored as two words in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Two Halfwords from Even *EVX-form*

evstwhe RS,D(RA)

4	RS	RA	UI	817
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)32:47
```

D in the instruction mnemonic is UI × 4. The even halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Two Halfwords from Even Indexed *EVX-form*

evstwhex RS,RA,RB

4	RS	RA	RB	816
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)0:15
MEM(EA+2,2) ← (RS)32:47
```

The even halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Two Halfwords from Odd *EVX-form*

evstwho RS,D(RA)

4	RS	RA	UI	821
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
MEM(EA,2) ← (RS)16:31
MEM(EA+2,2) ← (RS)48:63
```

D in the instruction mnemonic is UI × 4. The odd halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Two Halfwords from Odd Indexed *EVX-form*

evstwhox RS,RA,RB

4	RS	RA	RB	820
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)16:31
MEM(EA+2,2) ← (RS)48:63
```

The odd halfwords from each element of RS are stored as two halfwords in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Word from Even *EVX-form*

evstwwe RS,D(RA)

4	RS	RA	UI	825
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
MEM(EA,4) ← (RS)0:31
```

D in the instruction mnemonic is UI × 4. The even word of RS is stored in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Word from Even Indexed *EVX-form*

evstwwex RS,RA,RB

4	RS	RA	RB	824
0	6	11	16	21 31

```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,4) ← (RS)0:31
```

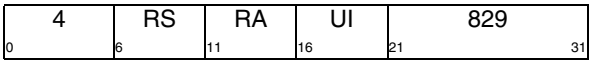
The even word of RS is stored in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Word from Odd *EVX-form*

evstwwo RS,D(RA)




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + EXTZ(UI×4)
MEM(EA,4) ← (RS)32:63
```

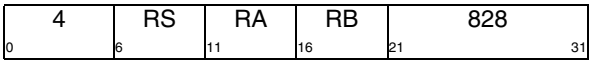
D in the instruction mnemonic is UI × 4. The odd word of RS is stored in storage addressed by EA.

Special Registers Altered:

None

Vector Store Word of Word from Odd Indexed *EVX-form*

evstwwox RS,RA,RB




```
if (RA = 0) then b ← 0
else b ← (RA)
EA ← b + (RB)
MEM(EA,4) ← (RS)32:63
```

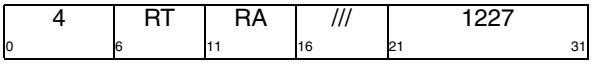
The odd word of RS is stored in storage addressed by EA.

Special Registers Altered:

None

Vector Subtract Signed, Modulo, Integer to Accumulator Word *EVX-form*

evsubfsmiaaw RT,RA




```
RT0:31 ← (ACC)0:31 - (RA)0:31
RT32:63 ← (ACC)32:63 - (RA)32:63
ACC0:63 ← (RT)0:63
```

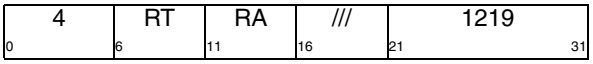
Each word element in RA is subtracted from the corresponding element in the accumulator and the difference is placed into the corresponding RT word and into the accumulator.

Special Registers Altered:

ACC

Vector Subtract Signed, Saturate, Integer to Accumulator Word *EVX-form*

evsubfssiaaw RT,RA




```
temp0:63 ← EXTS((ACC)0:31) - EXTS((RA)0:31)
ovh ← temp31 ⊕ temp32
RT0:31 ← SATURATE(ovh, temp31, 0x8000_0000,
                  0x7FFF_FFFF, temp32:63)
temp0:63 ← EXTS((ACC)32:63) - EXTS((RA)32:63)
ovl ← temp31 ⊕ temp32
RT32:63 ← SATURATE(ovl, temp31, 0x8000_0000,
                   0x7FFF_FFFF, temp32:63)
ACC0:63 ← (RT)0:63
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each signed-integer word element in RA is sign-extended and subtracted from the corresponding sign-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:

ACC OV OVH SOV SOVH

Vector Subtract Unsigned, Modulo, Integer to Accumulator Word EVX-form

evsubfumiaaw RT,RA

4	RT	RA	///	1226
0	6	11	16	21 31

$RT_{0:31} \leftarrow (ACC)_{0:31} - (RA)_{0:31}$
$RT_{32:63} \leftarrow (ACC)_{32:63} - (RA)_{32:63}$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

Each unsigned-integer word element in RA is subtracted from the corresponding element in the accumulator and the results are placed in RT and into the accumulator.

Special Registers Altered:
ACC

Vector Subtract Unsigned, Saturate, Integer to Accumulator Word EVX-form

evsubfusiaaw RT,RA

4	RT	RA	///	1218
0	6	11	16	21 31

$temp_{0:63} \leftarrow EXTZ((ACC)_{0:31}) - EXTZ((RA)_{0:31})$
$ovh \leftarrow temp_{31}$
$RT_{0:31} \leftarrow SATURATE(ovh, temp_{31}, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$
$temp_{0:63} \leftarrow EXTS((ACC)_{32:63}) - EXTS((RA)_{32:63})$
$ovl \leftarrow temp_{31}$
$RT_{32:63} \leftarrow SATURATE(ovl, temp_{31}, 0x0000\_0000, 0x0000\_0000, temp_{32:63})$
$ACC_{0:63} \leftarrow (RT)_{0:63}$

$SPEFSCR_{OVH} \leftarrow ovh$
$SPEFSCR_{OV} \leftarrow ovl$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl$

Each unsigned-integer word element in RA is zero-extended and subtracted from the corresponding zero-extended element in the accumulator saturating if overflow occurs, and the results are placed in RT and the accumulator.

Special Registers Altered:
ACC OV OVH SOV SOVH

Vector Subtract from Word EVX-form

evsubfw RT,RA,RB

4	RT	RA	RB	516
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RB)_{0:31} - (RA)_{0:31}$
$RT_{32:63} \leftarrow (RB)_{32:63} - (RA)_{32:63}$

Each signed-integer element of RA is subtracted from the corresponding element of RB and the results are placed in RT.

Special Registers Altered:
None

Vector Subtract Immediate from Word EVX-form

evsubifw RT,UI,RB

4	RT	UI	RB	518
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RB)_{0:31} - EXTZ(UI)$
$RT_{32:63} \leftarrow (RB)_{32:63} - EXTZ(UI)$

UI is zero-extended and subtracted from both the high and low elements of RB. Note that the same value is subtracted from both elements of the register.

Special Registers Altered:
None

Vector XOR EVX-form

evxor RT,RA,RB

4	RT	RA	RB	534
0	6	11	16	21 31

$RT_{0:31} \leftarrow (RA)_{0:31} \oplus (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \oplus (RB)_{32:63}$

Each element of RA and RB is exclusive-ORed. The results are placed in RT.

Special Registers Altered:
None

Chapter 9. Embedded Floating-Point [Category: SPE.Embedded Float Scalar Double] [Category: SPE.Embedded Float Scalar Single] [Category: SPE.Embedded Float Vector]

9.1 Overview

The *Embedded Floating-Point* categories require the implementation of the Signal Processing Engine (SPE) category and consist of three distinct categories:

- Embedded vector single-precision floating-point (SPE.Embedded Float Vector [SP.FV])
- Embedded scalar single-precision floating-point (SPE.Embedded Float Scalar Single [SP.FS])
- Embedded scalar double-precision floating-point (SPE.Embedded Float Scalar Double [SP.FD])

Although each of these may be implemented independently, they are defined in a single chapter because it is likely that they may be implemented together.

References to *Embedded Floating-Point* categories, *Embedded Floating-Point* instructions, or *Embedded Floating-Point* operations apply to all 3 categories.

Single-precision floating-point is handled by the SPE.Embedded Float Vector and SPE.Embedded Float Scalar Single categories; double-precision floating-point is handled by the SPE.Embedded Float Scalar Double category.

9.2 Programming Model

Embedded floating-point operations are performed in the GPRs of the processor.

The SPE.Embedded Float Vector and SPE.Embedded Float Scalar Double categories require a GPR register file with thirty-two 64-bit registers as required by the Signal Processing Engine category.

The SPE.Embedded Float Scalar Single category requires a GPR register file with thirty-two 32-bit registers. When implemented with a 64-bit register file on a 32-bit implementation, instructions in this category only use and modify bits 32:63 of the GPR. In this case, bits 0:31 of the GPR are left unchanged by the operation. For 64-bit implementations, bits 0:31 are unchanged after the operation.

Instructions in the SPE.Embedded Float Scalar Double category operate on the entire 64 bits of the GPRs.

Instructions in the SPE.Embedded Float Vector category operate on the entire 64 bits of the GPRs as well, but contain two 32-bit data items that are operated on independently of each other in a SIMD fashion. The format of both data items is the same as the format of a data item in the SPE.Embedded Float Scalar Single category. The data item contained in bits 0:31 is called the 'high word'. The data item contained in bits 32:63 is called the 'low word'.

There are no record forms of *Embedded Floating-Point* instructions. *Embedded Floating-Point Compare* instructions treat NaNs, Infinity, and Denorm as normalized numbers for the comparison calculation when default results are provided.

9.2.1 Signal Processing Embedded Floating-Point Status and Control Register (SPEFSCR)

Status and control for the *Embedded Floating-Point* categories uses the SPEFSCR. This register is defined by the Signal Processing Engine category in Section 8.3.4. Status and control bits are shared for *Embedded Floating-Point* and SPE operations. Instructions in the SPE.Embedded Float Vector category affect both the high element (bits 34:39) and low element floating-point status flags (bits 50:55). Instructions in the SPE.Embedded Float Scalar Double and SPE.Embedded Float Scalar Single categories affect only the low element floating-point status flags and leave the high element floating-point status flags undefined.

9.2.2 Floating-Point Data Formats

Single-precision floating-point data elements are 32 bits wide with 1 sign bit (*s*), 8 bits of biased exponent (*e*) and 23 bits of fraction (*f*). Double-precision floating-point data elements are 64 bits wide with 1 sign bit (*s*), 11 bits of biased exponent (*e*) and 52 bits of fraction (*f*).

In the IEEE 754 specification, floating-point values are represented in a format consisting of three explicit fields (sign field, biased exponent field, and fraction field) and an implicit hidden bit.

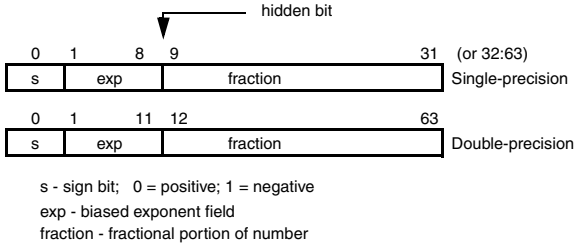


Figure 130.Floating-Point Data Format

For single-precision normalized numbers, the biased exponent value *e* lies in the range of 1 to 254 corresponding to an actual exponent value E in the range -126 to +127. For double-precision normalized numbers, the biased exponent value *e* lies in the range of 1 to 2046 corresponding to an actual exponent value E in the range -1022 to +1023. With the hidden bit implied to be '1' (for normalized numbers), the value of the number is interpreted as follows:

$$(-1)^{s} \times 2^{E} \times (1.\text{fraction})$$

where E is the unbiased exponent and 1.fraction is the mantissa (or significand) consisting of a leading '1' (the hidden bit) and a fractional part (fraction field). For the single-precision format, the maximum positive normalized number (*pmax*) is represented by the encoding `0x7F7FFFFF` which is approximately 3.4E+38 (2^{128}), and the minimum positive normalized value (*pmin*) is represented by the encoding `0x00800000` which is approximately 1.2E-38 (2^{-126}). For the double-precision format, the maximum positive normalized number (*pmax*) is represented by the encoding `0x7feFFFFF_FFFFFFFF` which is approximately 1.8E+307 (2^{1024}), and the minimum positive normalized value (*pmin*) is represented by the encoding `0x00100000_00000000` which is approximately 2.2E-308 (2^{-1022}).

Two specific values of the biased exponent are reserved (0 and 255 for single-precision; 0 and 2047 for double-precision) for encoding special values of +0, -0, +infinity, -infinity, and NaNs.

Zeros of both positive and negative sign are represented by a biased exponent value *e* of 0 and a fraction *f* which is 0.

Infinities of both positive and negative sign are represented by a maximum exponent field value (255 for single-precision, 2047 for double-precision) and a fraction which is 0.

Denormalized numbers of both positive and negative sign are represented by a biased exponent value *e* of 0 and a fraction *f,* which is nonzero. For these numbers, the hidden bit is defined by the IEEE 754 standard to be 0. This number type is not directly supported in hardware. Instead, either a software interrupt handler is invoked, or a default value is defined.

Not-a-Numbers (NaNs) are represented by a maximum exponent field value (255 for single-precision, 2047 for double-precision) and a fraction *f* which is nonzero.

9.2.3 Exception Conditions

9.2.3.1 Denormalized Values on Input

Any denormalized value used as an operand may be truncated by the implementation to a properly signed zero value.

9.2.3.2 Embedded Floating-Point Overflow and Underflow

Defining *pmax* to be the most positive normalized value (farthest from zero), *pmin* the smallest positive normalized value (closest to zero), *nmax* the most negative normalized value (farthest from zero) and *nmin* the smallest normalized negative value (closest to zero), an overflow is said to have occurred if the numerically correct result (*r*) of an instruction is such that *r*>*pmax* or *r*<*nmax*. An underflow is said to have occurred if the numerically correct result of an instruction is such that 0<*r*<*pmin* or *nmin*<*r*<0. In this case, *r* may be denormalized, or may be smaller than the smallest denormalized number.

The *Embedded Floating-Point* categories do not produce +Infinity, -Infinity, NaN, or denormalized numbers. If the result of an instruction overflows and Embedded Floating-Point Overflow exceptions are disabled ($SPEFSCR_{FOVFE}$=0), *pmax* or *nmax* is generated as the result of that instruction depending upon the sign of the result. If the result of an instruction underflows and Embedded Floating-Point Underflow exceptions are disabled ($SPEFSCR_{FUNFE}$=0), +0 or -0 is generated as the result of that instruction based upon the sign of the result.

If an overflow occurs, $SPEFSCR_{FOVF\ FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF\ FUNFH}$ are set appropriately. If either Embedded Floating-Point Underflow or Embedded Floating-Point Overflow exceptions are enabled and a corresponding status bit is 1, an Embedded Floating-Point Data interrupt is taken and the destination register is not updated.

Programming Note

On some implementations, operations that result in overflow or underflow are likely to take significantly longer than operations that do not. For example, these operations may cause a system error handler to be invoked; on such implementations, the system error handler updates the overflow bits appropriately.

9.2.3.3 Embedded Floating-Point Invalid Operation/Input Errors

Embedded Floating-Point Invalid Operation/Input errors occur when an operand to an operation contains an invalid input value. If any of the input values are Infinity, Denorm, or NaN, or for an *Embedded Floating-Point Divide* instruction both operands are +/-0, $SPEFSCR_{FINV\ FINVH}$ are set to 1 appropriately, and $SPEFSCR_{FGH\ FXH\ FG\ FX}$ are set to 0 appropriately. If $SPEFSCR_{FINVE}$=1, an Embedded Floating-Point Data interrupt is taken and the destination register is not updated.

9.2.3.4 Embedded Floating-Point Round (Inexact)

If any result element of an *Embedded Floating-Point* instruction is inexact, or overflows but Embedded Floating-Point Overflow exceptions are disabled, or underflows but Embedded Floating-Point Underflow exceptions are disabled, and no higher priority interrupt occurs, $SPEFSCR_{FINXS}$ is set to 1. If the Embedded Floating-Point Round (Inexact) exception is enabled, an Embedded Floating-Point Round interrupt occurs. In this case, the destination register is updated with the truncated result(s). The $SPEFSCR_{FGH\ FXH\ FG\ FX}$ bits are properly updated to allow rounding to be performed in the interrupt handler.

$SPEFSCR_{FG\ FX}$ ($SPEFSCR_{FGH\ FXH}$) are set to 0 if an Embedded Floating-Point Data interrupt is taken due to overflow, underflow, or if an Embedded Floating-Point Invalid Operation/Input error is signaled for the low (high) element (regardless of $SPEFSCR_{FINVE}$).

9.2.3.5 Embedded Floating-Point Divide by Zero

If an *Embedded Floating-Point Divide* instruction executes and an Embedded Floating-Point Invalid Operation/Input error does not occur and the instruction is executed with a +/-0 divisor value and a finite normalized nonzero dividend value, an Embedded Floating-Point Divide By Zero exception occurs and $SPEFSCR_{FDBZ\ FDBZH}$ are set appropriately. If Embedded Floating-Point Divide By Zero exceptions are enabled, an Embedded Floating-Point Data

interrupt is then taken and the destination register is not updated.

9.2.3.6 Default Results

Default results are generated when an Embedded Floating-Point Invalid Operation/Input Error, Embedded Floating-Point Overflow, Embedded Floating-Point Underflow, or Embedded Floating-Point Divide by Zero occurs on an *Embedded Floating-Point* operation. Default results provide a normalized value as a result of the operation. In general, Denorm results and underflows are set to 0 and overflows are saturated to the maximum representable number.

Default results produced for each operation are described in Section 9.4, “Embedded Floating-Point Results Summary”.

9.2.4 IEEE 754 Compliance

The *Embedded Floating-Point* categories require a floating-point system as defined in the ANSI/IEEE Standard 754-1985 but may rely on software support in order to conform fully with the standard. Thus, whenever an input operand of the *Embedded Floating-Point* instruction has data values that are +Infinity, -Infinity, Denormalized, NaN, or when the result of an operation produces an overflow or an underflow, an Embedded Floating-Point Data interrupt may be taken and the interrupt handler is responsible for delivering IEEE 754 compliant behavior if desired.

When Embedded Floating-Point Invalid Operation/Input Error exceptions are disabled ($SPEFSCR_{FINVE}$ = 0), default results are provided by the hardware when an Infinity, Denormalized, or NaN input is received, or for the operation 0/0. When Embedded Floating-Point Underflow exceptions are disabled ($SPEFSCR_{FUNFE}$ = 0) and the result of a floating-point operation underflows, a signed zero result is produced. The Embedded Floating-Point Round (Inexact) exception is also signaled for this condition. When Embedded Floating-Point Overflow exceptions are disabled ($SPEFSCR_{FOVFE}$ = 0) and the result of a floating-point operation overflows, a *pmax* or *nmax* result is produced. The Embedded Floating-Point Round (Inexact) exception is also signaled for this condition. An exception enable flag ($SPEFSCR_{FINXE}$) is also provided for generating an Embedded Floating-Point Round interrupt when an inexact result is produced, to allow a software handler to conform to the IEEE 754 standard. An Embedded Floating-Point Divide By Zero exception enable flag ($SPEFSCR_{FDBZE}$) is provided for generating an Embedded Floating-Point Data interrupt when a divide by zero operation is attempted to allow a software handler to conform to the IEEE 754 standard. All of these exceptions may be disabled, and the hardware will then deliver an appropriate default result.

The sign of the result of an addition operation is the sign of the source operand having the larger absolute value. If both operands have the same sign, the sign of the result is the same as the sign of the operands. This includes subtraction which is addition with the negation of the sign of the second operand. The sign of the result of an addition operation with operands of differing signs for which the result is zero is positive except when rounding to negative infinity. Thus -0 + -0 = -0, and all other cases which result in a zero value give +0 unless the rounding mode is round to negative infinity.

Programming Note

Note that when exceptions are disabled and default results computed, operations having input values that are denormalized may provide different results on different implementations. An implementation may choose to use the denormalized value or a zero value for any computation. Thus a computational operation involving a denormalized value and a normal value may return different results depending on the implementation.

9.2.4.1 Sticky Bit Handling For Exception Conditions

The SPEFSCR register defines sticky bits for retaining information about exception conditions that are detected. There are 5 sticky bits (FINXS, FINVS, FDBZS, FUNFS and FOVFS) that can be used to help provide IEEE 754 compliance. The sticky bits represent the combined ‘or’ of all the previous status bits produced from any *Embedded Floating-Point* operation since the last time software zeroed the sticky bit. The hardware will never set a sticky bit to 0.

9.3 Embedded Floating-Point Instructions

9.3.1 Load/Store Instructions

Embedded Floating-Point instructions use GPRs to hold and operate on floating-point values. The *Embedded Floating-Point* categories do not define *Load* and *Store* instructions to move the data to and from memory, but instead rely on existing instructions in Book I to load and store data.

.

9.3.2 SPE.Embedded Float Vector Instructions [Category: SPE.Embedded Float Vector]

All SPE.Embedded Float Vector instructions are single-precision. There are no vector floating-point double-precision instructions

Vector Floating-Point Single-Precision Absolute Value EVX-form

evfsabs RT,RA

4	RT	RA	///	644
0	6	11	16	21 31

$RT_{0:31} \leftarrow 0b0 \ || \ (RA)_{1:31}$
$RT_{32:63} \leftarrow 0b0 \ || \ (RA)_{33:63}$

The sign bit of each element in register RA is set to 0 and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Vector Floating-Point Single-Precision Negative Absolute Value EVX-form

evfsnabs RT,RA

4	RT	RA	///	645
0	6	11	16	21 31

$RT_{0:31} \leftarrow 0b1 \ || \ (RA)_{1:31}$
$RT_{32:63} \leftarrow 0b1 \ || \ (RA)_{33:63}$

The sign bit of each element in register RA is set to 1 and the results are placed into register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Vector Floating-Point Single-Precision Negate EVX-form

evfsneg RT,RA

4	RT	RA	///	646
0	6	11	16	21 31

$RT_{0:31} \leftarrow \neg(RA)_{0} \ || \ (RA)_{1:31}$
$RT_{32:63} \leftarrow \neg(RA)_{32} \ || \ (RA)_{33:63}$

The sign bit of each element in register RA is complemented and the results are placed into register RT.

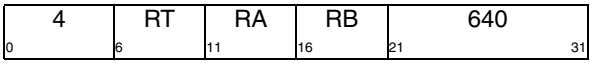
Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Vector Floating-Point Single-Precision Add *EVX-form*

evfsadd RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31} +_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} +_{sp} (RB)_{32:63}$

Each single-precision floating-point element of register RA is added to the corresponding element of register RB and the results are stored in register RT.

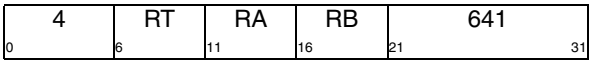
If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of register RT.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Subtract *EVX-form*

evfssub RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31} -_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} -_{sp} (RB)_{32:63}$

Each single-precision floating-point element of register RB is subtracted from the corresponding element of register RA and the results are stored in register RT.

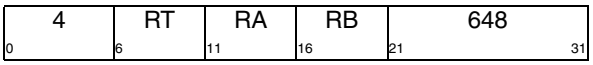
If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of register RT.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Multiply *EVX-form*

evfsmul RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31} \times_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \times_{sp} (RB)_{32:63}$

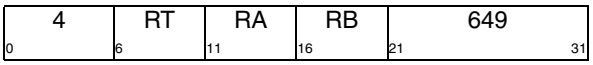
Each single-precision floating-point element of register RA is multiplied with the corresponding element of register RB and the result is stored in register RT.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Divide *EVX-form*

evfsdiv RT,RA,RB



$RT_{0:31} \leftarrow (RA)_{0:31} \div_{sp} (RB)_{0:31}$
$RT_{32:63} \leftarrow (RA)_{32:63} \div_{sp} (RB)_{32:63}$

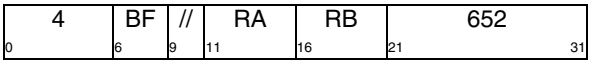
Each single-precision floating-point element of register RA is divided by the corresponding element of register RB and the result is stored in register RT.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS
FDBZ FDBZH FDBZS
FOVF FOVFH FOVFS
FUNF FUNFH FUNFS

Vector Floating-Point Single-Precision Compare Greater Than EVX-form

evfscmpgt BF,RA,RB




```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is greater than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

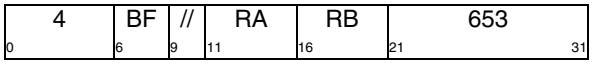
If an input error occurs and default results are generated, NaNs, Infinities, and Denorms as treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX
CR field BF

Vector Floating-Point Single-Precision Compare Less Than EVX-form

evfscmplt BF,RA,RB




```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is less than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

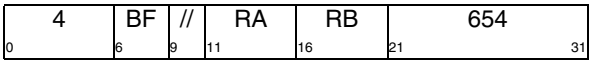
If an input error occurs and default results are generated, NaNs, Infinities, and Denorms as treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX
CR field BF

Vector Floating-Point Single-Precision Compare Equal *EVX-form*

evfscmpeq BF,RA,RB




```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is equal to $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0).

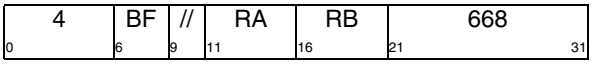
If an input error occurs and default results are generated, NaNs, Infinities, and Denorms as treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX
CR field BF

Vector Floating-Point Single-Precision Test Greater Than *EVX-form*

evfststgt BF,RA,RB




```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah > bh) then ch ← 1
else ch ← 0
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB.The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is greater than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are taken during the execution of ***evfststgt***.

Special Registers Altered:

CR field BF

Programming Note

In an implementation, the execution of ***evfststgt*** is likely to be faster than the execution of ***evfscmpgt***; however, if strict IEEE 754 compliance is required, the program should use ***evfscmpgt***.

Vector Floating-Point Single-Precision Test Less Than *EVX-form*

evfststlt BF,RA,RB

4	BF	//	RA	RB	669
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah < bh) then ch ← 1
else ch ← 0
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared with the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is less than $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are taken during the execution of ***evfststlt***.

Special Registers Altered:

CR field BF

Programming Note

In an implementation, the execution of ***evfststlt*** is likely to be faster than the execution of ***evfscmplt***; however, if strict IEEE 754 compliance is required, the program should use ***evfscmplt***.

Vector Floating-Point Single-Precision Test Equal *EVX-form*

evfststeq BF,RA,RB

4	BF	//	RA	RB	670
0	6	9	11	16	21 31

```
ah ← (RA)0:31
al ← (RA)32:63
bh ← (RB)0:31
bl ← (RB)32:63
if (ah = bh) then ch ← 1
else ch ← 0
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← ch || cl || (ch | cl) || (ch & cl)
```

Each element of register RA is compared against the corresponding element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{0:31}$ is equal to $RB_{0:31}$, bit 0 of CR field BF is set to 1, otherwise it is set to 0. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bit 2 of CR field BF is set to the OR of both result bits and Bit 3 of CR field BF is set to the AND of both result bits. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are taken during the execution of ***evfststeq***.

Special Registers Altered:

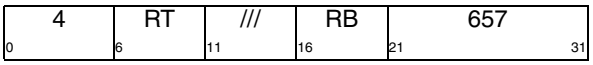
CR field BF

Programming Note

In an implementation, the execution of ***evfststeq*** is likely to be faster than the execution of ***evfscmpeq***; however, if strict IEEE 754 compliance is required, the program should use ***evfscmpeq***.

Vector Convert Floating-Point Single-Precision from Signed Integer *EVX-form*

evfscfsi RT,RB



$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, S, HI, I)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, S, LO, I)

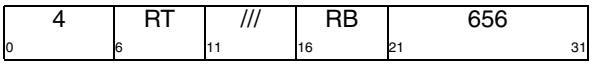
Each signed integer element of register RB is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding element of register RT.

Special Registers Altered:

FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision from Unsigned Integer *EVX-form*

evfscfui RT,RB



$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, U, HI, I)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, U, LO, I)

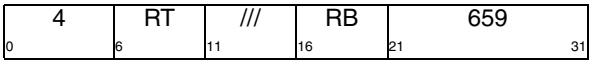
Each unsigned integer element of register RB is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

Special Registers Altered:

FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision from Signed Fraction *EVX-form*

evfscfsf RT,RB



$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, S, HI, F)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, S, LO, F)

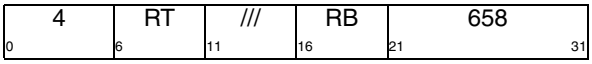
Each signed fractional element of register RB is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

Special Registers Altered:

FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision from Unsigned Fraction *EVX-form*

evfscfuf RT,RB



$RT_{0:31} \leftarrow$ CnvtI32ToFP32((RB)$_{0:31}$, U, HI, F)
$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, U, LO, F)

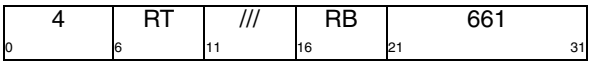
Each unsigned fractional element of register RB is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of register RT.

Special Registers Altered:

FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Signed Integer *EVX-form*

evfsctsi RT,RB



$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, S, HI, RND, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, RND, I)

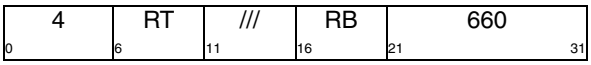
Each single-precision floating-point element in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Unsigned Integer *EVX-form*

evfsctui RT,RB



$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, U, HI, RND, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$,U, LO, RND, I)

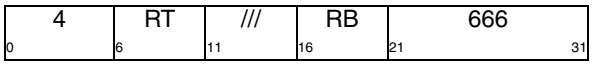
Each single-precision floating-point element in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero *EVX-form*

evfsctsiz RT,RB



$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, S, HI, ZER, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, ZER, I)

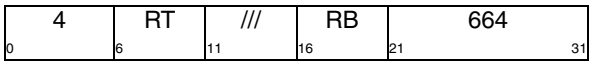
Each single-precision floating-point element in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero *EVX-form*

evfsctuiz RT,RB



$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, U, HI, ZER, I)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, U, LO, ZER, I)

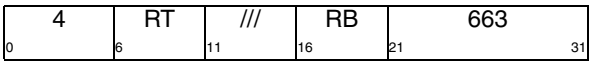
Each single-precision floating-point element in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Signed Fraction *EVX-form*

evfsctsf RT,RB



$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, S, HI, RND ,F)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, S, LO, RND, F)

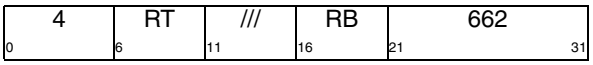
Each single-precision floating-point element in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit signed fraction. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS

Vector Convert Floating-Point Single-Precision to Unsigned Fraction *EVX-form*

evfsctuf RT,RB



$RT_{0:31}$ ← CnvtFP32ToI32Sat($(RB)_{0:31}$, U, HI, RND, F)
$RT_{32:63}$ ← CnvtFP32ToI32Sat($(RB)_{32:63}$, U, LO, RND, F)

Each single-precision floating-point element in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVH FINVS
FGH FXH FG FX FINXS

9.3.3 SPE.Embedded Float Scalar Single Instructions [Category: SPE.Embedded Float Scalar Single]

Floating-Point Single-Precision Absolute Value *EVX-form*

efsabs RT,RA

4	RT	RA	///	708
0	6	11	16	21 31

$RT_{32:63} \leftarrow 0b0 \ || \ (RA)_{33:63}$

The sign bit of the low element of register RA is set to 0 and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Floating-Point Single-Precision Negate *EVX-form*

efsneg RT,RA

4	RT	RA	///	710
0	6	11	16	21 31

$RT_{32:63} \leftarrow \neg(RA)_{32} \ || \ (RA)_{33:63}$

The sign bit of the low element of register RA is complemented and the result is placed into the low element of register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Floating-Point Single-Precision Negative Absolute Value *EVX-form*

efsnabs RT,RA

4	RT	RA	///	709
0	6	11	16	21 31

$RT_{32:63} \leftarrow 0b1 \ || \ (RA)_{33:63}$

The sign bit of the low element of register RA is set to 1 and the result is placed into the low element of register RT.

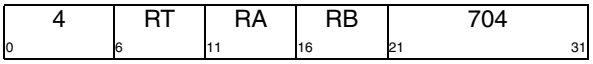
Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Floating-Point Single-Precision Add EVX-form

efsadd RT,RA,RB



$RT_{32:63} \leftarrow (RA)_{32:63} +_{sp} (RB)_{32:63}$

The low element of register RA is added to the low element of register RB and the result is stored in the low element of register RT.

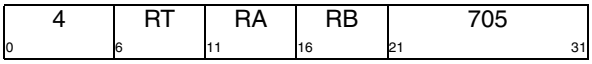
If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Single-Precision Subtract EVX-form

efssub RT,RA,RB



$RT_{32:63} \leftarrow (RA)_{32:63} -_{sp} (RB)_{32:63}$

The low element of register RB is subtracted from the low element of register RA and the result is stored in the low element of register RT.

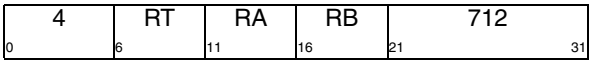
If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Single-Precision Multiply EVX-form

efsmul RT,RA,RB



$RT_{32:63} \leftarrow (RA)_{32:63} \times_{sp} (RB)_{32:63}$

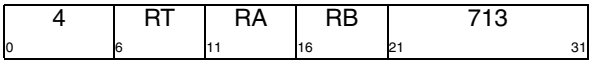
The low element of register RA is multiplied by the low element of register RB and the result is stored in the low element of register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Single-Precision Divide EVX-form

efsdiv RT,RA,RB



$RT_{32:63} \leftarrow (RA)_{32:63} \div_{sp} (RB)_{32:63}$

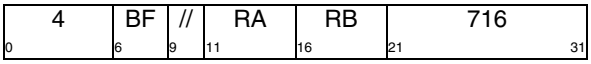
The low element of register RA is divided by the low element of register RB and the result is stored in the low element of register RT.

Special Registers Altered:

FINV FINVS
FG FX FINXS
FDBZ FDBZS
FOVF FOVFS
FUNF FUNFS

Floating-Point Single-Precision Compare Greater Than EVX-form

efscmpgt BF,RA,RB




```
al ← (RA)32:63
bl ← (RB)32:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. The results of the comparisons are placed into CR field BF. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

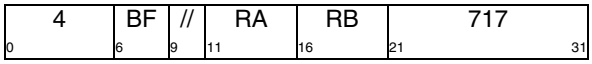
If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Single-Precision Compare Less Than EVX-form

efscmplt BF,RA,RB




```
al ← (RA)32:63
bl ← (RB)32:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

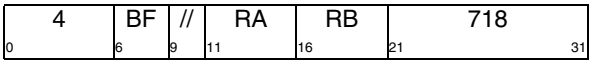
If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Single-Precision Compare Equal *EVX-form*

efscmpeq BF,RA,RB




```
al ← (RA)32:63
bl ← (RB)32:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

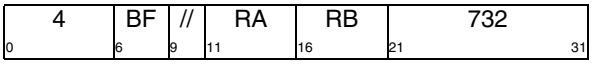
If an Input Error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Single-Precision Test Greater Than *EVX-form*

efststgt BF,RA,RB




```
al ← (RA)32:63
bl ← (RB)32:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is greater than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efststgt***.

Special Registers Altered:

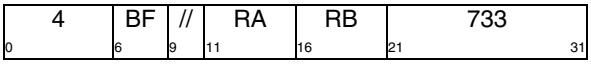
CR field BF

Programming Note

In an implementation, the execution of ***efststgt*** is likely to be faster than the execution of ***efscmpgt***; however, if strict IEEE 754 compliance is required, the program should use ***efscmpgt***.

Floating-Point Single-Precision Test Less Than EVX-form

efststlt BF,RA,RB




```
al ← (RA)32:63
bl ← (RB)32:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is less than $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efststlt***.

Special Registers Altered:

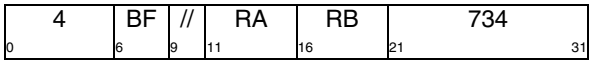
CR field BF

Programming Note

In an implementation, the execution of ***efststlt*** is likely to be faster than the execution of ***efscmplt***; however, if strict IEEE 754 compliance is required, the program should use ***efscmplt***.

Floating-Point Single-Precision Test Equal EVX-form

efststeq BF,RA,RB




```
al ← (RA)32:63
bl ← (RB)32:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

The low element of register RA is compared against the low element of register RB. If $RA_{32:63}$ is equal to $RB_{32:63}$, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efststeq***.

Special Registers Altered:

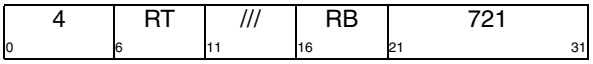
CR field BF

Programming Note

In an implementation, the execution of ***efststeq*** is likely to be faster than the execution of ***efscmpeq***; however, if strict IEEE 754 compliance is required, the program should use ***efscmpeq***.

Convert Floating-Point Single-Precision from Signed Integer EVX-form

efscfsi RT,RB



$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, S, LO, I)

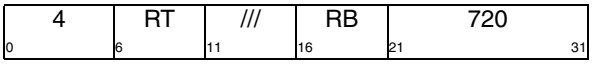
The signed integer low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:

FINXS FG FX

Convert Floating-Point Single-Precision from Unsigned Integer EVX-form

efscfui RT,RB



$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, U, LO, I)

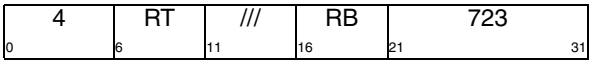
The unsigned integer low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:

FINXS FG FX

Convert Floating-Point Single-Precision from Signed Fraction EVX-form

efscfsf RT,RB



$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, S, LO, F)

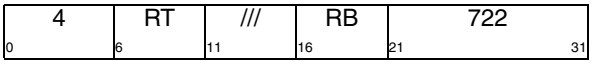
The signed fractional low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:

FINXS FG FX

Convert Floating-Point Single-Precision from Unsigned Fraction EVX-form

efscfuf RT,RB



$RT_{32:63} \leftarrow$ CnvtI32ToFP32((RB)$_{32:63}$, U, LO, F)

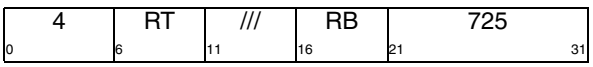
The unsigned fractional low element in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Special Registers Altered:

FINXS FG FX

Convert Floating-Point Single-Precision to Signed Integer EVX-form

efsctsi RT,RB



$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat((RB)$_{32:63}$, S, LO, RND, I)

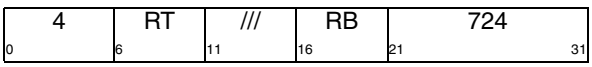
The single-precision floating-point low element in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Unsigned Integer EVX-form

efsctui RT,RB



$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat((RB)$_{32:63}$, U, LO, RND, I)

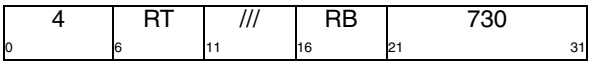
The single-precision floating-point low element in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero *EVX-form*

efsctsiz RT,RB



$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat((RB)$_{32:63}$, S, LO, ZER, I)

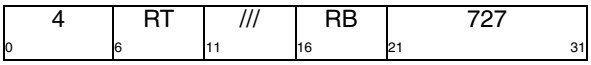
The single-precision floating-point low element in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Signed Fraction *EVX-form*

efsctsf RT,RB



$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat((RB)$_{32:63}$, S, LO, RND, F)

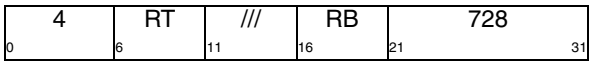
The single-precision floating-point low element in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero *EVX-form*

efsctuiz RT,RB



$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat((RB)$_{32:63}$, U, LO, ZER, I)

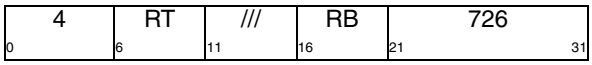
The single-precision floating-point low element in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Single-Precision to Unsigned Fraction *EVX-form*

efsctuf RT,RB



$RT_{32:63} \leftarrow$ CnvtFP32ToI32Sat((RB)$_{32:63}$, U, LO, RND, F)

The single-precision floating-point low element in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

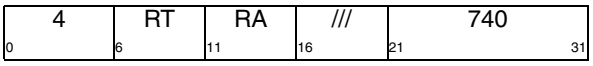
Special Registers Altered:

FINV FINVS
FINXS FG FX

9.3.4 SPE.Embedded Float Scalar Double Instructions [Category: SPE.Embedded Float Scalar Double]

Floating-Point Double-Precision Absolute Value *EVX-form*

efdabs RT,RA



$RT_{0:63} \leftarrow 0b0 \ || \ (RA)_{1:63}$

The sign bit of register RA is set to 0 and the result is placed in register RT.

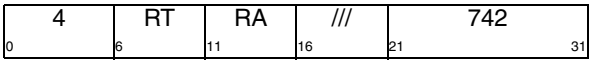
Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Floating-Point Double-Precision Negate *EVX-form*

efdneg RT,RA



$RT_{0:63} \leftarrow \neg(RA)_{0} \ || \ (RA)_{1:63}$

The sign bit of register RA is complemented and the result is placed in register RT.

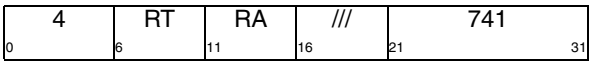
Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Floating-Point Double-Precision Negative Absolute Value *EVX-form*

efdnabs RT,RA



$RT_{0:63} \leftarrow 0b1 \ || \ (RA)_{1:63}$

The sign bit of register RA is set to 1 and the result is placed in register RT.

Regardless of the value of register RA, no exceptions are taken during the execution of this instruction.

Special Registers Altered:

None

Floating-Point Double-Precision Add EVX-form

efdadd RT,RA,RB

4	RT	RA	RB	736
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} +_{dp} (RB)_{0:63}$

RA is added to RB and the result is stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Subtract EVX-form

efdsub RT,RA,RB

4	RT	RA	RB	737
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} -_{dp} (RB)_{0:63}$

RB is subtracted from RA and the result is stored in register RT.

If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Multiply EVX-form

efdmul RT,RA,RB

4	RT	RA	RB	744
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} \times_{dp} (RB)_{0:63}$

RA is multiplied by RB and the result is stored in register RT.

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

Floating-Point Double-Precision Divide EVX-form

efddiv RT,RA,RB

4	RT	RA	RB	745
0	6	11	16	21 31

$RT_{0:63} \leftarrow (RA)_{0:63} \div_{dp} (RB)_{0:63}$

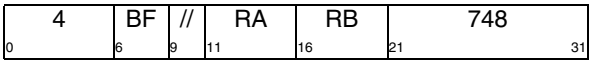
RA is divided by RB and the result is stored in register RT.

Special Registers Altered:

FINV FINVS
FG FX FINXS
FDBZ FDBZS
FOVF FOVFS
FUNF FUNFS

Floating-Point Double-Precision Compare Greater Than EVX-form

efdcmpgt BF,RA,RB




```
al ← (RA)0:63
bl ← (RB)0:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is greater than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

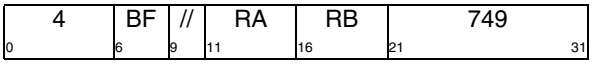
If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Double-Precision Compare Less Than EVX-form

efdcmplt BF,RA,RB




```
al ← (RA)0:63
bl ← (RB)0:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is less than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

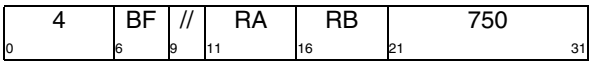
If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Double-Precision Compare Equal EVX-form

efdcmpeq BF,RA,RB




```
al ← (RA)0:63
bl ← (RB)0:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is equal to RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0).

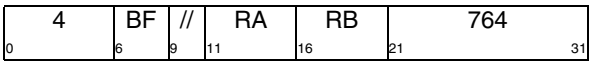
If an input error occurs and default results are generated, NaNs, Infinities, and Denorms are treated as normalized numbers, using their values of '*e*' and '*f*' directly.

Special Registers Altered:

FINV FINVS
FG FX
CR field BF

Floating-Point Double-Precision Test Greater Than EVX-form

efdtstgt BF,RA,RB




```
al ← (RA)0:63
bl ← (RB)0:63
if (al > bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is greater than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efdtstgt***.

Special Registers Altered:

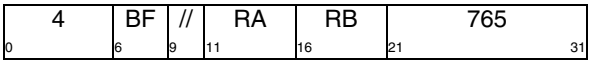
CR field BF

Programming Note

In an implementation, the execution of ***efdtstgt*** is likely to be faster than the execution of ***efdcmpgt***; however, if strict IEEE 754 compliance is required, the program should use ***efdcmpgt***.

Floating-Point Double-Precision Test Less Than EVX-form

efdtstlt BF,RA,RB




```
al ← (RA)0:63
bl ← (RB)0:63
if (al < bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is less than RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efdtstlt***.

Special Registers Altered:

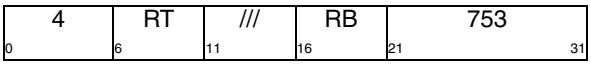
CR field BF

Programming Note

In an implementation, the execution of ***efdtstlt*** is likely to be faster than the execution of ***efdcmplt***; however, if strict IEEE 754 compliance is required, the program should use ***efdcmplt***.

Convert Floating-Point Double-Precision from Signed Integer EVX-form

efdcfsi RT,RB




```
RT0:63 ← CnvtI32ToFP64((RB)32:63, S, I)
```

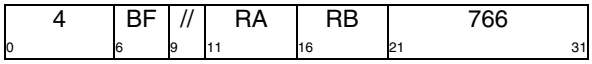
The signed integer low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:

None

Floating-Point Double-Precision Test Equal EVX-form

efdtsteq BF,RA,RB




```
al ← (RA)0:63
bl ← (RB)0:63
if (al = bl) then cl ← 1
else cl ← 0
CR4×BF:4×BF+3 ← undefined || cl || undefined || undefined
```

RA is compared against RB. If RA is equal to RB, bit 1 of CR field BF is set to 1, otherwise it is set to 0. Bits 0, 2, and 3 of CR field BF are undefined. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of ***efdtsteq***.

Special Registers Altered:

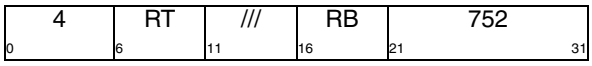
CR field BF

Programming Note

In an implementation, the execution of ***efdtsteq*** is likely to be faster than the execution of ***efdcmpeq***; however, if strict IEEE 754 compliance is required, the program should use ***efdcmpeq***.

Convert Floating-Point Double-Precision from Unsigned Integer EVX-form

efdcfui RT,RB




```
RT0:63 ← CnvtI32ToFP64((RB)32:63, U, I)
```

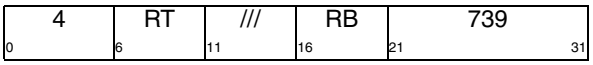
The unsigned integer low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:

None

Convert Floating-Point Double-Precision from Signed Integer Doubleword *EVX-form*

efdcfsid RT,RB



$RT_{0:63} \leftarrow$ CnvtI64ToFP64((RB)$_{0:63}$, S)

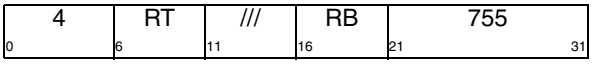
The signed integer doubleword in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Corequisite Categories:
64-Bit

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Double-Precision from Signed Fraction *EVX-form*

efdcfsf RT,RB



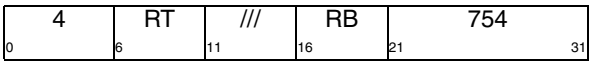
$RT_{0:63} \leftarrow$ CnvtI32ToFP64((RB)$_{32:63}$, S, F)

The signed fractional low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:
None

Convert Floating-Point Double-Precision from Unsigned Fraction *EVX-form*

efdcfuf RT,RB



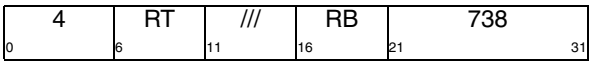
$RT_{0:63} \leftarrow$ CnvtI32ToFP64((RB)$_{32:63}$, U, F)

The unsigned fractional low element in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Special Registers Altered:
None

Convert Floating-Point Double-Precision from Unsigned Integer Doubleword *EVX-form*

efdcfuid RT,RB



$RT_{0:63} \leftarrow$ CnvtI64ToFP64((RB)$_{0:63}$, U)

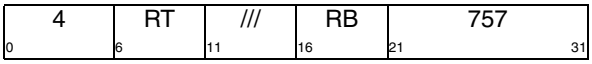
The unsigned integer doubleword in register RB is converted to a double-precision floating-point value using the current rounding mode and the result is placed in register RT.

Corequisite Categories:
64-Bit

Special Registers Altered:
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Integer *EVX-form*

efdctsi RT,RB



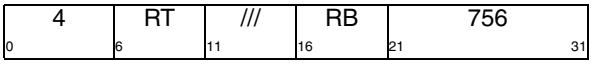
$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat((RB)$_{0:63}$, S, RND, I)

The double-precision floating-point value in register RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Integer *EVX-form*

efdctui RT,RB



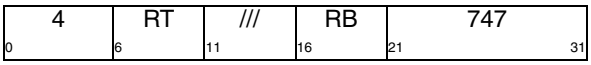
$RT_{32:63} \leftarrow$ CnvtFP64ToI32Sat((RB)$_{0:63}$, U, RND, I)

The double-precision floating-point value in register RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:
FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero EVX-form

efdctsidz RT,RB



$RT_{0:63}$ ← CnvtFP64ToI64Sat($(RB)_{0:63}$, S, ZER)

The double-precision floating-point value in register RB is converted to a signed integer doubleword using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 64-bit integer. NaNs are converted as though they were zero.

Corequisite Categories:

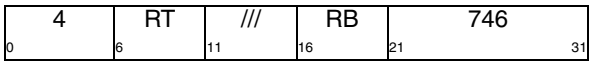
64-Bit

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero EVX-form

efdctuidz RT,RB



$RT_{0:63}$ ← CnvtFP64ToI64Sat($(RB)_{0:63}$, U, ZER)

The double-precision floating-point value in register RB is converted to an unsigned integer doubleword using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 64-bit integer. NaNs are converted as though they were zero.

Corequisite Categories:

64-Bit

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero *EVX-form*

efdctsiz RT,RB

4	RT	///	RB	762
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP64ToI32Sat($(RB)_{0:63}$, S, ZER, I)

The double-precision floating-point value in register RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Signed Fraction *EVX-form*

efdctsf RT,RB

4	RT	///	RB	759
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP64ToI32Sat($(RB)_{0:63}$, S, RND, F)

The double-precision floating-point value in register RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Fraction *EVX-form*

efdctuf RT,RB

4	RT	///	RB	758
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP64ToI32Sat($(RB)_{0:63}$, U, RND, F)

The double-precision floating-point value in register RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero *EVX-form*

efdctuiz RT,RB

4	RT	///	RB	760
0	6	11	16	21 31

$RT_{32:63}$ ← CnvtFP64ToI32Sat($(RB)_{0:63}$, U, ZER, I)

The double-precision floating-point value in register RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Special Registers Altered:

FINV FINVS
FINXS FG FX

Floating-Point Double-Precision Convert from Single-Precision *EVX-form*

efdcfs RT,RB

4	RT	///	RB	751
0	6	11	16	21 31

```
FP32format f;
FP64format result;
f ← (RB)32:63
if (fexp = 0) & (ffrac = 0)) then
    result ← fsign || 630
else if Isa32NaNorInfinity(f) | Isa32Denorm(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 0b11111111110 || 521
else if Isa32Denorm(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 630
else
    resultsign ← fsign
    resultexp ← fexp - 127 + 1023
    resultfrac ← ffrac || 290
RT0:63 ← result
```

The single-precision floating-point value in the low element of register RB is converted to a double-precision floating-point value and the result is placed in register RT.

Corequisite Categories:

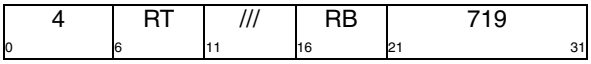
SPE.Embedded Float Scalar Single or
SPE.Embedded Float Vector

Special Registers Altered:

FINV FINVS
FG FX

Floating-Point Single-Precision Convert from Double-Precision EVX-form

efscfd RT,RB




```
FP64format f;
FP32format result;
f ← (RB)0:63
if (fexp = 0) & (ffrac = 0)) then
    result ← fsign || 310
else if Isa64NaNorInfinity(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 0b11111110 || 231
else if Isa64Denorm(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 310
else
    unbias ← fexp - 1023
    if unbias > 127 then
        result ← fsign || 0b11111110 || 231
        SPEFSCRFOVF ← 1
    else if unbias < -126 then
        result ← fsign || 310
        SPEFSCRFUNF ← 1
    else
        resultsign ← fsign
        resultexp ← unbias + 127
        resultfrac ← ffrac[0:22]
        guard ← ffrac[23]
        sticky ← (ffrac[24:51] ≠ 0)
        result ← Round32(result, LO, guard,
sticky)
        SPEFSCRFG ← guard
        SPEFSCRFX ← sticky
        if guard | sticky then
            SPEFSCRFINXS ← 1
RT32:63 ← result
```

The double-precision floating-point value in register RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of register RT.

Corequisite Categories:

SPE.Embedded Float Scalar Scalar

Special Registers Altered:

FINV FINVS
FOVF FOVFS
FUNF FUNFS
FG FX FINXS

9.4 Embedded Floating-Point Results Summary

The following tables summarize the results of various types of *Embedded Floating-Point* operations on various combinations of input operands. Flag settings are performed on appropriate element flags. For all the tables the following annotation and general rules apply:

- * denotes that this status flag is set based on the results of the calculation.
- *_Calc_* denotes that the result is updated with the results of the computation.
- *max* denotes the maximum normalized number with the sign set to the computation [sign(operand A) XOR sign(operand B)].
- *amax* denotes the maximum normalized number with the sign set to the sign of Operand A.
- *bmax* denotes the maximum normalized number with the sign set to the sign of Operand B.
- *pmax* denotes the maximum normalized positive number. The encoding for single-precision is: 0x7F7FFFFF. The encoding for double-precision is: 0x7FEFFFFF_FFFFFFFF.
- *nmax* denotes the maximum normalized negative number. The encoding for single-precision is: 0xFF7FFFFF. The encoding for double-precision is: 0xFFEFFFFF_FFFFFFFF.
- *pmin* denotes the minimum normalized positive number. The encoding for single-precision is: 0x00800000. The encoding for double-precision is: 0x00100000_00000000.
- *nmin* denotes the minimum normalized negative number. The encoding for single-precision is: 0x80800000. The encoding for double-precision is: 0x80100000_00000000.
- Calculations that overflow or underflow saturate. Overflow for operations that have a floating-point result force the result to *max*. Underflow for operations that have a floating-point result force the result to zero. Overflow for operations that have a signed integer result force the result to 0x7FFFFFFF (positive) or 0x80000000 (negative). Overflow for operations that have an unsigned integer result force the result to 0xFFFFFFFF (positive) or 0x00000000 (negative).
- [1] (superscript) denotes that the sign of the result is positive when the sign of Operand A and the sign of Operand B are different, for all rounding modes except round to -infinity, where the sign of the result is then negative.
- [2] (superscript) denotes that the sign of the result is positive when the sign of Operand A and the sign of Operand B are the same, for all rounding modes except round to -infinity, where the sign of the result is then negative.
- [3] (superscript) denotes that the sign for any multiply or divide is always the result of the operation [sign(Operand A) XOR sign(Operand B)].
- [4] (superscript) denotes that if an overflow is detected, the result may be saturated.

Table 115:Embedded Floating-Point Results Summary—Add, Sub, Mul, Div

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Add								
Add	∞	∞	amax	1	0	0	0	0
Add	∞	NaN	amax	1	0	0	0	0
Add	∞	denorm	amax	1	0	0	0	0
Add	∞	zero	amax	1	0	0	0	0
Add	∞	Norm	amax	1	0	0	0	0
Add	NaN	∞	amax	1	0	0	0	0
Add	NaN	NaN	amax	1	0	0	0	0
Add	NaN	denorm	amax	1	0	0	0	0
Add	NaN	zero	amax	1	0	0	0	0
Add	NaN	norm	amax	1	0	0	0	0
Add	denorm	∞	bmax	1	0	0	0	0
Add	denorm	NaN	bmax	1	0	0	0	0
Add	denorm	denorm	zero[1]	1	0	0	0	0
Add	denorm	zero	zero[1]	1	0	0	0	0
Add	denorm	norm	operand_b[4]	1	0	0	0	0
Add	zero	∞	bmax	1	0	0	0	0
Add	zero	NaN	bmax	1	0	0	0	0
Add	zero	denorm	zero[1]	1	0	0	0	0

Table 115:Embedded Floating-Point Results Summary—Add, Sub, Mul, Div (Continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Add	zero	zero	zero[1]	0	0	0	0	0
Add	zero	norm	operand_b[4]	0	0	0	0	0
Add	norm	∞	bmax	1	0	0	0	0
Add	norm	NaN	bmax	1	0	0	0	0
Add	norm	denorm	operand_a[4]	1	0	0	0	0
Add	norm	zero	operand_a[4]	0	0	0	0	0
Add	norm	norm	_Calc_	0	*	*	0	*
Subtract								
Sub	∞	∞	amax	1	0	0	0	0
Sub	∞	NaN	amax	1	0	0	0	0
Sub	∞	denorm	amax	1	0	0	0	0
Sub	∞	zero	amax	1	0	0	0	0
Sub	∞	Norm	amax	1	0	0	0	0
Sub	NaN	∞	amax	1	0	0	0	0
Sub	NaN	NaN	amax	1	0	0	0	0
Sub	NaN	denorm	amax	1	0	0	0	0
Sub	NaN	zero	amax	1	0	0	0	0
Sub	NaN	norm	amax	1	0	0	0	0
Sub	denorm	∞	-bmax	1	0	0	0	0
Sub	denorm	NaN	-bmax	1	0	0	0	0
Sub	denorm	denorm	zero[2]	1	0	0	0	0
Sub	denorm	zero	zero[2]	1	0	0	0	0
Sub	denorm	norm	-operand_b[4]	1	0	0	0	0
Sub	zero	∞	-bmax	1	0	0	0	0
Sub	zero	NaN	-bmax	1	0	0	0	0
Sub	zero	denorm	zero[2]	1	0	0	0	0
Sub	zero	zero	zero[2]	0	0	0	0	0
Sub	zero	norm	-operand_b[4]	0	0	0	0	0
Sub	norm	∞	-bmax	1	0	0	0	0
Sub	norm	NaN	-bmax	1	0	0	0	0
Sub	norm	denorm	operand_a[4]	1	0	0	0	0
Sub	norm	zero	operand_a[4]	0	0	0	0	0
Sub	norm	norm	_Calc_	0	*	*	0	*
Multiply[3]								
Mul	∞	∞	max	1	0	0	0	0
Mul	∞	NaN	max	1	0	0	0	0
Mul	∞	denorm	zero	1	0	0	0	0
Mul	∞	zero	zero	1	0	0	0	0
Mul	∞	Norm	max	1	0	0	0	0
Mul	NaN	∞	max	1	0	0	0	0
Mul	NaN	NaN	max	1	0	0	0	0
Mul	NaN	denorm	zero	1	0	0	0	0
Mul	NaN	zero	zero	1	0	0	0	0
Mul	NaN	norm	max	1	0	0	0	0

Table 115:Embedded Floating-Point Results Summary—Add, Sub, Mul, Div (Continued)								
Operation	**Operand A**	**Operand B**	**Result**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
Mul	denorm	∞	zero	1	0	0	0	0
Mul	denorm	NaN	zero	1	0	0	0	0
Mul	denorm	denorm	zero	1	0	0	0	0
Mul	denorm	zero	zero	1	0	0	0	0
Mul	denorm	norm	zero	1	0	0	0	0
Mul	zero	∞	zero	1	0	0	0	0
Mul	zero	NaN	zero	1	0	0	0	0
Mul	zero	denorm	zero	1	0	0	0	0
Mul	zero	zero	zero	0	0	0	0	0
Mul	zero	norm	zero	0	0	0	0	0
Mul	norm	∞	max	1	0	0	0	0
Mul	norm	NaN	max	1	0	0	0	0
Mul	norm	denorm	zero	1	0	0	0	0
Mul	norm	zero	zero	0	0	0	0	0
Mul	norm	norm	_Calc_	0	*	*	0	*
Divide[3]								
Div	∞	∞	zero	1	0	0	0	0
Div	∞	NaN	zero	1	0	0	0	0
Div	∞	denorm	max	1	0	0	0	0
Div	∞	zero	max	1	0	0	0	0
Div	∞	Norm	max	1	0	0	0	0
Div	NaN	∞	zero	1	0	0	0	0
Div	NaN	NaN	zero	1	0	0	0	0
Div	NaN	denorm	max	1	0	0	0	0
Div	NaN	zero	max	1	0	0	0	0
Div	NaN	norm	max	1	0	0	0	0
Div	denorm	∞	zero	1	0	0	0	0
Div	denorm	NaN	zero	1	0	0	0	0
Div	denorm	denorm	max	1	0	0	0	0
Div	denorm	zero	max	1	0	0	0	0
Div	denorm	norm	zero	1	0	0	0	0
Div	zero	∞	zero	1	0	0	0	0
Div	zero	NaN	zero	1	0	0	0	0
Div	zero	denorm	max	1	0	0	0	0
Div	zero	zero	max	1	0	0	0	0
Div	zero	norm	zero	0	0	0	0	0
Div	norm	∞	zero	1	0	0	0	0
Div	norm	NaN	zero	1	0	0	0	0
Div	norm	denorm	max	1	0	0	0	0
Div	norm	zero	max	0	0	0	1	0
Div	norm	norm	_Calc_	0	*	*	0	*

Table 116:Embedded Floating-Point Results Summary—Single Convert from Double						
Operand B	**efscfd result**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
+∞	pmax	1	0	0	0	0
-∞	nmax	1	0	0	0	0
+NaN	pmax	1	0	0	0	0
-NaN	nmax	1	0	0	0	0
+denorm	+zero	1	0	0	0	0
-denorm	-zero	1	0	0	0	0
+zero	+zero	0	0	0	0	0
-zero	-zero	0	0	0	0	0
norm	_Calc_	0	*	*	0	*

Table 117:Embedded Floating-Point Results Summary—Double Convert from Single						
Operand B	**efdcfs result**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
+∞	pmax	1	0	0	0	0
-∞	nmax	1	0	0	0	0
+NaN	pmax	1	0	0	0	0
-NaN	nmax	1	0	0	0	0
+denorm	+zero	1	0	0	0	0
-denorm	-zero	1	0	0	0	0
+zero	+zero	0	0	0	0	0
-zero	-zero	0	0	0	0	0
norm	_Calc_	0	0	0	0	0

Table 118:Embedded Floating-Point Results Summary—Convert to Unsigned								
Operand B	**Integer Result ctui[d][z]**	**Fractional Result ctuf**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
+∞	0xFFFF_FFFF 0xFFFF_FFFF_FFFF_FFFF	0x7FFF_FFFF	1	0	0	0	0
-∞	0	0	1	0	0	0	0
+NaN	0	0	1	0	0	0	0
-NaN	0	0	1	0	0	0	0
denorm	0	0	1	0	0	0	0
zero	0	0	0	0	0	0	0
+norm	_Calc_	_Calc_	*	0	0	0	*
-norm	_Calc_	_Calc_	*	0	0	0	*

Table 119:Embedded Floating-Point Results Summary—Convert to Signed							
Operand B	**Integer Result ctsi[d][z]**	**Fractional Result ctsf**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
+∞	0x7FFF_FFFF 0x7FFF_FFFF_FFFF_FFFF	0x7FFF_FFFF	1	0	0	0	0
-∞	0x8000_0000 0x8000_0000_0000_0000	0x8000_0000	1	0	0	0	0
+NaN	0	0	1	0	0	0	0
-NaN	0	0	1	0	0	0	0
denorm	0	0	1	0	0	0	0
zero	0	0	0	0	0	0	0
+norm	_Calc_	_Calc_	*	0	0	0	*
-norm	_Calc_	_Calc_	*	0	0	0	*

Table 120:Embedded Floating-Point Results Summary—Convert from Unsigned							
Operand B	**Integer Source cfui**	**Fractional Source cfuf**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 121:Embedded Floating-Point Results Summary—Convert from Signed							
Operand B	**Integer Source cfsi**	**Fractional Source cfsf**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 122:Embedded Floating-Point Results Summary—*abs, *nabs, *neg								
Operand A	***abs**	***nabs**	***neg**	**FINV**	**FOVF**	**FUNF**	**FDBZ**	**FINX**
+∞	pmax | +∞	nmax | -∞	-amax | -∞	1	0	0	0	0
-∞	pmax | +∞	nmax | -∞	-amax | +∞	1	0	0	0	0
+NaN	pmax | NaN	nmax | -NaN	-amax | -NaN	1	0	0	0	0
-NaN	pmax | NaN	nmax | -NaN	-amax | +NaN	1	0	0	0	0
+denorm	+zero | +denorm	-zero | -denorm	-zero | -denorm	1	0	0	0	0
-denorm	+zero | +denorm	-zero | -denorm	+zero | +denorm	1	0	0	0	0
+zero	+zero	-zero	-zero	0	0	0	0	0
-zero	+zero	-zero	+zero	0	0	0	0	0
+norm	+norm	-norm	-norm	0	0	0	0	0
-norm	+norm	-norm	+norm	0	0	0	0	0

Chapter 10. Legacy Move Assist Instruction [Category: Legacy Move Assist]

Determine Leftmost Zero Byte *X-form*

dlmzb RA,RS,RB (Rc=0)
dlmzb. RA,RS,RB (Rc=1)

31	RS	RA	RB	78	Rc
0	6	11	16	21	31

```
d0:63 ← (RS)32:63 || (RB)32:63
i ← 0
x ← 0
y ← 0
do while (x<8) & (y=0)
   x ← x + 1
   if di+32:i+39 = 0 then
      y ← 1
   else
      i ← i + 8
RA ← x
XER57:63 ← x
if Rc = 1 then do
   CR35 ← SO
   if y = 1 then do
      if x<5 then CR32:34 ← 0b010
      else        CR32:34 ← 0b100
   else
      CR32:34 ← 0b001
```

The contents of bits 32:63 of register RS and the contents of bits 32:63 of register RB are concatenated to form an 8-byte operand. The operand is searched for the leftmost byte in which each bit is 0 (i.e., a null byte).

Bytes in the operand are numbered from left to right starting with 1. If a null byte is found, its byte number is placed into bits 57:63 of the XER and into register RA. Otherwise, the value `0b000_1000` is placed into both bits 57:63 of the XER and register RA.

If Rc is equal to 1, SO is copied into bit 35 of the CR and bits 32:34 of the CR are updated as follows:

- If no null byte is found, bits 32:34 of the CR are set to `0b001`.
- If the leftmost null byte is in the first 4 bytes (i.e., from register RS), bits 32:34 of the CR are set to `0b010`.
- If the leftmost null byte is in the last 4 bytes (i.e., from register RB), bits 32:34 of the CR are set to `0b100`.

Special Registers Altered:

$XER_{57:63}$
CR0 (if Rc=1)

Chapter 11. Legacy Integer Multiply-Accumulate Instructions [Category: Legacy Integer Multiply-Accumulate]

The *Legacy Integer Multiply-Accumulate* instructions with Rc=1 set the first three bits of CR Field 0 based on the 32-bit result, as described in Section 3.3.8, “Other Fixed-Point Instructions”.

The XO-form *Legacy Integer Multiply-Accumulate* instructions set SO and OV when OE=1 to reflect overflow of the 32-bit result.

Programming Note

Notice that CR Field 0 may not reflect the “true” (infinitely precise) result if overflow occurs.

Multiply Accumulate Cross Halfword to Word Modulo Signed XO-form

macchw	RT,RA,RB	(OE=0 Rc=0)
macchw.	RT,RA,RB	(OE=0 Rc=1)
macchwo	RT,RA,RB	(OE=1 Rc=0)
macchwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	172	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Cross Halfword to Word Saturate Signed XO-form

macchws	RT,RA,RB	(OE=0 Rc=0)
macchws.	RT,RA,RB	(OE=0 Rc=1)
macchwso	RT,RA,RB	(OE=1 Rc=0)
macchwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	236	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + RT_{32:63}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the sum is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Cross Halfword to Word Modulo Unsigned XO-form

macchwu RT,RA,RB (OE=0 Rc=0)
macchwu. RT,RA,RB (OE=0 Rc=1)
macchwuo RT,RA,RB (OE=1 Rc=0)
macchwuo. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	140	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63}\ x_{ui}\ (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT \leftarrow temp_{1:32}$

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate Cross Halfword to Word Saturate Unsigned XO-form

macchwsu RT,RA,RB (OE=0 Rc=0)
macchwsu. RT,RA,RB (OE=0 Rc=1)
macchwsuo RT,RA,RB (OE=1 Rc=0)
macchwsuo. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	204	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63}\ x_{ui}\ (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp > 2^{32}-1 then $RT \leftarrow$ 0xFFFF_FFFF
else $RT \leftarrow temp_{1:32}$

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^{32}-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate High Halfword to Word Modulo Signed XO-form

machhw RT,RA,RB (OE=0 Rc=0)
machhw. RT,RA,RB (OE=0 Rc=1)
machhwo RT,RA,RB (OE=1 Rc=0)
machhwo. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	44	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate High Halfword to Word Saturate Signed XO-form

machhws RT,RA,RB (OE=0 Rc=0)
machhws. RT,RA,RB (OE=0 Rc=1)
machhwso RT,RA,RB (OE=1 Rc=0)
machhwso. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	108	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if $temp < -2^{31}$ then $RT_{32:63} \leftarrow$ 0x8000_0000
else if $temp > 2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the sum is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

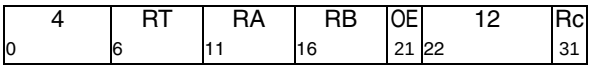
The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate High Halfword to Word Modulo Unsigned XO-form

machhwu RT,RA,RB (OE=0 Rc=0)
machhwu. RT,RA,RB (OE=0 Rc=1)
machhwuo RT,RA,RB (OE=1 Rc=0)
machhwuo. RT,RA,RB (OE=1 Rc=1)



$prod_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

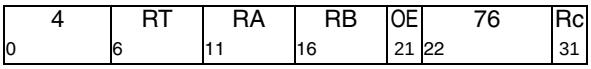
The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate High Halfword to Word Saturate Unsigned XO-form

machhwsu RT,RA,RB (OE=0 Rc=0)
machhwsu. RT,RA,RB (OE=0 Rc=1)
machhwsuo RT,RA,RB (OE=1 Rc=0)
machhwsuo. RT,RA,RB (OE=1 Rc=1)



$prod_{0:31} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp > 2^{32}-1 then RT ← 0xFFFF_FFFF
else RT ← $temp_{1:32}$

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^{32}-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate Low Halfword to Word Modulo Signed XO-form

maclhw	RT,RA,RB	(OE=0 Rc=0)
maclhw.	RT,RA,RB	(OE=0 Rc=1)
maclhwo	RT,RA,RB	(OE=1 Rc=0)
maclhwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	428	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Low Halfword to Word Saturate Signed XO-form

maclhws	RT,RA,RB	(OE=0 Rc=0)
maclhws.	RT,RA,RB	(OE=0 Rc=1)
maclhwso	RT,RA,RB	(OE=1 Rc=0)
maclhwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	492	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is added to the signed-integer word in bits 32:63 of register RT.

If the sum is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the sum is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Multiply Accumulate Low Halfword to Word Modulo Unsigned XO-form

maclhwu RT,RA,RB (OE=0 Rc=0)
maclhwu. RT,RA,RB (OE=0 Rc=1)
maclhwuo RT,RA,RB (OE=1 Rc=0)
maclhwuo. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	396	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

The low-order 32 bits of the sum are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Accumulate Low Halfword to Word Saturate Unsigned XO-form

maclhwsu RT,RA,RB (OE=0 Rc=0)
maclhwsu. RT,RA,RB (OE=0 Rc=1)
maclhwsuo RT,RA,RB (OE=1 Rc=0)
maclhwsuo. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	460	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$temp_{0:32} \leftarrow prod_{0:31} + (RT)_{32:63}$
if temp > 2^{32}-1 then RT ← 0xFFFF_FFFF
else RT ← $temp_{1:32}$

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB.

The 32-bit unsigned-integer product is added to the unsigned-integer word in bits 32:63 of register RT.

If the sum is greater than 2^{32}-1, then the value 0xFFFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the sum is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Multiply Cross Halfword to Word Signed X-form

mulchw RT,RA,RB (Rc=0)
mulchw. RT,RA,RB (Rc=1)

4	RT	RA	RB	168	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

CR0 (if Rc=1)

Multiply Cross Halfword to Word Unsigned X-form

mulchwu RT,RA,RB (Rc=0)
mulchwu. RT,RA,RB (Rc=1)

4	RT	RA	RB	136	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

CR0 (if Rc=1)

Multiply High Halfword to Word Signed X-form

mulhhw RT,RA,RB (Rc=0)
mulhhw. RT,RA,RB (Rc=1)

4	RT	RA	RB	40	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{32:47} \times_{si} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Multiply High Halfword to Word Unsigned X-form

mulhhwu RT,RA,RB (Rc=0)
mulhhwu. RT,RA,RB (Rc=1)

4	RT	RA	RB	8	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{32:47} \times_{ui} (RB)_{32:47}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 32:47 of register RA is multiplied by the unsigned-integer halfword in bits 32:47 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Multiply Low Halfword to Word Signed X-form

mullhw RT,RA,RB (Rc=0)
mullhw. RT,RA,RB (Rc=1)

4	RT	RA	RB	424	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB and the signed-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Multiply Low Halfword to Word Unsigned X-form

mullhwu RT,RA,RB (Rc=0)
mullhwu. RT,RA,RB (Rc=1)

4	RT	RA	RB	392	Rc
0	6	11	16	21	31

$RT_{32:63} \leftarrow (RA)_{48:63} \times_{ui} (RB)_{48:63}$
$RT_{0:31} \leftarrow$ undefined

The unsigned-integer halfword in bits 48:63 of register RA is multiplied by the unsigned-integer halfword in bits 48:63 of register RB and the unsigned-integer word result is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:
CR0 (if Rc=1)

Negative Multiply Accumulate Cross Halfword to Word Modulo Signed

XO-form

nmacchw RT,RA,RB (OE=0 Rc=0)
nmacchw. RT,RA,RB (OE=0 Rc=1)
nmacchwo RT,RA,RB (OE=1 Rc=0)
nmacchwo. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	174	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63}\ x_{si}\ (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63}\ -_{si}\ prod_{0:31}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Negative Multiply Accumulate Cross Halfword to Word Saturate Signed

XO-form

nmacchws RT,RA,RB (OE=0 Rc=0)
nmacchws. RT,RA,RB (OE=0 Rc=1)
nmacchwso RT,RA,RB (OE=1 Rc=0)
nmacchwso. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	238	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63}\ x_{si}\ (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63}\ -_{si}\ prod_{0:31}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the difference is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Negative Multiply Accumulate High Halfword to Word Modulo Signed

XO-form

nmachhw RT,RA,RB (OE=0 Rc=0)
nmachhw. RT,RA,RB (OE=0 Rc=1)
nmachhwo RT,RA,RB (OE=1 Rc=0)
nmachhwo. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	46	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47}\ {}^{x}_{si}\ (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63}\ {}^{-}_{si}\ prod_{0:31}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Negative Multiply Accumulate High Halfword to Word Saturate Signed

XO-form

nmachhws RT,RA,RB (OE=0 Rc=0)
nmachhws. RT,RA,RB (OE=0 Rc=1)
nmachhwso RT,RA,RB (OE=1 Rc=0)
nmachhwso. RT,RA,RB (OE=1 Rc=1)

4	RT	RA	RB	OE	110	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{32:47}\ {}^{x}_{si}\ (RB)_{32:47}$
$temp_{0:32} \leftarrow (RT)_{32:63}\ {}^{-}_{si}\ prod_{0:31}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 32:47 of register RA is multiplied by the signed-integer halfword in bits 32:47 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the difference is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV (if OE=1)
CR0 (if Rc=1)

Negative Multiply Accumulate Low Halfword to Word Modulo Signed

XO-form

nmaclhw	RT,RA,RB	(OE=0 Rc=0)
nmaclhw.	RT,RA,RB	(OE=0 Rc=1)
nmaclhwo	RT,RA,RB	(OE=1 Rc=0)
nmaclhwo.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	430	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
$RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

The low-order 32 bits of the difference are placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Negative Multiply Accumulate Low Halfword to Word Saturate Signed

XO-form

nmaclhws	RT,RA,RB	(OE=0 Rc=0)
nmaclhws.	RT,RA,RB	(OE=0 Rc=1)
nmaclhwso	RT,RA,RB	(OE=1 Rc=0)
nmaclhwso.	RT,RA,RB	(OE=1 Rc=1)

4	RT	RA	RB	OE	494	Rc
0	6	11	16	21	22	31

$prod_{0:31} \leftarrow (RA)_{48:63} \times_{si} (RB)_{48:63}$
$temp_{0:32} \leftarrow (RT)_{32:63} -_{si} prod_{0:31}$
if temp < -2^{31} then $RT_{32:63} \leftarrow$ 0x8000_0000
else if temp > $2^{31}-1$ then $RT_{32:63} \leftarrow$ 0x7FFF_FFFF
else $RT_{32:63} \leftarrow temp_{1:32}$
$RT_{0:31} \leftarrow$ undefined

The signed-integer halfword in bits 48:63 of register RA is multiplied by the signed-integer halfword in bits 48:63 of register RB.

The 32-bit signed-integer product is subtracted from the signed-integer word in bits 32:63 of register RT.

If the difference is less than -2^{31}, then the value 0x8000_0000 is placed into bits 32:63 of register RT.

If the difference is greater than $2^{31}-1$, then the value 0x7FFF_FFFF is placed into bits 32:63 of register RT.

Otherwise, the difference is placed into bits 32:63 of register RT.

The contents of bits 0:31 of register RT are undefined.

Special Registers Altered:

SO OV	(if OE=1)
CR0	(if Rc=1)

Appendix A. Suggested Floating-Point Models [Category: Floating-Point]

A.1 Floating-Point Round to Single-Precision Model

The following describes algorithmically the operation of the *Floating Round to Single-Precision* instruction.

```
If (FRB)1:11 < 897 and (FRB)1:63 > 0 then
    Do
        If FPSCRUE = 0 then goto Disabled Exponent Underflow
        If FPSCRUE = 1 then goto Enabled Exponent Underflow
    End

If (FRB)1:11 > 1150 and (FRB)1:11 < 2047 then
    Do
        If FPSCROE = 0 then goto Disabled Exponent Overflow
        If FPSCROE = 1 then goto Enabled Exponent Overflow
    End

If (FRB)1:11 > 896 and (FRB)1:11 < 1151 then goto Normal Operand

If (FRB)1:63 = 0 then goto Zero Operand

If (FRB)1:11 = 2047 then
    Do
        If (FRB)12:63 = 0 then goto Infinity Operand
        If (FRB)12 = 1 then goto QNaN Operand
        If (FRB)12 = 0 and (FRB)13:63 > 0 then goto SNaN Operand
    End
```

Disabled Exponent Underflow:

```
sign ← (FRB)0
If (FRB)1:11 = 0 then
    Do
        exp ← -1022
        frac0:52 ← 0b0 || (FRB)12:63
    End
If (FRB)1:11 > 0 then
    Do
        exp ← (FRB)1:11 - 1023
        frac0:52 ← 0b1 || (FRB)12:63
    End
Denormalize operand:
    G || R || X ← 0b000
    Do while exp < -126
        exp ← exp + 1
        frac0:52 || G || R || X ← 0b0 || frac0:52 || G || (R | X)
    End
FPSCRUX ← (frac24:52 || G || R || X) > 0
Round Single(sign,exp,frac0:52,G,R,X)
FPSCRXX ← FPSCRXX | FPSCRFI
If frac0:52 = 0 then
    Do
```

```
            FRT0 ← sign
            FRT1:63 ← 0
            If sign = 0 then FPSCRFPRF ← "+ zero"
            If sign = 1 then FPSCRFPRF ← "- zero"
        End
    If frac0:52 > 0 then
        Do
            If frac0 = 1 then
                Do
                    If sign = 0 then FPSCRFPRF ← "+ normal number"
                    If sign = 1 then FPSCRFPRF ← "- normal number"
                End
            If frac0 = 0 then
                Do
                    If sign = 0 then FPSCRFPRF ← "+ denormalized number"
                    If sign = 1 then FPSCRFPRF ← "- denormalized number"
                End
            Normalize operand:
                Do while frac0 = 0
                    exp ← exp-1
                    frac0:52 ← frac1:52 || 0b0
                End
            FRT0 ← sign
            FRT1:11 ← exp + 1023
            FRT12:63 ← frac1:52
        End
    Done
```

Enabled Exponent Underflow:

```
    FPSCRUX ← 1
    sign ← (FRB)0
    If (FRB)1:11 = 0 then
        Do
            exp ← -1022
            frac0:52 ← 0b0 || (FRB)12:63
        End
    If (FRB)1:11 > 0 then
        Do
            exp ← (FRB)1:11 - 1023
            frac0:52 ← 0b1 || (FRB)12:63
        End
    Normalize operand:
        Do while frac0 = 0
            exp ← exp - 1
            frac0:52 ← frac1:52 || 0b0
        End
    Round Single(sign,exp,frac0:52,0,0,0)
    FPSCRXX ← FPSCRXX | FPSCRFI
    exp ← exp + 192
    FRT0 ← sign
    FRT1:11 ← exp + 1023
    FRT12:63 ← frac1:52
    If sign = 0 then FPSCRFPRF ← "+ normal number"
    If sign = 1 then FPSCRFPRF ← "- normal number"
    Done
```

Disabled Exponent Overflow:

```
    FPSCROX ← 1
    If FPSCRRN = 0b00 then              /* Round to Nearest */
        Do
            If (FRB)0 = 0 then FRT ← 0x7FF0_0000_0000_0000
            If (FRB)0 = 1 then FRT ← 0xFFF0_0000_0000_0000
            If (FRB)0 = 0 then FPSCRFPRF ← "+ infinity"
            If (FRB)0 = 1 then FPSCRFPRF ← "- infinity"
        End
```

If $FPSCR_{RN}$ = 0b01 then /* Round toward Zero */
Do
If $(FRB)_0$ = 0 then FRT ← 0x47EF_FFFF_E000_0000
If $(FRB)_0$ = 1 then FRT ← 0xC7EF_FFFF_E000_0000
If $(FRB)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ normal number"
If $(FRB)_0$ = 1 then $FPSCR_{FPRF}$ ← "- normal number"
End
If $FPSCR_{RN}$ = 0b10 then /* Round toward +Infinity */
Do
If $(FRB)_0$ = 0 then FRT ← 0x7FF0_0000_0000_0000
If $(FRB)_0$ = 1 then FRT ← 0xC7EF_FFFF_E000_0000
If $(FRB)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ infinity"
If $(FRB)_0$ = 1 then $FPSCR_{FPRF}$ ← "- normal number"
End
If $FPSCR_{RN}$ = 0b11 then /* Round toward -Infinity */
Do
If $(FRB)_0$ = 0 then FRT ← 0x47EF_FFFF_E000_0000
If $(FRB)_0$ = 1 then FRT ← 0xFFF0_0000_0000_0000
If $(FRB)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ normal number"
If $(FRB)_0$ = 1 then $FPSCR_{FPRF}$ ← "- infinity"
End
$FPSCR_{FR}$ ← undefined
$FPSCR_{FI}$ ← 1
$FPSCR_{XX}$ ← 1
Done

Enabled Exponent Overflow:
sign ← $(FRB)_0$
exp ← $(FRB)_{1:11}$ - 1023
$frac_{0:52}$ ← 0b1 || $(FRB)_{12:63}$
Round Single(sign,exp,$frac_{0:52}$,0,0,0)
$FPSCR_{XX}$ ← $FPSCR_{XX}$ | $FPSCR_{FI}$
Enabled Overflow:
$FPSCR_{OX}$ ← 1
exp ← exp - 192
FRT_0 ← sign
$FRT_{1:11}$ ← exp + 1023
$FRT_{12:63}$ ← $frac_{1:52}$
If sign = 0 then $FPSCR_{FPRF}$ ← "+ normal number"
If sign = 1 then $FPSCR_{FPRF}$ ← "- normal number"
Done

Zero Operand:
FRT ← (FRB)
If $(FRB)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ zero"
If $(FRB)_0$ = 1 then $FPSCR_{FPRF}$ ← "- zero"
$FPSCRFR_{FI}$ ← 0b00
Done

Infinity Operand:
FRT ← (FRB)
If $(FRB)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ infinity"
If $(FRB)_0$ = 1 then $FPSCR_{FPRF}$ ← "- infinity"
$FPSCRFR_{FI}$ ← 0b00
Done

QNaN Operand:
FRT ← $(FRB)_{0:34}$ || $^{29}0$
$FPSCR_{FPRF}$ ← "QNaN"
$FPSCR_{FR\ FI}$ ← 0b00
Done

SNaN Operand:

$FPSCR_{VXSNAN} \leftarrow 1$
If $FPSCR_{VE} = 0$ then
  Do
    $FRT_{0:11} \leftarrow (FRB)_{0:11}$
    $FRT_{12} \leftarrow 1$
    $FRT_{13:63} \leftarrow (FRB)_{13:34} \,||\, {}^{29}0$
    $FPSCR_{FPRF} \leftarrow$ "QNaN"
  End
$FPSCR_{FR\ FI} \leftarrow 0b00$
Done

Normal Operand:

$sign \leftarrow (FRB)_0$
$exp \leftarrow (FRB)_{1:11} - 1023$
$frac_{0:52} \leftarrow 0b1 \,||\, (FRB)_{12:63}$
Round Single(sign,exp,$frac_{0:52}$,0,0,0)
$FPSCR_{XX} \leftarrow FPSCR_{XX} \,|\, FPSCR_{FI}$
If exp > 127 and $FPSCR_{OE} = 0$ then go to Disabled Exponent Overflow
If exp > 127 and $FPSCR_{OE} = 1$ then go to Enabled Overflow
$FRT_0 \leftarrow sign$
$FRT_{1:11} \leftarrow exp + 1023$
$FRT_{12:63} \leftarrow frac_{1:52}$
If sign = 0 then $FPSCR_{FPRF} \leftarrow$ "+ normal number"
If sign = 1 then $FPSCR_{FPRF} \leftarrow$ "- normal number"
Done

Round Single(sign,exp,$frac_{0:52}$,G,R,X):

$inc \leftarrow 0$
$lsb \leftarrow frac_{23}$
$gbit \leftarrow frac_{24}$
$rbit \leftarrow frac_{25}$
$xbit \leftarrow (frac_{26:52} || G || R || X) \neq 0$
If $FPSCR_{RN} = 0b00$ then /* Round to Nearest */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0bu11uu then $inc \leftarrow 1$
    If sign || lsb || gbit || rbit || xbit = 0bu011u then $inc \leftarrow 1$
    If sign || lsb || gbit || rbit || xbit = 0bu01u1 then $inc \leftarrow 1$
  End
If $FPSCR_{RN} = 0b10$ then /* Round toward + Infinity */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0b0u1uu then $inc \leftarrow 1$
    If sign || lsb || gbit || rbit || xbit = 0b0uu1u then $inc \leftarrow 1$
    If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then $inc \leftarrow 1$
  End
If $FPSCR_{RN} = 0b11$ then /* Round toward - Infinity */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0b1u1uu then $inc \leftarrow 1$
    If sign || lsb || gbit || rbit || xbit = 0b1uu1u then $inc \leftarrow 1$
    If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then $inc \leftarrow 1$
  End
$frac_{0:23} \leftarrow frac_{0:23} + inc$
If carry_out = 1 then
  Do
    $frac_{0:23} \leftarrow 0b1 \,||\, frac_{0:22}$
    $exp \leftarrow exp + 1$
  End
$frac_{24:52} \leftarrow {}^{29}0$
$FPSCR_{FR} \leftarrow inc$
$FPSCR_{FI} \leftarrow gbit \,|\, rbit \,|\, xbit$
Return

A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the *Floating Convert To Integer* instructions.

```
if Floating Convert To Integer Word then do
   round_mode    ← FPSCR_RN
   tgt_precision ← “32-bit signed integer”
end

if Floating Convert To Integer Word Unsigned then do
   round_mode    ← FPSCR_RN
   tgt_precision ← “32-bit unsigned integer”
end

if Floating Convert To Integer Word with round toward Zero then do
   round_mode    ← 0b01
   tgt_precision ← “32-bit signed integer”
end

if Floating Convert To Integer Word Unsigned with round toward Zero then do
   round_mode    ← 0b01
   tgt_precision ← “32-bit unsigned integer”
end

if Floating Convert To Integer Doubleword then do
   round_mode    ← FPSCR_RN
   tgt_precision ← “64-bit signed integer”
end

if Floating Convert To Integer Doubleword Unsigned then do
   round_mode    ← FPSCR_RN
   tgt_precision ← “64-bit unsigned integer”
end

if Floating Convert To Integer Doubleword with round toward Zero then do
   round_mode ← 0b01
   tgt_precision ← “64-bit signed integer”
end

if Floating Convert To Integer Doubleword Unsigned with round toward Zero then do
   round_mode    ← 0b01
   tgt_precision ← “64-bit unsigned integer”
end

sign ← (FRB)_0
if (FRB)_1:11 = 2047 and (FRB)_12:63 = 0 then goto Infinity Operand
if (FRB)_1:11 = 2047 and (FRB)_12 = 0 then goto SNaN Operand
if (FRB)_1:11 = 2047 and (FRB)_12 = 1 then goto QNaN Operand
if (FRB)_1:11 > 1086 then goto Large Operand

if (FRB)_1:11 > 0 then exp ← (FRB)_1:11 - 1023   /* exp - bias */
if (FRB)_1:11 = 0 then exp ← -1022
if (FRB)_1:11 > 0 then frac_0:64 ← 0b01 || (FRB)_12:63 || 11^0   /* normal */
if (FRB)_1:11 = 0 then frac_0:64 ← 0b00 || (FRB)_12:63 || 11^0   /* denormal */

gbit || rbit || xbit ← 0b000
do i=1,63-exp   /* do the loop 0 times if exp = 63 */
   frac_0:64 || gbit || rbit || xbit ← 0b0 || frac_0:64 || gbit || (rbit | xbit)
end

Round Integer( sign, frac_0:64, gbit, rbit, xbit, round_mode )

if sign = 1 then frac_0:64 ← ¬frac_0:64 + 1  /* needed leading 0 for -2^64<(FRB)<-2^63 */
```

```
if tgt_precision = “32-bit signed integer” and frac0:64 > 2^31-1 then
   goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 > 2^63-1 then
   goto Large Operand
if tgt_precision = “32-bit signed integer” and frac0:64 < -2^31  then
   goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 < -2^63  then
   goto Large Operand

if tgt_precision = “32-bit unsigned integer” & frac0:64 > 2^32-1 then
   goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 > 2^64-1 then
   goto Large Operand
if tgt_precision = “32-bit unsigned integer” & frac0:64 < 0 then
   goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 < 0 then
   goto Large Operand

FPSCRXX ← FPSCRXX | FPSCRFI

if tgt_precision = “32-bit signed integer”   then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “64-bit signed integer”   then FRT ← frac1:64
if tgt_precision = “64-bit unsigned integer” then FRT ← frac1:64
FPSCRFPRF ← 0bUUUUU
done
```

Round Integer(sign, $frac_{0:64}$, gbit, rbit, xbit, round_mode):

```
inc ← 0
if round_mode = 0b00 then do   /* Round to Nearest */
   if sign || frac64 || gbit || rbit || xbit = 0bU11UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0bU011U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end
if round_mode = 0b10 then do   /* Round toward +Infinity */
   if sign || frac64 || gbit || rbit || xbit = 0b0U1UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b0UU1U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end
if round_mode = 0b11 then do   /* Round toward -Infinity */
   if sign || frac64 || gbit || rbit || xbit = 0b1U1UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b1UU1U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end
frac0:64 ← frac0:64 + inc
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
return
```

Infinity Operand:

```
FPSCRFR ← 0b0
FPSCRFI ← 0b0
FPSCRVXCVI ← 0b1
if FPSCRVE = 0 then do
   if tgt_precision = “32-bit signed integer” then do
      if sign=0 then FRT ← 0xUUUU_UUUU_7FFF_FFFF
      if sign=1 then FRT ← 0xUUUU_UUUU_8000_0000
   end
   else if tgt_precision = “32-bit unsigned integer” then do
      if sign=0 then FRT ← 0xUUUU_UUUU_FFFF_FFFF
      if sign=1 then FRT ← 0xUUUU_UUUU_0000_0000
   end
   else if tgt_precision = “64-bit signed integer” then do
      if sign=0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
      if sign=1 then FRT ← 0x8000_0000_0000_0000
```

```
      end

      else if tgt_precision = “64-bit unsigned integer” then do
         if sign=0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
         if sign=1 then FRT ← 0x0000_0000_0000_0000
      end
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

SNaN Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 0b0
   FPSCR_VXSNAN ← 0b1
   FPSCR_VXCVI ← 0b1
   if FPSCR_VE = 0 then do
      if tgt_precision = “32-bit signed integer” then FRT ← 0xUUUU_UUUU_8000_0000
      if tgt_precision = “64-bit signed integer” then FRT ← 0x8000_0000_0000_0000
      if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU_0000_0000
      if tgt_precision = “64-bit unsigned integer” then FRT ← 0x0000_0000_0000_0000
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

QNaN Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 0b0
   FPSCR_VXCVI ← 0b1
   if FPSCR_VE = 0 then do
      if tgt_precision = “32-bit signed integer” then FRT ← 0xUUUU_UUUU_8000_0000
      if tgt_precision = “64-bit signed integer” then FRT ← 0x8000_0000_0000_0000
      if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU_0000_0000
      if tgt_precision = “64-bit unsigned integer” then FRT ← 0x0000_0000_0000_0000
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

Large Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 0b0
   FPSCR_VXCVI ← 0b1
   if FPSCR_VE = 0 then do
      if tgt_precision = “32-bit signed integer” then do
         if sign = 0 then FRT ← 0xUUUU_UUUU_7FFF_FFFF
         if sign = 1 then FRT ← 0xUUUU_UUUU_8000_0000
      end
      else if tgt_precision = “64-bit signed integer” then do
         if sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
         if sign = 1 then FRT ← 0x8000_0000_0000_0000
      end
      else if tgt_precision = “32-bit unsigned integer” then do
         if sign = 0 then FRT ← 0xUUUU_UUUU_FFFF_FFFF
         if sign = 1 then FRT ← 0xUUUU_UUUU_0000_0000
      end
      else if tgt_precision = “64-bit unsigned integer” then do
         if sign = 0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
         if sign = 1 then FRT ← 0x0000_0000_0000_0000
      end
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert From Integer* instructions.

```
if Floating Convert From Integer Doubleword then do
   tgt_precision ← "double-precision"
   sign    ← (FRB)0
   exp     ← 63
   frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Single then do
   tgt_precision ← "single-precision"
   sign    ← (FRB)0
   exp     ← 63
   frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Unsigned then do
   tgt_precision ← "double-precision"
   sign    ← 0
   exp     ← 63
   frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Unsigned Single then do
   tgt_precision ← "single-precision"
   sign    ← 0
   exp     ← 63
   frac0:63 ← (FRB)
end

if frac0:63 = 0 then go to Zero Operand
if sign = 1 then frac0:63 ← ¬frac0:63 + 1

/* do the loop 0 times if (FRB) = max negative 64-bit integer or */
/*                     if (FRB) = max unsigned 64-bit integer    */
do while frac0 = 0
   frac0:63 ← frac1:63 || 0b0
   exp ← exp - 1
end

Round Float( sign, exp, frac0:63, RN )
if sign = 0 then FPSCRFPRF ← "+normal number"
if sign = 1 then FPSCRFPRF ← "-normal number"
FRT0     ← sign
FRT1:11  ← exp + 1023   /* exp + bias */
FRT12:63 ← frac1:52
done
```

Zero Operand:

```
FPSCRFR ← 0b00
FPSCRFI ← 0b00
FPSCRFPRF ← "+ zero"
FRT ← 0x0000_0000_0000_0000
done
```

Round Float(sign, exp, $frac_{0:63}$, round_mode):

```
inc  ← 0

if tgt_precision = "single-precision" then do
   lsb  ← frac23
   gbit ← frac24
   rbit ← frac25
   xbit ← frac26:63 > 0
end
else do  /* tgt_precision = "double-precision" */
```

```
    lsb  ← frac52
    gbit ← frac53
    rbit ← frac54
    xbit ← frac55:63 > 0
end

if round_mode = 0b00 then do                /* Round to Nearest */
    if sign || lsb || gbit || rbit || xbit = 0bU11UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0bU011U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end
if round_mode = 0b10 then do                /* Round toward + Infinity */
    if sign || lsb || gbit || rbit || xbit = 0b0U1UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b0UU1U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end
if round_mode = 0b11 then do                /* Round toward - Infinity */
    if sign || lsb || gbit || rbit || xbit = 0b1U1UU then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b1UU1U then inc ← 1
    if sign || lsb || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end

if tgt_precision = “single-precision” then
    frac0:23 ← frac0:23 + inc
else  /* tgt_precision = “double-precision” */
    frac0:52 ← frac0:52 + inc

if carry_out = 1 then exp ← exp + 1

FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
FPSCRXX ← FPSCRXX | FPSCRFI
return
```

A.4 Floating-Point Round to Integer Model

The following describes algorithmically the operation of the *Floating Round To Integer* instructions.

```
If (FRB)1:11 = 2047 and (FRB)12:63 = 0, then goto Infinity Operand
If (FRB)1:11 = 2047 and (FRB)12 = 0, then goto SNaN Operand
If (FRB)1:11 = 2047 and (FRB)12 = 1, then goto QNaN Operand
if (FRB)1:63 = 0 then goto Zero Operand
If (FRB)1:11 < 1023 then goto Small Operand /* exp < 0; |value| < 1*/
If (FRB)1:11 > 1074 then goto Large Operand /* exp > 51; integral value */

sign ← (FRB)0
exp ← (FRB)1:11 - 1023 /* exp - bias */
frac0:52 ← 0b1 || (FRB)12:63
gbit || rbit || xbit ← 0b000

Do i = 1, 52 - exp
     frac0:52 || gbit || rbit || xbit ← 0b0 || frac0:52 || gbit || (rbit | xbit)
End

Round Integer (sign, frac0:52, gbit, rbit, xbit)

Do i = 2, 52 - exp
     frac0:52 ← frac1:52 || 0b0
End

If frac0 = 1, then exp ← exp + 1
Else frac0:52 ← frac1:52 || 0b0

FRT0 ← sign
FRT1:11 ← exp + 1023
FRT12:63 ← frac1:52

If (FRT)0 = 0 then FPSCRFPRF ← "+ normal number"
Else FPSCRFPRF ← "- normal number"
FPSCRFR FI  ← 0b00
Done

Round Integer(sign, frac0:52, gbit, rbit, xbit):
     inc ← 0
     If inst = Floating Round to Integer Nearest then          /* ties away from zero */
          Do   /* comparisons ignore u bits */
               If sign || frac52 || gbit || rbit || xbit = 0buu1uu then inc ← 1
          End
     If inst = Floating Round to Integer Plus then
          Do   /* comparisons ignore u bits */
               If sign || frac52 || gbit || rbit || xbit = 0b0u1uu then inc ← 1
               If sign || frac52 || gbit || rbit || xbit = 0b0uu1u then inc ← 1
               If sign || frac52 || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
          End
     If inst = Floating Round to Integer Minus then
          Do   /* comparisons ignore u bits */
               If sign || frac52 || gbit || rbit || xbit = 0b1u1uu then inc ← 1
               If sign || frac52 || gbit || rbit || xbit = 0b1uu1u then inc ← 1
               If sign || frac52 || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
          End
     frac0:52  ← frac0:52 + inc
     Return
```

Infinity Operand:

FRT ← (FRB)
If $(FRB)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ infinity"
If $(FRB)_0$ = 1 then $FPSCR_{FPRF}$ ← "- infinity"
$FPSCR_{FR\ FI}$ ← 0b00
Done

SNaN Operand:

$FPSCR_{VXSNAN}$ ← 1
If $FPSCR_{VE}$ = 0 then
    Do
        FRT ← (FRB)
        FRT_{12} ← 1
        $FPSCR_{FPRF}$ ← "QNaN"
    End
$FPSCR_{FR\ FI}$ ← 0b00
Done

QNaN Operand:

FRT ← (FRB)
$FPSCR_{FPRF}$ ← "QNaN"
$FPSCR_{FR\ FI}$ ← 0b00
Done

Zero Operand:

If $(FRB)_0$ = 0 then
    Do
        FRT ← 0x0000_0000_0000_0000
        $FPSCR_{FPRF}$ ← "+ zero"
    End
Else
    Do
        FRT ← 0x8000_0000_0000_0000
        $FPSCR_{FPRF}$ ← "- zero"
    End
$FPSCR_{FR\ FI}$ ← 0b00
Done

Small Operand:

If inst = Floating Round to Integer Nearest and $(FRB)_{1:11}$ < 1022 then goto Zero Operand
If inst = Floating Round to Integer Toward Zero then goto Zero Operand
If inst = Floating Round to Integer Plus and $(FRB)_0$ = 1 then goto Zero Operand
If inst = Floating Round to Integer Minus and $(FRB)_0$ = 0 then goto Zero Operand

If $(FRB)_0$ = 0 then
    Do
        FRT ← 0x3FF0_0000_0000_0000
/* value = 1.0 */
        $FPSCR_{FPRF}$ ← "+ normal number"
    End
Else
    Do
        FRT ← 0xBFF0_0000_0000_0000
/* value = -1.0 */
        $FPSCR_{FPRF}$ ← "- normal number"
    End
$FPSCR_{FR\ FI}$ ← 0b00
Done

Large Operand:

FRT ← (FRB)
If FRT_0 = 0 then $FPSCR_{FPRF}$ ← "+ normal number"
Else $FPSCR_{FPRF}$ ← "- normal number"
$FPSCR_{FR\ FI}$ ← 0b00
Done

Appendix B. Densely Packed Decimal

The trailing significand field of the decimal floating-point data format is encoded using Densely Packed Decimal (DPD). DPD encoding is a compression technique which supports the representation of decimal integers of arbitrary length. Translation operates on three Binary Coded Decimal (BCD) digits at a time compressing the 12 bits into 10 bits with an algorithm that can be applied or reversed using simple Boolean operations. In the following examples, a 3-digit BCD number is represented as (abcd)(efgh)(ijkm), a 10-bit DPD number is represented as (pqr)(stu)(v)(wxy), and the Boolean operations, & (AND), | (OR), and ¬ (NOT) are used.

B.1 BCD-to-DPD Translation

The translation from a 3-digit BCD number to a 10-bit DPD can be performed through the following Boolean operations.

```
p = (f & a & i & ¬e) | (j & a & ¬i) | (b & ¬a)
q = (g & a & i & ¬e) | (k & a & ¬i) | (c & ¬a)
r = d

s = (j & ¬a & e & ¬i) | (f & ¬i & ¬e) |
    (f & ¬a & ¬e) | (e & i)
t = (k & ¬a & e & ¬i) | (g & ¬i & ¬e) |
    (g & ¬a & ¬e) | (a & i)
u = h

v = a | e | i

w = (¬e & j & ¬i) | (e & i) | a
x = (¬a & k & ¬i) | (a & i) | e
y = m
```

Alternatively, the following table can be used to perform the translation. The most significant bit of the three BCD digits (left column) is used to select a specific 10-bit encoding (right column) of the DPD.

aei	pqr stu v wxy
000	bcd fgh 0 jkm
001	bcd fgh 1 00m
010	bcd jkh 1 01m
011	bcd 10h 1 11m
100	jkd fgh 1 10m
101	fgd 01h 1 11m
110	jkd 00h 1 11m
111	00d 11h 1 11m

The full translation of a 3-digit BCD number (000 - 999) to a 10-bit DPD is shown in Table 123 on page 699, with the DPD entries shown in hexadecimal format. The BCD number is produced by replacing '_' in the leftmost column with the corresponding digit along the top row. The table is split into two halves, with the right half being a continuation of the left half.

B.2 DPD-to-BCD Translation

The translation from a 10-bit DPD to a 3-digit BCD number can be performed through the following Boolean operations.

```
a = (¬s & v & w) | (t & v & w & s) | (v & w & ¬x)
b = (p & s & x & ¬t) | (p & ¬w) | (p & ¬v)
c = (q & s & x & ¬t) | (q & ¬w) | (q & ¬v)
d = r

e = (v & ¬w & x) | (s & v & w & x) |
    (¬t & v & x & w)
f = (p & t & v & w & x & ¬s) | (s & ¬x & v) |
    (s & ¬v)
g = (q & t & w & v & x & ¬s) | (t & ¬x & v) |
    (t & ¬v)
h = u

i = (t & v & w & x) | (s & v & w & x) |
    (v & ¬w & ¬x)
j = (p & ¬s & ¬t & w & v) | (s & v & ¬w & x) |
    (p & w & ¬x & v) | (w & ¬v)
k = (q & ¬s & ¬t & v & w) | (t & v & ¬w & x) |
    (q & v & w & ¬x) | (x & ¬v)
m = y
```

Alternatively, the following table can be used to perform the translation. A combination of five bits in the DPD encoding (leftmost column) are used to specify a translation to the 3-digit BCD encoding. Dashes (-) in the table are don't cares, and can be either one or zero.

vwxst	abcd	efgh	ijkm
0----	0pqr	0stu	0wxy
100--	0pqr	0stu	100y
101--	0pqr	100u	0sty
110--	100r	0stu	0pqy
11100	100r	100u	0pqy
11101	100r	0pqu	100y
11110	0pqr	100u	100y
11111	100r	100u	100y

The full translation of the 10-bit DPD to a 3-digit BCD number is shown in Table 124 on page 700. The 10-bit DPD index is produced by concatenating the 6-bit value shown in the left column with the 4-bit index along the top row, both represented in hexadecimal. The values in parentheses are non-preferred translations and are explained further in the following section.

B.3 Preferred DPD encoding

Translating from a 3-digit BCD number (1000 numbers) to a 10-bit DPD encoding (1024 combinations) leaves 24 redundant translations. The 24 redundant combinations are evenly assigned to eight BCD numbers and are shown in the following table, with the non-preferred encoding in parentheses. The preferred encoding is produced by translating a 3-digit BCD number with the translation table or Boolean operations shown in Section B.1. The redundant DPD encodings are all valid and will be correctly translated to their respective BCD value through the mechanisms provided in Section B.2. For decimal floating-point operations all DPD encodings are recognized as source operands.

DPD Code	BCD Value	DPD Code	BCD Value
0x06E	888	0x0EE	988
(0x16E)		(0x1EE)	
(0x26E)		(0x2EE)	
(0x36E)		(0x3EE)	
0x06F	889	0x0EF	989
(0x16F)		(0x1EF)	
(0x26F)		(0x2EF)	
(0x36F)		(0x3EF)	
0x07E	898	0x0FE	998
(0x17E)		(0x1FE)	
(0x27E)		(0x2FE)	
(0x37E)		(0x3FE)	
0x07F	899	0x0FF	999
(0x17F)		(0x1FF)	
(0x27F)		(0x2FF)	
(0x37F)		(0x3FF)	

Table 123:BCD-to-DPD translation

	0	1	2	3	4	5	6	7	8	9
00_	000	001	002	003	004	005	006	007	008	009
01_	010	011	012	013	014	015	016	017	018	019
02_	020	021	022	023	024	025	026	027	028	029
03_	030	031	032	033	034	035	036	037	038	039
04_	040	041	042	043	044	045	046	047	048	049
05_	050	051	052	053	054	055	056	057	058	059
06_	060	061	062	063	064	065	066	067	068	069
07_	070	071	072	073	074	075	076	077	078	079
08_	00A	00B	02A	02B	04A	04B	06A	06B	04E	04F
09_	01A	01B	03A	03B	05A	05B	07A	07B	05E	05F
10_	080	081	082	083	084	085	086	087	088	089
11_	090	091	092	093	094	095	096	097	098	099
12_	0A0	0A1	0A2	0A3	0A4	0A5	0A6	0A7	0A8	0A9
13_	0B0	0B1	0B2	0B3	0B4	0B5	0B6	0B7	0B8	0B9
14_	0C0	0C1	0C2	0C3	0C4	0C5	0C6	0C7	0C8	0C9
15_	0D0	0D1	0D2	0D3	0D4	0D5	0D6	0D7	0D8	0D9
16_	0E0	0E1	0E2	0E3	0E4	0E5	0E6	0E7	0E8	0E9
17_	0F0	0F1	0F2	0F3	0F4	0F5	0F6	0F7	0F8	0F9
18_	08A	08B	0AA	0AB	0CA	0CB	0EA	0EB	0CE	0CF
19_	09A	09B	0BA	0BB	0DA	0DB	0FA	0FB	0DE	0DF
20_	100	101	102	103	104	105	106	107	108	109
21_	110	111	112	113	114	115	116	117	118	119
22_	120	121	122	123	124	125	126	127	128	129
23_	130	131	132	133	134	135	136	137	138	139
24_	140	141	142	143	144	145	146	147	148	149
25_	150	151	152	153	154	155	156	157	158	159
26_	160	161	162	163	164	165	166	167	168	169
27_	170	171	172	173	174	175	176	177	178	179
28_	10A	10B	12A	12B	14A	14B	16A	16B	14E	14F
29_	11A	11B	13A	13B	15A	15B	17A	17B	15E	15F
30_	180	181	182	183	184	185	186	187	188	189
31_	190	191	192	193	194	195	196	197	198	199
32_	1A0	1A1	1A2	1A3	1A4	1A5	1A6	1A7	1A8	1A9
33_	1B0	1B1	1B2	1B3	1B4	1B5	1B6	1B7	1B8	1B9
34_	1C0	1C1	1C2	1C3	1C4	1C5	1C6	1C7	1C8	1C9
35_	1D0	1D1	1D2	1D3	1D4	1D5	1D6	1D7	1D8	1D9
36_	1E0	1E1	1E2	1E3	1E4	1E5	1E6	1E7	1E8	1E9
37_	1F0	1F1	1F2	1F3	1F4	1F5	1F6	1F7	1F8	1F9
38_	18A	18B	1AA	1AB	1CA	1CB	1EA	1EB	1CE	1CF
39_	19A	19B	1BA	1BB	1DA	1DB	1FA	1FB	1DE	1DF
40_	200	201	202	203	204	205	206	207	208	209
41_	210	211	212	213	214	215	216	217	218	219
42_	220	221	222	223	224	225	226	227	228	229
43_	230	231	232	233	234	235	236	237	238	239
44_	240	241	242	243	244	245	246	247	248	249
45_	250	251	252	253	254	255	256	257	258	259
46_	260	261	262	263	264	265	266	267	268	269
47_	270	271	272	273	274	275	276	277	278	279
48_	20A	20B	22A	22B	24A	24B	26A	26B	24E	24F
49_	21A	21B	23A	23B	25A	25B	27A	27B	25E	25F

	0	1	2	3	4	5	6	7	8	9
50_	280	281	282	283	284	285	286	287	288	289
51_	290	291	292	293	294	295	296	297	298	299
52_	2A0	2A1	2A2	2A3	2A4	2A5	2A6	2A7	2A8	2A9
53_	2B0	2B1	2B2	2B3	2B4	2B5	2B6	2B7	2B8	2B9
54_	2C0	2C1	2C2	2C3	2C4	2C5	2C6	2C7	2C8	2C9
55_	2D0	2D1	2D2	2D3	2D4	2D5	2D6	2D7	2D8	2D9
56_	2E0	2E1	2E2	2E3	2E4	2E5	2E6	2E7	2E8	2E9
57_	2F0	2F1	2F2	2F3	2F4	2F5	2F6	2F7	2F8	2F9
58_	28A	28B	2AA	2AB	2CA	2CB	2EA	2EB	2CE	2CF
59_	29A	29B	2BA	2BB	2DA	2DB	2FA	2FB	2DE	2DF
60_	300	301	302	303	304	305	306	307	308	309
61_	310	311	312	313	314	315	316	317	318	319
62_	320	321	322	323	324	325	326	327	328	329
63_	330	331	332	333	334	335	336	337	338	339
64_	340	341	342	343	344	345	346	347	348	349
65_	350	351	352	353	354	355	356	357	358	359
66_	360	361	362	363	364	365	366	367	368	369
67_	370	371	372	373	374	375	376	377	378	379
68_	30A	30B	32A	32B	34A	34B	36A	36B	34E	34F
69_	31A	31B	33A	33B	35A	35B	37A	37B	35E	35F
70_	380	381	382	383	384	385	386	387	388	389
71_	390	391	392	393	394	395	396	397	398	399
72_	3A0	3A1	3A2	3A3	3A4	3A5	3A6	3A7	3A8	3A9
73_	3B0	3B1	3B2	3B3	3B4	3B5	3B6	3B7	3B8	3B9
74_	3C0	3C1	3C2	3C3	3C4	3C5	3C6	3C7	3C8	3C9
75_	3D0	3D1	3D2	3D3	3D4	3D5	3D6	3D7	3D8	3D9
76_	3E0	3E1	3E2	3E3	3E4	3E5	3E6	3E7	3E8	3E9
77_	3F0	3F1	3F2	3F3	3F4	3F5	3F6	3F7	3F8	3F9
78_	38A	38B	3AA	3AB	3CA	3CB	3EA	3EB	3CE	3CF
79_	39A	39B	3BA	3BB	3DA	3DB	3FA	3FB	3DE	3DF
80_	00C	00D	10C	10D	20C	20D	30C	30D	02E	02F
81_	01C	01D	11C	11D	21C	21D	31C	31D	03E	03F
82_	02C	02D	12C	12D	22C	22D	32C	32D	12E	12F
83_	03C	03D	13C	13D	23C	23D	33C	33D	13E	13F
84_	04C	04D	14C	14D	24C	24D	34C	34D	22E	22F
85_	05C	05D	15C	15D	25C	25D	35C	35D	23E	23F
86_	06C	06D	16C	16D	26C	26D	36C	36D	32E	32F
87_	07C	07D	17C	17D	27C	27D	37C	37D	33E	33F
88_	00E	00F	10E	10F	20E	20F	30E	30F	06E	06F
89_	01E	01F	11E	11F	21E	21F	31E	31F	07E	07F
90_	08C	08D	18C	18D	28C	28D	38C	38D	0AE	0AF
91_	09C	09D	19C	19D	29C	29D	39C	39D	0BE	0BF
92_	0AC	0AD	1AC	1AD	2AC	2AD	3AC	3AD	1AE	1AF
93_	0BC	0BD	1BC	1BD	2BC	2BD	3BC	3BD	1BE	1BF
94_	0CC	0CD	1CC	1CD	2CC	2CD	3CC	3CD	2AE	2AF
95_	0DC	0DD	1DC	1DD	2DC	2DD	3DC	3DD	2BE	2BF
96_	0EC	0ED	1EC	1ED	2EC	2ED	3EC	3ED	3AE	3AF
97_	0FC	0FD	1FC	1FD	2FC	2FD	3FC	3FD	3BE	3BF
98_	08E	08F	18E	18F	28E	28F	38E	38F	0EE	0EF
99_	09E	09F	19E	19F	29E	29F	39E	39F	0FE	0FF

	Table 124: DPD-to-BCD translation															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
00_	000	001	002	003	004	005	006	007	008	009	080	081	800	801	880	881
01_	010	011	012	013	014	015	016	017	018	019	090	091	810	811	890	891
02_	020	021	022	023	024	025	026	027	028	029	082	083	820	821	808	809
03_	030	031	032	033	034	035	036	037	038	039	092	093	830	831	818	819
04_	040	041	042	043	044	045	046	047	048	049	084	085	840	841	088	089
05_	050	051	052	053	054	055	056	057	058	059	094	095	850	851	098	099
06_	060	061	062	063	064	065	066	067	068	069	086	087	860	861	888	889
07_	070	071	072	073	074	075	076	077	078	079	096	097	870	871	898	899
08_	100	101	102	103	104	105	106	107	108	109	180	181	900	901	980	981
09_	110	111	112	113	114	115	116	117	118	119	190	191	910	911	990	991
0A_	120	121	122	123	124	125	126	127	128	129	182	183	920	921	908	909
0B_	130	131	132	133	134	135	136	137	138	139	192	193	930	931	918	919
0C_	140	141	142	143	144	145	146	147	148	149	184	185	940	941	188	189
0D_	150	151	152	153	154	155	156	157	158	159	194	195	950	951	198	199
0E_	160	161	162	163	164	165	166	167	168	169	186	187	960	961	988	989
0F_	170	171	172	173	174	175	176	177	178	179	196	197	970	971	998	999
10_	200	201	202	203	204	205	206	207	208	209	280	281	802	803	882	883
11_	210	211	212	213	214	215	216	217	218	219	290	291	812	813	892	893
12_	220	221	222	223	224	225	226	227	228	229	282	283	822	823	828	829
13_	230	231	232	233	234	235	236	237	238	239	292	293	832	833	838	839
14_	240	241	242	243	244	245	246	247	248	249	284	285	842	843	288	289
15_	250	251	252	253	254	255	256	257	258	259	294	295	852	853	298	299
16_	260	261	262	263	264	265	266	267	268	269	286	287	862	863	(888)	(889)
17_	270	271	272	273	274	275	276	277	278	279	296	297	872	873	(898)	(899)
18_	300	301	302	303	304	305	306	307	308	309	380	381	902	903	982	983
19_	310	311	312	313	314	315	316	317	318	319	390	391	912	913	992	993
1A_	320	321	322	323	324	325	326	327	328	329	382	383	922	923	928	929
1B_	330	331	332	333	334	335	336	337	338	339	392	393	932	933	938	939
1C_	340	341	342	343	344	345	346	347	348	349	384	385	942	943	388	389
1D_	350	351	352	353	354	355	356	357	358	359	394	395	952	953	398	399
1E_	360	361	362	363	364	365	366	367	368	369	386	387	962	963	(988)	(989)
1F_	370	371	372	373	374	375	376	377	378	379	396	397	972	973	(998)	(999)
20_	400	401	402	403	404	405	406	407	408	409	480	481	804	805	884	885
21_	410	411	412	413	414	415	416	417	418	419	490	491	814	815	894	895
22_	420	421	422	423	424	425	426	427	428	429	482	483	824	825	848	849
23_	430	431	432	433	434	435	436	437	438	439	492	493	834	835	858	859
24_	440	441	442	443	444	445	446	447	448	449	484	485	844	845	488	489
25_	450	451	452	453	454	455	456	457	458	459	494	495	854	855	498	499
26_	460	461	462	463	464	465	466	467	468	469	486	487	864	865	(888)	(889)
27_	470	471	472	473	474	475	476	477	478	479	496	497	874	875	(898)	(899)
28_	500	501	502	503	504	505	506	507	508	509	580	581	904	905	984	985
29_	510	511	512	513	514	515	516	517	518	519	590	591	914	915	994	995
2A_	520	521	522	523	524	525	526	527	528	529	582	583	924	925	948	949
2B_	530	531	532	533	534	535	536	537	538	539	592	593	934	935	958	959
2C_	540	541	542	543	544	545	546	547	548	549	584	585	944	945	588	589
2D_	550	551	552	553	554	555	556	557	558	559	594	595	954	955	598	599
2E_	560	561	562	563	564	565	566	567	568	569	586	587	964	965	(988)	(989)
2F_	570	571	572	573	574	575	576	577	578	579	596	597	974	975	(998)	(999)
30_	600	601	602	603	604	605	606	607	608	609	680	681	806	807	886	887
31_	610	611	612	613	614	615	616	617	618	619	690	691	816	817	896	897
32_	620	621	622	623	624	625	626	627	628	629	682	683	826	827	868	869
33_	630	631	632	633	634	635	636	637	638	639	692	693	836	837	878	879
34_	640	641	642	643	644	645	646	647	648	649	684	685	846	847	688	689
35_	650	651	652	653	654	655	656	657	658	659	694	695	856	857	698	699
36_	660	661	662	663	664	665	666	667	668	669	686	687	866	867	(888)	(889)
37_	670	671	672	673	674	675	676	677	678	679	696	697	876	877	(898)	(899)
38_	700	701	702	703	704	705	706	707	708	709	780	781	906	907	986	987
39_	710	711	712	713	714	715	716	717	718	719	790	791	916	917	996	997
3A_	720	721	722	723	724	725	726	727	728	729	782	783	926	927	968	969
3B_	730	731	732	733	734	735	736	737	738	739	792	793	936	937	978	979
3C_	740	741	742	743	744	745	746	747	748	749	784	785	946	947	788	789
3D_	750	751	752	753	754	755	756	757	758	759	794	795	956	957	798	799
3E_	760	761	762	763	764	765	766	767	768	769	786	787	966	967	(988)	(989)
3F_	770	771	772	773	774	775	776	777	778	779	796	797	976	977	(998)	(999)

Appendix C. Vector RTL Functions [Category: Vector]

```
ConvertSPtoSXWsaturate( X, Y )
   sign    = X0
   exp0:7  = X1:8
   frac0:30 = X9:31 || 0b0000_0000
   if((exp==255)&(frac!=0)) then return(0x0000_0000) // NaN operand
   if((exp==255)&(frac==0)) then do // infinity operand
          VSCRSAT = 1
          return( (sign==1) ? 0x8000_0000 : 0x7FFF_FFFF )
   if((exp+Y-127)>30) then do // large operand
          VSCRSAT = 1
          return( (sign==1) ? 0x8000_0000 : 0x7FFF_FFFF )
   if((exp+Y-127)<0) then return(0x0000_0000) // -1.0 < value < 1.0 (value rounds to 0)
   significand0:31 = 0b1 || frac
   do i=1 to 31-(exp+Y-127)
          significand = significand >>ui 1
   return( (sign==0) ? significand : (¬significand + 1) )

ConvertSPtoUXWsaturate( X, Y )
   sign    = X0
   exp0:7  = X1:8
   frac0:30 = X9:31 || 0b0000_0000
   if((exp==255)&&(frac!=0)) then return(0x0000_0000) // NaN operand
   if((exp==255)&&(frac==0)) then do   // infinity operand
      VSCRSAT = 1
      return( (sign==1) ? 0x0000_0000 : 0xFFFF_FFFF )
   if((exp+Y-127)>31) then do // large operand
      VSCRSAT = 1
      return( (sign==1) ? 0x0000_0000 : 0xFFFF_FFFF )
   if((exp+Y-127)<0) then return(0x0000_0000) // -1.0 < value < 1.0
          //   value rounds to 0
   if( sign==1 ) then do // negative operand
      VSCRSAT = 1
      return(0x0000_0000)
   significand0:31 = 0b1 || frac
   do i=1 to 31-(exp+Y-127)
      significand = significand >>ui 1
   return( significand )

ConvertSXWtoSP( X )
   sign    = X0
   exp0:7  = 32 + 127
   frac0:32 = X0 || X0:31
   if( frac==0 ) return( 0x0000_0000 ) // Zero operand
   if( sign==1 ) then frac = ¬frac + 1
   do while( frac0==0 )
      frac = frac << 1
      exp = exp - 1
   lsb = frac23
   gbit = frac24
   xbit = frac25:32!=0
   inc = ( lsb && gbit ) | ( gbit && xbit )
   frac0:23 = frac0:23 + inc
   if( carry_out==1 ) exp = exp + 1
   return( sign || exp || frac1:23 )
```

```
ConvertUXWtoSP( X )
   exp0:7 = 31 + 127
   frac0:31 = X0:31
   if( frac==0 ) return( 0x0000_0000 ) // Zero Operand
   do while( frac0==0 )
      frac = frac << 1
      exp = exp - 1
   lsb = frac23
   gbit = frac24
   xbit = frac25:31!=0
   inc = ( lsb && gbit ) | ( gbit && xbit )
   frac0:23 = frac0:23 + inc
   if( carry_out==1 ) exp = exp + 1
   return( 0b0 || exp || frac1:23 )
```

Appendix D. Embedded Floating-Point RTL Functions

[Category: SPE.Embedded Float Scalar Double] [Category: SPE.Embedded Float Scalar Single] [Category: SPE.Embedded Float Vector]

D.1 Common Functions

```
// Check if 32-bit fp value is a NaN or Infinity
Isa32NaNorInfinity(fp)
return (fp_exp = 255)

Isa32NaN(fp)
return ((fp_exp = 255) & (fp_frac ≠ 0))

// Check if 32-bit fp value is denormalized
Isa32Denorm(fp)
return ((fp_exp = 0) & (fp_frac ≠ 0))

// Check if 64-bit fp value is a NaN or Infinity
Isa64NaNorInfinity(fp)
return (fp_exp = 2047)

Isa64NaN(fp)
return ((fp_exp = 2047) & (fp_frac ≠ 0))

// Check if 32-bit fp value is denormalized
Isa64Denorm(fp)
return ((fp_exp = 0) & (fp_frac ≠ 0))

// Signal an error in the SPEFSCR
SignalFPError(upper_lower, bits)
if (upper_lower = HI) then
   bits ← bits << 15
SPEFSCR ← SPEFSCR | bits
bits ← (FG | FX)
if (upper_lower = HI) then
   bits ← bits << 15
SPEFSCR ← SPEFSCR & ¬bits
```

```
// Round a 32-bit fp result
Round32(fp, guard, sticky)

FP32format fp;
if (SPEFSCR_FINXE = 0) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | fp_frac[22]) then
            v_0:23 ← fp_frac + 1
            if v_0 then
               if (fp_exp >= 254) then
                  // overflow
                  fp ← fp_sign || 0b11111110 || 23^1
               else
                  fp_exp ← fp_exp + 1
                  fp_frac ← v_1:23
            else
               fp_frac ← v_1:23
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
return fp

// Round a 64-bit fp result
Round64(fp, guard, sticky)

FP32format fp;
if (SPEFSCR_FINXE = 0) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | fp_frac[51]) then
            v_0:52 ← fp_frac + 1
            if v_0 then
               if (fp_exp >= 2046) then
                  // overflow
                  fp ← fp_sign ||
                       0b11111111110 || 52^1
               else
                  fp_exp ← fp_exp + 1
                  fp_frac ← v_1:52
            else
               fp_frac ← v_1:52
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
return fp
```

D.2 Convert from Single-Precision Embedded Floating-Point to Integer Word with Saturation

```
// Convert 32-bit Floating-Point to 32-bit integer
// or fractional
//   signed = S (signed) or U (unsigned)
//   upper_lower = HI (high word) or LO (low word)
//   round = RND (round) or ZER (truncate)
//   fractional = F (fractional) or I (integer)
```

CnvtFP32ToI32Sat(fp, signed, upper_lower, round, fractional)

```
FP32format fp;
if (Isa32NaNorInfinity(fp)) then
   SignalFPError(upper_lower, FINV)
   if (Isa32NaN(fp)) then
      return 0x00000000    // all NaNs
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      if (fp_sign = 1) then
         return 0x00000000
      else
         return 0xffffffff
if (Isa32Denorm(fp)) then
   SignalFPError(upper_lower, FINV)
   return 0x00000000    // regardless of sign
if ((signed = U) & (fp_sign = 1)) then
   SignalFPError(upper_lower, FOVF)  // overflow
   return 0x00000000
if ((fp_exp = 0) & (fp_frac = 0)) then
   return 0x00000000    // all zero values
if (fractional = I) then  // convert to integer
   max_exp ← 158
   shift ← 158 - fp_exp
   if (signed = S) then
      if ((fp_exp≠158)|(fp_frac≠0)|(fp_sign≠1)) then
         max_exp ← max_exp - 1
else  // fractional conversion
   max_exp ← 126
   shift ← 126 - fp_exp
   if (signed = S) then
      shift ← shift + 1
if (fp_exp > max_exp) then
   SignalFPError(upper_lower, FOVF)  // overflow
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      return 0xffffffff

result ← 0b1 || fp_frac || 0b00000000  // add U bit
guard ← 0
sticky ← 0
for (n ← 0; n < shift; n ← n + 1) do
   sticky ← sticky | guard
   guard ← result & 0x00000001
   result ← result > 1
// Report sticky and guard bits
if (upper_lower = HI) then
   SPEFSCR_FGH ← guard
   SPEFSCR_FXH ← sticky
else
   SPEFSCR_FG ← guard
   SPEFSCR_FX ← sticky

if (guard | sticky) then
   SPEFSCR_FINXS ← 1
// Round the integer result
if ((round = RND) & (SPEFSCR_FINXE = 0)) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | (result & 0x00000001)) then
            result ← result + 1
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
if (signed = S) then
   if (fp_sign = 1) then
      result ← ¬result + 1
return result
```

D.3 Convert from Double-Precision Embedded Floating-Point to Integer Word with Saturation

```
// Convert 64-bit Floating-Point to 32-bit integer
// or fractional
//   signed = S (signed) or U (unsigned)
//   round = RND (round) or ZER (truncate)
//   fractional = F (fractional) or I (integer)

CnvtFP64ToI32Sat(fp, signed, round,
fractional)
FP64format fp;

if (Isa64NaNorInfinity(fp)) then
   SignalFPError(LO, FINV)
   if (Isa64NaN(fp)) then
      return 0x00000000    // all NaNs
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      if (fp_sign = 1) then
         return 0x00000000
      else
         return 0xffffffff

if (Isa64Denorm(fp)) then
   SignalFPError(LO, FINV)
   return 0x00000000    // regardless of sign
if ((signed = U) & (fp_sign = 1)) then
   SignalFPError(LO, FOVF)  // overflow
   return 0x00000000
if ((fp_exp = 0) & (fp_frac = 0)) then
   return 0x00000000    // all zero values
if (fractional = I) then  // convert to integer
   max_exp ← 1054
   shift ← 1054 - fp_exp
   if (signed ← S) then
      if ((fp_exp≠1054)|(fp_frac≠0)|(fp_sign≠1)) then
         max_exp ← max_exp - 1
else                        // fractional conversion
   max_exp ← 1022
   shift ← 1022 - fp_exp
   if (signed = S) then
      shift ← shift + 1

if (fp_exp > max_exp) then
   SignalFPError(LO, FOVF)  // overflow
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000
      else
         return 0x7fffffff
   else
      return 0xffffffff

result ← 0b1 || fp_frac[0:30]  // add U to frac
guard ← fp_frac[31]
sticky ← (fp_frac[32:63] ≠ 0)
for (n ← 0; n < shift; n ← n + 1) do
   sticky ← sticky | guard
   guard ← result & 0x00000001
   result ← result > 1
// Report sticky and guard bits

SPEFSCR_FG ← guard
SPEFSCR_FX ← sticky

if (guard | sticky) then
   SPEFSCR_FINXS ← 1
// Round the result
if ((round = RND) & (SPEFSCR_FINXE = 0)) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | (result & 0x00000001)) then
            result ← result + 1
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
if (signed = S) then
   if (fp_sign = 1) then
      result ← ¬result + 1
return result
```

D.4 Convert from Double-Precision Embedded Floating-Point to Integer Doubleword with Saturation

```
// Convert 64-bit Floating-Point to 64-bit integer
//   signed = S (signed) or U (unsigned)
//   round = RND (round) or ZER (truncate)

CnvtFP64ToI64Sat(fp, signed, round)
FP64format fp;
if (Isa64NaNorInfinity(fp)) then
   SignalFPError(LO, FINV)
   if (Isa64NaN(fp)) then
      return 0x00000000_00000000    // all NaNs
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000_00000000
      else
         return 0x7fffffff_ffffffff
   else
      if (fp_sign = 1) then
         return 0x00000000_00000000
      else
         return 0xffffffff_ffffffff

if (Isa64Denorm(fp)) then
   SignalFPError(LO, FINV)
   return 0x00000000_00000000

if ((signed = U) & (fp_sign = 1)) then
   SignalFPError(LO, FOVF)  // overflow
   return 0x00000000_00000000
if ((fp_exp = 0) & (fp_frac = 0)) then
   return 0x00000000_00000000    // all zero values

max_exp ← 1086
shift ← 1086 - fp_exp
if (signed = S) then
   if ((fp_exp≠1086)|(fp_frac≠0)|(fp_sign≠1)) then
      max_exp ← max_exp - 1

if (fp_exp > max_exp) then
   SignalFPError(LO, FOVF)  // overflow
   if (signed = S) then
      if (fp_sign = 1) then
         return 0x80000000_00000000
      else
         return 0x7fffffff_ffffffff
   else
      return 0xffffffff_ffffffff

result ← 0b1 || fp_frac || 0b00000000000 //add U bit
guard ← 0
sticky ← 0
for (n ← 0; n < shift; n ← n + 1) do
   sticky ← sticky | guard
   guard ← result & 0x00000000_00000001
   result ← result > 1
// Report sticky and guard bits
SPEFSCR_FG ← guard
SPEFSCR_FX ← sticky
if (guard | sticky) then
   SPEFSCR_FINXS ← 1
// Round the result
if ((round = RND) & (SPEFSCR_FINXE = 0)) then
   if (SPEFSCR_FRMC = 0b00) then    // nearest
      if (guard) then
         if (sticky | (result&0x00000000_00000001))
         then
            result ← result + 1
   else if ((SPEFSCR_FRMC & 0b10) = 0b10) then
      // infinity modes
      // implementation dependent
if (signed = S) then
   if (fp_sign = 1) then
      result ← ¬result + 1
return result
```

D.5 Convert to Single-Precision Embedded Floating-Point from Integer Word

```
// Convert from 32-bit integer or fractional to
// 32-bit Floating-Point
//   signed = S (signed) or U (unsigned)
//   round = RND (round) or ZER (truncate)
//   fractional = F (fractional) or I (integer)
CnvtI32ToFP32(v, signed, upper_lower,
fractional)
FP32format result;
result_sign ← 0
if (v = 0) then
   result ← 0
   if (upper_lower = HI) then
      SPEFSCR_FGH ← 0
      SPEFSCR_FXH ← 0
   else
      SPEFSCR_FG ← 0
      SPEFSCR_FX ← 0
else
   if (signed = S) then
      if (v_0 = 1) then
         v ← ¬v + 1
         result_sign ← 1
   if (fractional = F) then    // frac bit align
      maxexp ← 127
      if (signed = U) then
         maxexp ← maxexp - 1
   else
      maxexp ← 158    // integer bit alignment
   sc ← 0
   while (v_0 = 0)
      v ← v << 1
      sc ← sc + 1
   v_0 ← 0    // clear U bit
   result_exp ← maxexp - sc
   guard ← v_24
   sticky ← (v_25:31 ≠ 0)

   // Report sticky and guard bits
   if (upper_lower = HI) then
      SPEFSCR_FGH ← guard
      SPEFSCR_FXH ← sticky
   else
      SPEFSCR_FG ← guard
      SPEFSCR_FX ← sticky

   if (guard | sticky) then
      SPEFSCR_FINXS ← 1
// Round the result

   result_frac ← v_1:23
   result ← Round32(result, guard, sticky)
return result
```

D.6 Convert to Double-Precision Embedded Floating-Point from Integer Word

```
// Convert from integer or fractional to 64 bit
// Floating-Point
//   signed = S (signed) or U (unsigned)
//   fractional = F (fractional) or I (integer)
CnvtI32ToFP64(v, signed, fractional)
FP64format result;
result_sign ← 0
if (v = 0) then
   result ← 0
   SPEFSCR_FG ← 0
   SPEFSCR_FX ← 0
else
   if (signed = S) then
      if (v_0 = 1) then
         v ← ¬v + 1
         result_sign ← 1
   if (fractional = F) then    // frac bit align
      maxexp ← 1023
      if (signed = U) then
         maxexp ← maxexp - 1
   else
      maxexp ← 1054    // integer bit align
   sc ← 0
   while (v_0 = 0)
      v ← v << 1
      sc ← sc + 1
   v_0 ← 0    // clear U bit
   result_exp ← maxexp - sc

// Report sticky and guard bits

   SPEFSCR_FG ← 0
   SPEFSCR_FX ← 0

   result_frac ← v_1:31 || 21^0
return result
```

D.7 Convert to Double-Precision Embedded Floating-Point from Integer Doubleword

```
// Convert from 64-bit integer to 64-bit
// floating-point
//   signed = S (signed) or U (unsigned)
CnvtI64ToFP64(v, signed)
FP64format result;
result_sign ← 0
if (v = 0) then
   result ← 0
   SPEFSCR_FG ← 0
   SPEFSCR_FX ← 0
else
   if (signed = S) then
      if (v_0 = 1) then
         v ← ¬v + 1
         result_sign ← 1
   maxexp ← 1054
   sc ← 0
   while (v_0 = 0)
      v ← v << 1
      sc ← sc + 1
   v_0 ← 0    // clear U bit
   result_exp ← maxexp - sc
   guard ← v_53
   sticky ← (v_54:63 ≠ 0)

// Report sticky and guard bits

   SPEFSCR_FG ← guard
   SPEFSCR_FX ← sticky
   if (guard | sticky) then
      SPEFSCR_FINXS ← 1
// Round the result

   result_frac ← v_1:52
   result ← Round64(result, guard, sticky)

return result
```

Appendix E. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided that defines simple shorthand for the most frequently used forms of *Branch Conditional*, *Compare*, *Trap*, *Rotate and Shift*, and certain other instructions.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

E.1 Symbols

The following symbols are defined for use in instructions (basic or extended mnemonics) that specify a Condition Register field or a Condition Register bit. The first five (lt, ..., un) identify a bit number within a CR field. The remainder (cr0, ..., cr7) identify a CR field. An expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol and 32 can be used to identify a CR bit.

Symbol	Value	Meaning
lt	0	Less than
gt	1	Greater than
eq	2	Equal
so	3	Summary overflow
un	3	Unordered (after floating-point comparison)
cr0	0	CR Field 0
cr1	1	CR Field 1
cr2	2	CR Field 2
cr3	3	CR Field 3
cr4	4	CR Field 4
cr5	5	CR Field 5
cr6	6	CR Field 6
cr7	7	CR Field 7

The extended mnemonics in Sections E.2.2 and E.3 require identification of a CR bit: if one of the CR field symbols is used, it must be multiplied by 4 and added to a bit-number-within-CR-field (value in the range 0-3, explicit or symbolic) and 32. The extended mnemonics in Sections E.2.3 and E.5 require identification of a CR field: if one of the CR field symbols is used, it must *not* be multiplied by 4 or added to 32. (For the extended mnemonics in Section E.2.3, the bit number within the CR field is part of the extended mnemonic. The programmer identifies the CR field, and the Assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)

E.2 Branch Mnemonics

The mnemonics discussed in this section are variations of the *Branch Conditional* instructions.

Note: ***bclr***, ***bclrl***, ***bcctr***, and ***bcctrl*** each serve as both a basic and an extended mnemonic. The Assembler will recognize a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with three operands as the basic form, and a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00. Similarly, for all the extended mnemonics described in Sections E.2.2 - E.2.4 that devolve to any of these four basic mnemonics the BH operand can either be coded or omitted. If it is omitted it is assumed to be 0b00.

E.2.1 BO and BI Fields

The 5-bit BO and BI fields control whether the branch is taken. Providing an extended mnemonic for every possible combination of these fields would be neither useful nor practical. The mnemonics described in Sections E.2.2 - E.2.4 include the most useful cases. Other cases can be coded using a basic *Branch Conditional* mnemonic (***bc***[***l***][***a***], ***bclr***[***l***], ***bcctr***[***l***]) with the appropriate operands.

E.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table 125 that tests a Condition Register bit specify the corresponding bit as the first operand. The symbols defined in Section E.1 can be used in this operand.

Notice that there are no extended mnemonics for relative and absolute unconditional branches. For these the basic mnemonics ***b***, ***ba***, ***bl***, and ***bla*** should be used.

Table 125:Simple branch mnemonics

Branch Semantics	LR not Set				LR Set			
	bc **Relative**	***bca*** **Absolute**	***bclr*** **To LR**	***bcctr*** **To CTR**	***bcl*** **Relative**	***bcla*** **Absolute**	***bclrl*** **To LR**	***bcctrl*** **To CTR**
Branch unconditionally	-	-	blr	bctr	-	-	blrl	bctrl
Branch if CR_{BI}=1	bt	bta	btlr	btctr	btl	btla	btlrl	btctrl
Branch if CR_{BI}=0	bf	bfa	bflr	bfctr	bfl	bfla	bflrl	bfctrl
Decrement CTR, branch if CTR nonzero	bdnz	bdnza	bdnzlr	-	bdnzl	bdnzla	bdnzlrl	-
Decrement CTR, branch if CTR nonzero and CR_{BI}=1	bdnzt	bdnzta	bdnztlr	-	bdnztl	bdnztla	bdnztlrl	-
Decrement CTR, branch if CTR nonzero and CR_{BI}=0	bdnzf	bdnzfa	bdnzflr	-	bdnzfl	bdnzfla	bdnzflrl	-
Decrement CTR, branch if CTR zero	bdz	bdza	bdzlr	-	bdzl	bdzla	bdzlrl	-
Decrement CTR, branch if CTR zero and CR_{BI}=1	bdzt	bdzta	bdztlr	-	bdztl	bdztla	bdztlrl	-
Decrement CTR, branch if CTR zero and CR_{BI}=0	bdzf	bdzfa	bdzflr	-	bdzfl	bdzfla	bdzflrl	-

Examples

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).

   bdnz target (equivalent to: bc 16,0,target)

2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is “equal”.

   bdnzt eq,target (equivalent to: bc 8,2,target)

3. Same as (2), but “equal” condition is in CR5.

   bdnzt 4×cr5+eq,target (equivalent to: bc 8,22,target)

4. Branch if bit 59 of CR is 0.

   bf 27,target (equivalent to: bc 4,27,target)

5. Same as (4), but set the Link Register. This is a form of conditional “call”.

   bfl 27,target (equivalent to: bcl 4,27,target)

E.2.3 Branch Mnemonics Incorporating Conditions

In the mnemonics defined in Table 126, the test of a bit in a Condition Register field is encoded in the mnemonic.

Instructions using the mnemonics in Table 126 specify the CR field as an optional first operand. One of the CR field symbols defined in Section E.1 can be used for this operand. If the CR field being tested is CR Field 0, this operand need not be specified unless the resulting basic mnemonic is ***bclr***[***l***] or ***bcctr***[***l***] and the BH operand is specified.

A standard set of codes has been adopted for the most common combinations of branch conditions.

Code	Meaning
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow
un	Unordered (after floating-point comparison)
nu	Not unordered (after floating-point comparison)

These codes are reflected in the mnemonics shown in Table 126.

Table 126:Branch mnemonics incorporating conditions

Branch Semantics	LR not Set				LR Set			
	bc **Relative**	***bca*** **Absolute**	***bclr*** **To LR**	***bcctr*** **To CTR**	***bcl*** **Relative**	***bcla*** **Absolute**	***bclrl*** **To LR**	***bcctrl*** **To CTR**
Branch if less than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if less than or equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if greater than or equal	bge	bgea	bgelr	bgectr	bgel	bgela	bgelrl	bgectrl
Branch if greater than	bgt	bgta	bgtlr	bgtctr	bgtl	bgtla	bgtlrl	bgtctrl
Branch if not less than	bnl	bnla	bnllr	bnlctr	bnll	bnlla	bnllrl	bnlctrl
Branch if not equal	bne	bnea	bnelr	bnectr	bnel	bnela	bnelrl	bnectrl
Branch if not greater than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if summary overflow	bso	bsoa	bsolr	bsoctr	bsol	bsola	bsolrl	bsoctrl
Branch if not summary overflow	bns	bnsa	bnslr	bnsctr	bnsl	bnsla	bnslrl	bnsctrl
Branch if unordered	bun	buna	bunlr	bunctr	bunl	bunla	bunlrl	bunctrl
Branch if not unordered	bnu	bnua	bnulr	bnuctr	bnul	bnula	bnulrl	bnuctrl

Examples

1. Branch if CR0 reflects condition “not equal”.

   bne target (equivalent to: bc 4,2,target)

2. Same as (1), but condition is in CR3.

bne cr3,target (equivalent to: bc 4,14,target)

3. Branch to an absolute target if CR4 specifies “greater than”, setting the Link Register. This is a form of conditional “call”.

   bgtla cr4,target (equivalent to: bcla 12,17,target)

4. Same as (3), but target address is in the Count Register.

   bgtctrl cr4 (equivalent to: bcctrl 12,17,0)

E.2.4 Branch Prediction

Software can use the “at” bits of *Branch Conditional* instructions to provide a hint to the processor about the behavior of the branch. If, for a given such instruction, the branch is almost always taken or almost always not taken, a suffix can be added to the mnemonic indicating the value to be used for the “at” bits.

\+ Predict branch to be taken (at=0b11)

\- Predict branch not to be taken (at=0b10)

Such a suffix can be added to any *Branch Conditional* mnemonic, either basic or extended, that tests either the Count Register or a CR bit (but not both). Assemblers should use 0b00 as the default value for the “at” bits, indicating that software has offered no prediction.

Examples

1. Branch if CR0 reflects condition “less than”, specifying that the branch should be predicted to be taken.

   blt+ target

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.

   bltlr-

E.3 Condition Register Logical Mnemonics

The *Condition Register Logical* instructions can be used to set (to 1), clear (to 0), copy, or invert a given Condition Register bit. Extended mnemonics are provided that allow these operations to be coded easily.

Table 127:Condition Register logical mnemonics

Operation	Extended Mnemonic	Equivalent to
Condition Register set	crset bx	creqv bx,bx,bx
Condition Register clear	crclr bx	crxor bx,bx,bx
Condition Register move	crmove bx,by	cror bx,by,by
Condition Register not	crnot bx,by	crnor bx,by,by

The symbols defined in Section E.1 can be used to identify the Condition Register bits.

Examples

1. Set CR bit 57.

   crset 25 (equivalent to: creqv 25,25,25)

2. Clear the SO bit of CR0.

   crclr so (equivalent to: crxor 3,3,3)

3. Same as (2), but SO bit to be cleared is in CR3.

   crclr 4×cr3+so (equivalent to: crxor 15,15,15)

4. Invert the EQ bit.

   crnot eq,eq (equivalent to: crnor 2,2,2)

5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.

   crnot 4×cr5+eq,4×cr4+eq (equivalent to: crnor 22,18,18)

E.4 Subtract Mnemonics

E.4.1 Subtract Immediate

Although there is no “Subtract Immediate” instruction, its effect can be achieved by using an Add Immediate instruction with the immediate operand negated. Extended mnemonics are provided that include this negation, making the intent of the computation clearer.

```
subi    Rx,Ry,value   (equivalent to:  addi    Rx,Ry,-value)
subis   Rx,Ry,value   (equivalent to:  addis   Rx,Ry,-value)
subic   Rx,Ry,value   (equivalent to:  addic   Rx,Ry,-value)
subic.  Rx,Ry,value   (equivalent to:  addic.  Rx,Ry,-value)
```

E.4.2 Subtract

The *Subtract From* instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more “normal” order, in which the third operand is subtracted from the second. Both these mnemonics can be coded with a final “o” and/or “.” to cause the OE and/or Rc bit to be set in the underlying instruction.

```
sub     Rx,Ry,Rz      (equivalent to:  subf    Rx,Rz,Ry)
subc    Rx,Ry,Rz      (equivalent to:  subfc   Rx,Rz,Ry)
```

E.5 Compare Mnemonics

The L field in the fixed-point *Compare* instructions controls whether the operands are treated as 64-bit quantities or as 32-bit quantities. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

The BF field can be omitted if the result of the comparison is to be placed into CR Field 0. Otherwise the target CR field must be specified as the first operand. One of the CR field symbols defined in Section E.1 can be used for this operand.

Note: The basic *Compare* mnemonics of Power ISA are the same as those of POWER, but the POWER instructions have three operands while the Power ISA instructions have four. The Assembler will recognize a basic *Compare* mnemonic with three operands as the POWER form, and will generate the instruction with L=0. (Thus the Assembler must require that the BF field, which normally can be omitted when CR Field 0 is the target, be specified explicitly if L is.)

E.5.1 Doubleword Comparisons

Table 128:Doubleword compare mnemonics

Operation	Extended Mnemonic	Equivalent to
Compare doubleword immediate	cmpdi bf,ra,si	cmpi bf,1,ra,si
Compare doubleword	cmpd bf,ra,rb	cmp bf,1,ra,rb
Compare logical doubleword immediate	cmpldi bf,ra,ui	cmpli bf,1,ra,ui
Compare logical doubleword	cmpld bf,ra,rb	cmpl bf,1,ra,rb

Examples

1. Compare register Rx and immediate value 100 as unsigned 64-bit integers and place result into CR0.

   cmpldi Rx,100 (equivalent to: cmpli 0,1,Rx,100)

2. Same as (1), but place result into CR4.

   cmpldi cr4,Rx,100 (equivalent to: cmpli 4,1,Rx,100)

3. Compare registers Rx and Ry as signed 64-bit integers and place result into CR0.

   cmpd Rx,Ry (equivalent to: cmp 0,1,Rx,Ry)

E.5.2 Word Comparisons

Table 129:Word compare mnemonics

Operation	Extended Mnemonic	Equivalent to
Compare word immediate	cmpwi bf,ra,si	cmpi bf,0,ra,si
Compare word	cmpw bf,ra,rb	cmp bf,0,ra,rb
Compare logical word immediate	cmplwi bf,ra,ui	cmpli bf,0,ra,ui
Compare logical word	cmplw bf,ra,rb	cmpl bf,0,ra,rb

Examples

1. Compare bits 32:63 of register Rx and immediate value 100 as signed 32-bit integers and place result into CR0.

   cmpwi Rx,100 (equivalent to: cmpi 0,0,Rx,100)

2. Same as (1), but place result into CR4.

   cmpwi cr4,Rx,100 (equivalent to: cmpi 4,0,Rx,100)

3. Compare bits 32:63 of registers Rx and Ry as unsigned 32-bit integers and place result into CR0.

   cmplw Rx,Ry (equivalent to: cmpl 0,0,Rx,Ry)

E.6 Trap Mnemonics

The mnemonics defined in Table 130 are variations of the *Trap* instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the most common combinations of trap conditions.

Code	Meaning	TO encoding	<	>	=	$<^u$	$>^u$
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
u	Unconditionally with parameters	31	1	1	1	1	1
(none)	Unconditional	31	1	1	1	1	1

These codes are reflected in the mnemonics shown in Table 130.

Table 130:Trap mnemonics

Trap Semantics	64-bit Comparison		32-bit Comparison	
	tdi **Immediate**	***td*** **Register**	***twi*** **Immediate**	***tw*** **Register**
Trap unconditionally	-	-	-	trap
Trap unconditionally with parameters	tdui	tdu	twui	twu
Trap if less than	tdlti	tdlt	twlti	twlt
Trap if less than or equal	tdlei	tdle	twlei	twle
Trap if equal	tdeqi	tdeq	tweqi	tweq
Trap if greater than or equal	tdgei	tdge	twgei	twge
Trap if greater than	tdgti	tdgt	twgti	twgt
Trap if not less than	tdnli	tdnl	twnli	twnl
Trap if not equal	tdnei	tdne	twnei	twne
Trap if not greater than	tdngi	tdng	twngi	twng
Trap if logically less than	tdllti	tdllt	twllti	twllt
Trap if logically less than or equal	tdllei	tdlle	twllei	twlle
Trap if logically greater than or equal	tdlgei	tdlge	twlgei	twlge
Trap if logically greater than	tdlgti	tdlgt	twlgti	twlgt
Trap if logically not less than	tdlnli	tdlnl	twlnli	twlnl
Trap if logically not greater than	tdlngi	tdlng	twlngi	twlng

Examples

1. Trap if register Rx is not 0.

```
tdnei   Rx,0        (equivalent to:   tdi   24,Rx,0)
```

2. Same as (1), but comparison is to register Ry.

```
tdne    Rx,Ry       (equivalent to:   td    24,Rx,Ry)
```

3. Trap if bits 32:63 of register Rx, considered as a 32-bit quantity, are logically greater than 0x7FF.

```
twlgti  Rx,0x7FF    (equivalent to:   twi   1,Rx,0x7FF)
```

4. Trap unconditionally.

```
trap                (equivalent to:   tw    31,0,0)
```

5. Trap unconditionally with immediate parameters Rx and Ry

```
tdu     Rx,Ry       (equivalent to:   td    31,Rx,Ry)
```

E.7 Integer Select Mnemonics

The mnemonics defined in Table 131, “Integer Select mnemonics,” on page 716 are variations of the *Integer Select* instructions, with the most useful values of BC represented in the mnemonic rather than specified as a numeric operand..

Code	Meaning
lt	Less than
eq	Equal
gt	Greater than

These codes are reflected in the mnemonics shown in Table 131.

Table 131: Integer Select mnemonics

Select semantics	***isel*** **extended mnemonic**
Integer Select if less than	isellt
Integer Select if equal	iseleq
Integer Select if greater than	iselgt

Examples

1. Set register Rx to Ry if the LT bit is set in CR0, and to Rz otherwise.

```
isellt  Rx,Ry,Rz    (equivalent to:   isel  Rx,Ry,Rz,0)
```

2. Set register Rx to Ry if the GT bit is set in CR0, and to Rz otherwise.

```
iselgt  Rx,Ry,Rz    (equivalent to:   isel  Rx,Ry,Rz,1)
```

3. Set register Rx to Ry if the EQ bit is set in CR0, and to Rz otherwise.

```
iseleq  Rx,Ry,Rz    (equivalent to:   isel  Rx,Ry,Rz,2)
```

E.8 Rotate and Shift Mnemonics

The *Rotate and Shift* instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Extended mnemonics are provided that allow some of the simpler operations to be coded easily.

Mnemonics are provided for the following types of operation.

Extract Select a field of n bits starting at bit position b in the source register; left or right justify this field in the target register; clear all other bits of the target register to 0.

Insert Select a left-justified or right-justified field of n bits in the source register; insert this field starting at bit position b of the target register; leave other bits of the target register unchanged. (No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, because such an insertion requires more than one instruction.)

Rotate Rotate the contents of a register right or left n bits without masking.

Shift Shift the contents of a register right or left n bits, clearing vacated bits to 0 (logical shift).

Clear Clear the leftmost or rightmost n bits of a register to 0.

Clear left and shift left
Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

E.8.1 Operations on Doublewords

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

Table 132:Doubleword rotate and shift mnemonics

Operation	Extended Mnemonic	Equivalent to
Extract and left justify immediate	extldi ra,rs,n,b (n > 0)	rldicr ra,rs,b,n-1
Extract and right justify immediate	extrdi ra,rs,n,b (n > 0)	rldicl ra,rs,b+n,64-n
Insert from right immediate	insrdi ra,rs,n,b (n > 0)	rldimi ra,rs,64-(b+n),b
Rotate left immediate	rotldi ra,rs,n	rldicl ra,rs,n,0
Rotate right immediate	rotrdi ra,rs,n	rldicl ra,rs,64-n,0
Rotate left	rotld ra,rs,rb	rldcl ra,rs,rb,0
Shift left immediate	sldi ra,rs,n (n < 64)	rldicr ra,rs,n,63-n
Shift right immediate	srdi ra,rs,n (n < 64)	rldicl ra,rs,64-n,n
Clear left immediate	clrldi ra,rs,n (n < 64)	rldicl ra,rs,0,n
Clear right immediate	clrrdi ra,rs,n (n < 64)	rldicr ra,rs,0,63-n
Clear left and shift left immediate	clrlsldi ra,rs,b,n (n <= b < 64)	rldic ra,rs,n,b-n

Examples

1. Extract the sign bit (bit 0) of register Ry and place the result right-justified into register Rx.

   extrdi Rx,Ry,1,0 (equivalent to: rldicl Rx,Ry,1,63)

2. Insert the bit extracted in (1) into the sign bit (bit 0) of register Rz.

   insrdi Rz,Rx,1,0 (equivalent to: rldimi Rz,Rx,63,0)

3. Shift the contents of register Rx left 8 bits.

   sldi Rx,Rx,8 (equivalent to: rldicr Rx,Rx,8,55)

4. Clear the high-order 32 bits of register Ry and place the result into register Rx.

   clrldi Rx,Ry,32 (equivalent to: rldicl Rx,Ry,0,32)

E.8.2 Operations on Words

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction. The operations as described above apply to the low-order 32 bits of the registers, as if the registers were 32-bit registers. The Insert operations either preserve the high-order 32 bits of the target register or place rotated data there; the other operations clear these bits.

Table 133:Word rotate and shift mnemonics

Operation	**Extended Mnemonic**	**Equivalent to**
Extract and left justify immediate	extlwi ra,rs,n,b (n > 0)	rlwinm ra,rs,b,0,n-1
Extract and right justify immediate	extrwi ra,rs,n,b (n > 0)	rlwinm ra,rs,b+n,32-n,31
Insert from left immediate	inslwi ra,rs,n,b (n > 0)	rlwimi ra,rs,32-b,b,(b+n)-1
Insert from right immediate	insrwi ra,rs,n,b (n > 0)	rlwimi ra,rs,32-(b+n),b,(b+n)-1
Rotate left immediate	rotlwi ra,rs,n	rlwinm ra,rs,n,0,31
Rotate right immediate	rotrwi ra,rs,n	rlwinm ra,rs,32-n,0,31
Rotate left	rotlw ra,rs,rb	rlwnm ra,rs,rb,0,31
Shift left immediate	slwi ra,rs,n (n < 32)	rlwinm ra,rs,n,0,31-n
Shift right immediate	srwi ra,rs,n (n < 32)	rlwinm ra,rs,32-n,n,31
Clear left immediate	clrlwi ra,rs,n (n < 32)	rlwinm ra,rs,0,n,31
Clear right immediate	clrrwi ra,rs,n (n < 32)	rlwinm ra,rs,0,0,31-n
Clear left and shift left immediate	clrlslwi ra,rs,b,n (n ≤ b < 32)	rlwinm ra,rs,n,b-n,31-n

Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.

   extrwi Rx,Ry,1,0 (equivalent to: rlwinm Rx,Ry,1,31,31)

2. Insert the bit extracted in (1) into the sign bit (bit 32) of register Rz.

   insrwi Rz,Rx,1,0 (equivalent to: rlwimi Rz,Rx,31,0,0)

3. Shift the contents of register Rx left 8 bits, clearing the high-order 32 bits.

   slwi Rx,Rx,8 (equivalent to: rlwinm Rx,Rx,8,0,23)

4. Clear the high-order 16 bits of the low-order 32 bits of register Ry and place the result into register Rx, clearing the high-order 32 bits of register Rx.

   clrlwi Rx,Ry,16 (equivalent to: rlwinm Rx,Ry,0,16,31)

E.9 Move To/From Special Purpose Register Mnemonics

The ***mtspr*** and ***mfspr*** instructions specify a Special Purpose Register (SPR) as a numeric operand. Extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand.

Table 134:Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR	
	Extended	Equivalent to	Extended	Equivalent to
Fixed-Point Exception Register (XER)	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register (LR)	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register (CTR)	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Authority Mask Register (AMR) <S>	mtuamr Rx	mtspr 13,Rx	mfuamr Rx	mfspr Rx,13
Program Priority Register (PPR) <S>	mtppr Rx	mtspr 896,Rx	mfppr Rx	mfspr Rx,896
Program Priority Register 32-Bit (PPR32)[1]	mtppr32 Rx	mtspr 898,Rx	mfppr32 Rx	mfspr Rx,898

Examples

1. Copy the contents of register Rx to the XER.

   mtxer Rx (equivalent to: mtspr 1,Rx)

2. Copy the contents of the LR to register Rx.

   mflr Rx (equivalent to: mfspr Rx,8)

3. Copy the contents of register Rx to the CTR.

   mtctr Rx (equivalent to: mtspr 9,Rx)

E.10 Miscellaneous Mnemonics

No-op

Many Power ISA instructions can be coded in a way such that, effectively, no operation is performed. An extended mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that will trigger this.

nop (equivalent to: ori 0,0,0)

For some uses of a no-op instruction, optimizations related to no-ops, such as removal from the execution stream, are not desireable. An extended mnemonic is provided for the executed form of no-op. This form of no-op will still consume execution resources.

xnop (equivalent to: xori 0,0,0)

Load Immediate

The ***addi*** and ***addis*** instructions can be used to load an immediate value into a register. Extended mnemonics are provided to convey the idea that no addition is being performed but merely data movement (from the immediate field of the instruction to a register).

Load a 16-bit signed immediate value into register Rx.

li Rx,value (equivalent to: addi Rx,0,value)

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx.

lis Rx,value (equivalent to: addis Rx,0,value)

Load Address

This mnemonic permits computing the value of a base-displacement operand, using the ***addi*** instruction which normally requires separate register and immediate operands.

```
la      Rx,D(Ry)        (equivalent to:    addi    Rx,Ry,D)
```

The ***la*** mnemonic is useful for obtaining the address of a variable specified by name, allowing the Assembler to supply the base register number and compute the displacement. If the variable v is located at offset Dv bytes from the address in register Rv, and the Assembler has been told to use register Rv as a base for references to the data structure containing v, then the following line causes the address of v to be loaded into register Rx.

```
la      Rx,v            (equivalent to:    addi    Rx,Rv,Dv)
```

Move Register

Several Power ISA instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Ry to register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

```
mr      Rx,Ry           (equivalent to:    or      Rx,Ry,Ry)
```

Complement Register

Several Power ISA instructions can be coded in a way such that they complement the contents of one register and place the result into another register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Ry and places the result into register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

```
not     Rx,Ry           (equivalent to:    nor     Rx,Ry,Ry)
```

Move To/From Condition Register

This mnemonic permits copying the contents of the low-order 32 bits of a GPR to the Condition Register, using the same style as the ***mfcr*** instruction.

```
mtcr    Rx              (equivalent to:    mtcrf   0xFF,Rx)
```

The following instructions may generate either the (old) ***mtcrf*** or ***mfcr*** instructions or the (new) ***mtocrf*** or ***mfocrf*** instruction, respectively, depending on the target machine type assembler parameter.

```
mtcrf   FXM,Rx
mfcr    Rx
```

All three extended mnemonics in this subsection are being phased out. In future assemblers the form "mtcr Rx" may not exist, and the ***mtcrf*** and ***mfcr*** mnemonics may generate the old form instructions (with bit 11 = 0) regardless of the target machine type assembler parameter, or may cease to exist.

Appendix F. Programming Examples

F.1 Multiple-Precision Shifts

This section gives examples of how multiple-precision shifts can be programmed.

A multiple-precision shift is defined to be a shift of an N-doubleword quantity (64-bit mode) or an N-word quantity (32-bit mode), where N>1. The quantity to be shifted is contained in N registers. The shift amount is specified either by an immediate value in the instruction, or by a value in a register.

The examples shown below distinguish between the cases N=2 and N>2. If N=2, the shift amount may be in the range 0 through 127 (64-bit mode) or 0 through 63 (32-bit mode), which are the maximum ranges supported by the *Shift* instructions used. However if N>2, the shift amount must be in the range 0 through 63 (64-bit mode) or 0 through 31 (32-bit mode), in order for the examples to yield the desired result. The specific instance shown for N>2 is N=3; extending those code sequences to larger N is straightforward, as is reducing them to the case N=2 when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case N=3 is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit mode for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in GPR 6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 30 and 31 are used as scratch registers.

For N>2, the number of instructions required is 2N-1 (immediate shifts) or 3N-1 (non-immediate shifts).

Multiple-precision shifts in 64-bit mode [Category: 64-Bit]

Shift Left Immediate, N = 3 (shift amnt < 64)

```
rldicr   r5,r4,sh,63-sh
rldimi   r4,r3,0,sh
rldicl   r4,r4,sh,0
rldimi   r3,r2,0,sh
rldicl   r3,r3,sh,0
```

Shift Left, N = 2 (shift amnt < 128)

```
subfic   r31,r6,64
sld      r2,r2,r6
srd      r30,r3,r31
or       r2,r2,r30
addi     r31,r6,-64
sld      r30,r3,r31
or       r2,r2,r30
sld      r3,r3,r6
```

Shift Left, N = 3 (shift amnt < 64)

```
subfic   r31,r6,64
sld      r2,r2,r6
srd      r30,r3,r31
or       r2,r2,r30
sld      r3,r3,r6
srd      r30,r4,r31
or       r3,r3,r30
sld      r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 64)

```
rldimi   r4,r3,0,64-sh
rldicl   r4,r4,64-sh,0
rldimi   r3,r2,0,64-sh
rldicl   r3,r3,64-sh,0
rldicl   r2,r2,64-sh,sh
```

Shift Right, N = 2 (shift amnt < 128)

```
subfic   r31,r6,64
srd      r3,r3,r6
sld      r30,r2,r31
or       r3,r3,r30
addi     r31,r6,-64
srd      r30,r2,r31
or       r3,r3,r30
srd      r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 64)

```
subfic   r31,r6,64
srd      r4,r4,r6
sld      r30,r3,r31
or       r4,r4,r30
srd      r3,r3,r6
sld      r30,r2,r31
or       r3,r3,r30
srd      r2,r2,r6
```

Multiple-precision shifts in 32-bit mode

Shift Left Immediate, N = 3 (shift amnt < 32)

```
rlwinm   r2,r2,sh,0,31-sh
rlwimi   r2,r3,sh,32-sh,31
rlwinm   r3,r3,sh,0,31-sh
rlwimi   r3,r4,sh,32-sh,31
rlwinm   r4,r4,sh,0,31-sh
```

Shift Left, N = 2 (shift amnt < 64)

```
subfic   r31,r6,32
slw      r2,r2,r6
srw      r30,r3,r31
or       r2,r2,r30
addi     r31,r6,-32
slw      r30,r3,r31
or       r2,r2,r30
slw      r3,r3,r6
```

Shift Left, N = 3 (shift amnt < 32)

```
subfic   r31,r6,32
slw      r2,r2,r6
srw      r30,r3,r31
or       r2,r2,r30
slw      r3,r3,r6
srw      r30,r4,r31
or       r3,r3,r30
slw      r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 32)

```
rlwinm   r4,r4,32-sh,sh,31
rlwimi   r4,r3,32-sh,0,sh-1
rlwinm   r3,r3,32-sh,sh,31
rlwimi   r3,r2,32-sh,0,sh-1
rlwinm   r2,r2,32-sh,sh,31
```

Shift Right, N = 2 (shift amnt < 64)

```
subfic   r31,r6,32
srw      r3,r3,r6
slw      r30,r2,r31
or       r3,r3,r30
addi     r31,r6,-32
srw      r30,r2,r31
or       r3,r3,r30
srw      r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 32)

```
subfic   r31,r6,32
srw      r4,r4,r6
slw      r30,r3,r31
or       r4,r4,r30
srw      r3,r3,r6
slw      r30,r2,r31
or       r3,r3,r30
srw      r2,r2,r6
```

Multiple-precision shifts in 64-bit mode, continued [Category: 64-Bit]

Shift Right Algebraic Immediate, N = 3 (shift amnt < 64)

```
rldimi    r4,r3,0,64-sh
rldicl    r4,r4,64-sh,0
rldimi    r3,r2,0,64-sh
rldicl    r3,r3,64-sh,0
sradi     r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 128)

```
subfic    r31,r6,64
srd       r3,r3,r6
sld       r30,r2,r31
or        r3,r3,r30
addic.    r31,r6,-64
srad      r30,r2,r31
isel      r3,r30,r3,gt
srad      r2,r2,r6
```

Shift Right Algebraic, N = 3 (shift amnt < 64)

```
subfic    r31,r6,64
srd       r4,r4,r6
sld       r30,r3,r31
or        r4,r4,r30
srd       r3,r3,r6
sld       r30,r2,r31
or        r3,r3,r30
srad      r2,r2,r6
```

Multiple-precision shifts in 32-bit mode, continued

Shift Right Algebraic Immediate, N = 3 (shift amnt < 32)

```
rlwinm    r4,r4,32-sh,sh,31
rlwimi    r4,r3,32-sh,0,sh-1
rlwinm    r3,r3,32-sh,sh,31
rlwimi    r3,r2,32-sh,0,sh-1
srawi     r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 64)

```
subfic    r31,r6,32
srw       r3,r3,r6
slw       r30,r2,r31
or        r3,r3,r30
addic.    r31,r6,-32
sraw      r30,r2,r31
isel      r3,r30,r3,gt
sraw      r2,r2,r6
```

Shift Right Algebraic, N = 3 (shift amnt < 32)

```
subfic    r31,r6,32
srw       r4,r4,r6
slw       r30,r3,r31
or        r4,r4,r30
srw       r3,r3,r6
slw       r30,r2,r31
or        r3,r3,r30
sraw      r2,r2,r6
```

F.2 Floating-Point Conversions [Category: Floating-Point]

This section gives examples of how the *Floating-Point Conversion* instructions can be used to perform various conversions.

Warning: Some of the examples use the ***fsel*** instruction. Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section F.3.4, “Notes” on page 730.

F.2.1 Conversion from Floating-Point Number to Floating-Point Integer

The full *convert to floating-point integer* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1 and the result is returned in FPR 3.

```
mtfsb0    23           #clear VXCVI
fctid[z]  f3,f1        #convert to fx int
fcfid     f3,f3        #convert back again
mcrfs     7,5          #VXCVI to CR
bf        31,$+8       #skip if VXCVI was 0
fmr       f3,f1        #input was fp int
```

F.2.2 Conversion from Floating-Point Number to Signed Fixed-Point Integer Doubleword

The full *convert to signed fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fctid[z]  f2,f1        #convert to dword int
stfd      f2,disp(r1)  #store float
ld        r3,disp(r1)  #load dword
```

F.2.3 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Doubleword

The full *convert to unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value $2^{64}-2048$ is in FPR 3, the value 2^{63} is in FPR 4 and GPR 4, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fsel      f2,f1,f1,f0  #use 0 if < 0
fsub      f5,f3,f1     #use max if > max
fsel      f2,f5,f2,f3
fsub      f5,f2,f4     #subtract 2^63
fcmpu     cr2,f2,f4    #use diff if >= 2^63
fsel      f2,f5,f5,f2
fctid[z]  f2,f2        #convert to fx int
stfd      f2,disp(r1)  #store float
ld        r3,disp(r1)  #load dword
blt       cr2,$+8      #add 2^63 if input
add       r3,r3,r4     #  was >= 2^63
```

F.2.4 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full *convert to signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fctiw[z]  f2,f1        #convert to fx int
stfd      f2,disp(r1)  #store float
lwa       r3,disp+4(r1)#load word algebraic
```

F.2.5 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

The full *convert to unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value 2^{32}-1 is in FPR 3, the result is returned in GPR 3, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
fsel      f2,f1,f1,f0   #use 0 if < 0
fsub      f4,f3,f1      #use max if > max
fsel      f2,f4,f2,f3
fctid[z]  f2,f2         #convert to fx int
stfd      f2,disp(r1)   #store float
lwz       r3,disp+4(r1) #load word and zero
```

F.2.6 Conversion from Signed Fixed-Point Integer Doubleword to Floating-Point Number

The full *convert from signed fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
std       r3,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
```

F.2.7 Conversion from Unsigned Fixed-Point Integer Doubleword to Floating-Point Number

The full *convert from unsigned fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the value 2^{32} is in FPR 4, the result is returned in FPR 1, and two doublewords at displacement "disp" from the address in GPR 1 can be used as scratch space.

```
rldicl    r2,r3,32,32   #isolate high half
rldicl    r0,r3,0,32    #isolate low half
std       r2,disp(r1)   #store dword both
std       r0,disp+8(r1)
lfd       f2,disp(r1)   #load float both
lfd       f1,disp+8(r1)
fcfid     f2,f2         #convert each half to
fcfid     f1,f1         #  fp int (exact result)
fmadd     f1,f4,f2,f1   #(2^32)×high + low
```

An alternative, shorter, sequence can be used if rounding according to $FSCPR_{RN}$ is desired and $FPSCR_{RN}$ specifies *Round toward +Infinity* or *Round toward -Infinity*, or if it is acceptable for the rounded answer to be either of the two representable floating-point integers nearest to the given fixed-point integer. In this case the full *convert from unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the value 2^{64} is in FPR 2.

```
std       r3,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
fadd      f4,f1,f2      #add 2^64
fsel      f1,f1,f1,f4   #  if r3 < 0
```

F.2.8 Conversion from Signed Fixed-Point Integer Word to Floating-Point Number

The full *convert from signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space. (The result is exact.)

```
extsw     r3,r3         #extend sign
std       r3,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
```

The following sequence can be used, assuming a word at the address in GPR 1 + GPR 2 can be used as scratch space.

```
stwx      r3,r1,r2      # store word
lfiwax    f1,r1,r2      # load float
fcfid     f1,f1         # convert to fp int
```

F.2.9 Conversion from Unsigned Fixed-Point Integer Word to Floating-Point Number

The full *convert from unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement "disp" from the address in GPR 1 can be used as scratch space. (The result is exact.)

```
rldicl    r0,r3,0,32    #zero-extend
std       r0,disp(r1)   #store dword
lfd       f1,disp(r1)   #load float
fcfid     f1,f1         #convert to fp int
```

F.2.10 Unsigned Single-Precision BCD Arithmetic

addg6s can be used to add or subtract two BCD operands. In these examples it is assumed that r0 contains 0x666...666. (BCD data formats are described in Section 6.3 of Book I.)

Addition of the unsigned BCD operand in register RA to the unsigned BCD operand in register RB can be accomplished as follows.

```
add       r1,RA,r0
add       r2,r1,RB
addg6s    RT,r1,RB
subf      RT,RT,r2    # RT = RA +BCD RB
```

Subtraction of the unsigned BCD operand in register RA from the unsigned BCD operand in register RB can be accomplished as follows. (In this example it is assumed that RB is not register 0.)

```
addi      r1,RB,1
nor       r2,RA,RA    # one's complement of RA
add       r3,r1,r2
addg6s    RT,r1,r2
subf      RT,RT,r3    # RT = RB -BCD RA
```

Additional instructions are needed to handle signed BCD operands, and BCD operands that occupy more than one register (e.g., unsigned BCD operands that have more than 16 decimal digits).

F.2.11 Signed Single-Precision BCD Arithmetic

Addition of the signed 15-digit BCD operand in register RA to the signed BCD operand in register RB can be accomplished as follows. If the signs of operands are different, then the operand of smaller magnitude is subtracted from the operand of larger magnitude and the sign of the larger operand is preserved; otherwise the operands are added and the sign is preserved.

The sign code is in the low order 4 bits of the operands and uses one of the standard encodings. (See Section 6.3 of Book I for a description of BCD and sign encodings.) This example assumes preferred sign option 1 (0b1100 is plus and 0b1101 is minus). For preferred sign option 2 (0b1111 is plus and 0b1101 is minus), replace the ***xori*** after the "SignedSub" label with "***xori RA,RA,2***".

Preserving the appropriate sign code is accomplished by zeroing the sign code of the other operand before performing a 16 digit BCD addition/subtraction. Other addends (ones complement or 6's) must leave the sign code position as zero.

(In this example r11 contains 0x6666 6666 6666 6660.)

```
SignedSub:
xori      RA,RA,1

SignedAdd:
xor       r5,RA,RB
andi.     r5,r5, 15    # compare sign codes
cmpld     cr1,RA,RB    # compare magnitudes
beq       cr0,samesign
ble       cr1,BminusA

# set up for RT = RA -BCD RB
nor       r9,RB,RB     # one's complement of RB
addi      r10,RA,16    # generate the carry in
b         submag

BminusA:
# set up for RT = RB -BCD RA
nor       r9,RA,RA     # one's complement of RA
addi      r10,RB,16    # generate the carry in

submag:
rldicr    r9,r9,0,59   # remove the sign code
add       r8,r10,r9
addg6s    RT,r10,r9
rldicr    RT,RT,0,59   # remove generated 6 from
                       # sign position
subf      RT,RT,r8
b         done

samesign:
rldicr    r8,RB,0,59   # remove the sign code
add       r10,RA,r11   # add 6's
add       r9,r10,r8
addg6s    RT,r10,RB
subf      RT,RT,r9     # RT = RA +BCD RB

done:
```

F.2.12 Unsigned Extended-Precision BCD Arithmetic

Multiple precision BCD arithmetic requires additional code to add/subtract higher order digits and handle the carry between 16 digit groups. For example, the following sequence implements a 32-digit BCD add. In this example the contents of register R3 concatenated with the contents of R4 represent the first 32-digit operand and the contents of register R5 concatenated with the contents of R6 represents the second operand. The contents of register R3 concatenated with the contents of register R4 represents the result.

(In this example r0 contains 0x6666 6666 6666 6666.)

```
add       r10,R4,r0
addc      r9,r10,R6    # generate the carry
addg6s    R4,r10,R6
subf      R4,R4,r9     # RT1 = RA1 +BCD RB1

addze     R5,R5        # propagate the carry
add       r10,R3,r0
add       r9,r10,R5
addg6s    R3,r10,R5
subf      R3,R3,r9     # RT0 = RA0 +BCD RB0
```

Note that an extra instruction (***addze***) is required to propagate the carry so that the same value is used in the subsequent ***add*** and ***addg6s***.

The following sequence implements a 32-digit BCD subtraction. In this example the first operand in R3 and R4 is subtracted from the 2nd operand in R5 and R6.The result is in R3 and R4.

```
addi      r10,R6,1
nor       r9,R4,R4     # one's complement of RA0
addc      r8,r10,r9    # Generate the carry
addg6s    R4,r10,r9
subf      R4,R4,r8     # RT1 = RB1 -BCD RA1

addze     r10,R5       # propagate the carry
nor       r9,R3,R3     # one's complement of RA0
add       r8,r10,r2
addg6s    R3,r10,r9
subf      R3,R3,r8     # RT0 = RB0 -BCD RA0
```

F.3 Floating-Point Selection [Category: Floating-Point]

This section gives examples of how the *Floating Select* instruction can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using ***fsel*** and other Power ISA instructions. In the examples, a, b, x, y, and z are floating-point variables, which are assumed to be in FPRs fa, fb, fx, fy, and fz. FPR fs is assumed to be available for scratch space.

Additional examples can be found in Section F.2, "Floating-Point Conversions [Category: Floating-Point]" on page 726.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section F.3.4.

F.3.1 Comparison to Zero

High-level language:	Power ISA:	Notes
`if a ≥ 0.0 then x ← y` `else x ← z`	`fsel fx,fa,fy,fz`	(1)
`if a > 0.0 then x ← y` `else x ← z`	`fneg fs,fa` `fsel fx,fs,fz,fy`	(1,2)

`if a = 0.0 then x ← y`
`else x ← z`
	`fsel fx,fa,fy,fz`
	`fneg fs,fa`
	`fsel fx,fs,fx,fz`
		(1)

F.3.2 Minimum and Maximum

High-level language:	Power ISA:	Notes
`x ← min(a,b)`	`fsub fs,fa,fb` `fsel fx,fs,fb,fa`	(3,4,5)
`x ← max(a,b)`	`fsub fs,fa,fb` `fsel fx,fs,fa,fb`	(3,4,5)

F.3.3 Simple if-then-else Constructions

High-level language:	Power ISA:	Notes
`if a ≥ b then x ← y` `else x ← z`	`fsub fs,fa,fb` `fsel fx,fs,fy,fz`	(4,5)
`if a > b then x ← y` `else x ← z`	`fsub fs,fb,fa` `fsel fx,fs,fz,fy`	(3,4,5)

`if a = b then x ← y`
`else x ← z`
	`fsub fs,fa,fb`
	`fsel fx,fs,fy,fz`
	`fneg fs,fs`
	`fsel fx,fs,fx,fz`
		(4,5)

F.3.4 Notes

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of ***fsel*** is contemplated.

In these Notes, the "optimized program" is the Power ISA program shown, and the "unoptimized program" (not shown) is the corresponding Power ISA program that uses ***fcmpu*** and *Branch Conditional* instructions instead of ***fsel***.

1. The unoptimized program affects the VXSNAN bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if a is a NaN.
3. The optimized program gives the incorrect result if a and/or b is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if a and b are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects the OX, UX, XX, and VXISI bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

F.4 Vector Unaligned Storage Operations [Category: Vector]

F.4.1 Loading a Unaligned Quadword Using Permute from Big-Endian Storage

The following sequence of instructions copies the unaligned quadword storage operand into VRT.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx       Vhi,0,Rb      # load MSQ
lvsl      Vp,0,Rb     # set permute control vector
addi      Rb,Rb,16      # address of LSQ
lvx       Vlo,0,Rb      # load LSQ
perm      Vt,Vhi,Vlo,Vp # align the data
```

Book II:

Power ISA Virtual Environment Architecture

Chapter 1. Storage Model

1.1 Definitions

The following definitions, in addition to those specified in Book I, are used in this Book. In these definitions, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

- **system**
  A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to system includes services provided by the privileged software.
- **main storage**
  The level of storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.
- **primary cache**
  The level of cache closest to the processor.
- **secondary cache**
  After the primary cache, the next closest level of cache to the processor.
- **instruction storage**
  The view of storage as seen by the mechanism that fetches instructions.
- **data storage**
  The view of storage as seen by a *Load* or *Store* instruction.
- **program order**
  The execution of instructions in the order required by the sequential execution model. (See Section 2.2 of Book I.) A ***dcbz*** instruction that modifies storage which contains instructions has the same effect with respect to the sequential execution model as a *Store* instruction as described there.) An additional exception to the sequential execution model beyond those described in Book I is caused by transaction failure (see Section 5.3.3).
- **storage location**
  A contiguous sequence of one or more bytes in storage. When used in association with a specific instruction or the instruction fetching mechanism, the length of the sequence of one or more bytes is typically implied by the operation. In other uses, it may refer more abstractly to a group of bytes which share common storage attributes.
- **storage access**
  An access to a storage location. There are three (mutually exclusive) kinds of storage access.
  - **data access**

    An access to the storage location specified by a *Load* or *Store* instruction, or, if the access is performed "out-of-order" (see Section 5.5 of Book III-S and Section 6.5 of Book III-E), an access to a storage location as if it were the storage location specified by a *Load* or *Store* instruction.
  - **instruction fetch**

    An access for the purpose of fetching an instruction.
  - **implicit access**

    An access by the processor for the purpose of address translation or reference and change recording (see Book III-S).
- **caused by, associated with**
  - **caused by**

    A storage access is said to be caused by an instruction if the instruction is a *Load* or *Store* and the access (data access) is to the storage location specified by the instruction.
  - **associated with**

    A storage access is said to be associated with an instruction if the access is for the purpose of fetching the instruction (instruction fetch), or is a data access caused by the instruction, or is an implicit access that occurs as a side effect of fetching or executing the instruction.

- **prefetched instructions**
  Instructions for which a copy of the instruction has been fetched from instruction storage, but the instruction has not yet been executed.
- **uniprocessor**
  A system that contains one processor.
- **multiprocessor**
  A system that contains two or more processors.
- **shared storage multiprocessor**
  A multiprocessor that contains some common storage, which all the processors in the system can access.
- **performed**
  A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is performed with respect to P2 when an instruction fetch by P2 will not be satisfied from the copy of the block that existed in its instruction cache when the instruction causing the invalidation was executed, and similarly for a data cache block invalidation.

  The preceding definitions apply regardless of whether P1 and P2 are the same entity.
- **page (virtual page)**
  2^n contiguous bytes of storage aligned such that the effective address of the first byte in the page is an integral multiple of the page size for which protection and control attributes are independently specifiable and for which reference and change status <S> are independently recorded.
- **block**
  The aligned unit of storage operated on by the *Cache Management* instructions. The size of an instruction cache block may differ from the size of a data cache block, and both sizes may vary between implementations. The maximum block size is equal to the minimum page size.
- **aligned storage access**
  A load or store is aligned if the address of the target storage location is a multiple of the size of the transfer effected by the instruction.
- **aggregate store**
  The set of stores caused by a successful transaction, which are performed as an atomic unit.

1.2 Introduction

The Power ISA User Instruction Set Architecture, discussed in Book I, defines storage as a linear array of bytes indexed from 0 to a maximum of 2^{64}-1. Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. The Power ISA Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The Power ISA Virtual Environment Architecture, in conjunction with services based on the Power ISA Operating Environment Architecture (see Book III) and provided by the operating system, permits explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, the Power ISA specifies a relaxed model of storage consistency. In a multiprocessor system that allows multiple copies of a storage location, aggressive implementations of the architecture can permit intervals of time during which different copies of a storage location have different values. This chapter describes features of the Power ISA that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The Power ISA system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a “virtual” address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of “effective address” (EA) space, subject to limitations imposed by the operating system. In a typical Power ISA system, each program's EA space is a subset of a larger “virtual address” (VA) space managed by the operating system.

Each effective address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this, using the address translation mechanism described in Book III. The operating system manages the real (physical) storage resources of the system, by setting up the tables and other information used by the hardware address translation mechanism.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the

operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, "paging" activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including read/write, read only, and no access; see Book III), based on system standards (e.g., program code might be read only) and application requests.

1.4 Single-Copy Atomicity

An access is *single-copy atomic*, or simply *atomic*, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

Vector storage accesses are not guaranteed to be atomic. The following other types of single-register accesses are always atomic:

- byte accesses (all bytes are aligned on byte boundaries)
- halfword accesses aligned on halfword boundaries
- word accesses aligned on word boundaries
- doubleword accesses aligned on doubleword boundaries (64-bit implementations only; see Section 1.2 of Book III-E<E>)

No other accesses are guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic.

- any *Load* or *Store* instruction for which the operand is unaligned
- ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, ***stswx***
- ***lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx***
- any *Cache Management* instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accesses. In general, the number and alignment of these accesses are implementation-dependent, as is the relative order in which they are performed. The only exception to the preceding rule is that, for ***lfdp***, ***lfdpx***, ***stfdp***, and ***stfdpx,*** if the access is aligned on a doubleword boundary, it is performed as a pair of disjoint atomic doubleword accesses.

The results for several combinations of loads and stores to the same or overlapping locations are described below.

1. When two processors execute atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.
2. When two processors execute atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.
3. When two processors execute stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
4. When two processors execute stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.
5. When a processor executes an atomic store to a location, a second processor executes an atomic load from that location, and no other store is performed to that location, the value returned by the load is the contents of the location before the store or the contents of the location after the store.
6. When a load and a store with the same target location can be executed simultaneously, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

1.5 Cache Model

A cache model in which there is one cache for instructions and another cache for data is called a "Harvard-style" cache. This is the model assumed by the Power ISA, e.g., in the descriptions of the *Cache Management* instructions in Section 4.3. Alternative cache models may be implemented (e.g., a "combined cache" model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches), but they support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with modifications to those storage locations (e.g., modifications caused by *Store* instructions).

A location in the data cache is considered to be modified in that cache if the location has been modified (e.g., by a *Store* instruction) and the modified data have not been written to main storage.

Cache Management instructions are provided so that programs can manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that will be executed (i.e., when the program modifies

data in storage and then attempts to execute the modified data as instructions). The *Cache Management* instructions are also useful in optimizing the use of memory bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage control attributes associated with the specified storage location (see Section 1.6, "Storage Control Attributes").

The *Cache Management* instructions allow the program to do the following.

- invalidate the copy of storage in an instruction cache block (***icbi***)
- provide a hint that an instruction will probably soon be accessed from a specified instruction cache block (***icbt***)
- provide a hint that the program will probably soon access a specified data cache block (***dcbt***, ***dcbtst***)
- <E> allocate a data cache block and set the contents of that block to zeros, but perform no operation if no write access is allowed to the data cache block (***dcba***)
- set the contents of a data cache block to zeros (***dcbz***)
- copy the contents of a modified data cache block to main storage (***dcbst***)
- copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid (***dcbf or dcbfl***)

1.6 Storage Control Attributes

Some operating systems may provide a means to allow programs to specify the storage control attributes described in this section. Because the support provided for these attributes by the operating system may vary between systems, the details of the specific system being used must be known before these attributes can be used.

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location, as described below. The storage control attributes are the following.

- Write Through Required
- Caching Inhibited
- Memory Coherence Required
- Guarded
- Endianness<E>
- Strong Access Order [Category: SAO]

These attributes have meaning only when an effective address is translated by the processor performing the storage access.

<E> Additional storage control attributes may be defined for some implementations. See Section 6.8 of Book III-E for additional information.

Programming Note

The Write Through Required and Caching Inhibited attributes are mutually exclusive because, as described below, the Write Through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not.

Storage that is Write Through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stbcx.***, ***sthcx.***, ***stwcx***., ***stdcx.***, and ***stqcx***. instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these attributes.

In the remainder of this section, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*" unless they are explicitly excluded, and similarly for "*Store* instruction".

1.6.1 Write Through Required

A store to a Write Through Required storage location is performed in main storage. A Store instruction that specifies a location in Write Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

In general, accesses caused by separate *Store* instructions that specify locations in Write Through Required storage may be combined into one access. Such combining does not occur if the *Store* instructions are separated by a ***sync***, ***eieio***<S>, or ***mbar***<E> instruction.

1.6.2 Caching Inhibited

An access to a Caching Inhibited storage location is performed in main storage. A *Load* instruction that specifies a location in Caching Inhibited storage may cause additional locations in main storage to be accessed unless the specified location is also Guarded. An instruction fetch from Caching Inhibited storage may cause additional words in main storage to be accessed. No copy of the accessed locations is placed into the caches.

In general, non-overlapping accesses caused by separate *Load* instructions that specify locations in Caching Inhibited storage may be combined into one access, as may non-overlapping accesses caused by separate *Store* instructions that specify locations in Caching Inhibited storage. Such combining does not occur if the *Load* or *Store* instructions are separated by a ***sync*** or ***mbar***<E> instruction. Combining may also occur

among such accesses from multiple processors that share a common memory interface. No combining occurs if the storage is also Guarded.

Programming Note

None of the memory barrier instructions prevent the combining of accesses from different processors. The Guarded storage attribute must be used in combination with Caching Inhibited to prevent such combining.

1.6.3 Memory Coherence Required [Category: Memory Coherence]

An access to a Memory Coherence Required storage location is performed coherently, as follows.

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are *coherent* if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a "newer" value first and then, later, load an "older" value.

Memory coherence is managed in blocks called coherence *blocks*. Their size is implementation-dependent, but is larger than a word and is usually the size of a cache block.

For storage that is not Memory Coherence Required, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this may be system-dependent.

Because the Memory Coherence Required attribute for a given storage location is of little use unless all processors that access the location do so coherently, in statements about Memory Coherence Required storage elsewhere in this document it is generally assumed that the storage has the Memory Coherence Required attribute for all processors that access it.

Programming Note

Operating systems that allow programs to request that storage not be Memory Coherence Required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.

In most systems the default is that all storage is Memory Coherence Required. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be Memory Coherence Required, and can manage the coherence of that storage by using the ***sync*** instruction, the *Cache Management* instructions, and services provided by the operating system.

1.6.4 Guarded

A data access to a Guarded storage location is performed only if either (a) the access is caused by an instruction that is known to be required by the sequential execution model, or (b) the access is a load and the storage location is already in a cache. If the storage is also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

For the Server environment, instructions are not fetched from virtual storage that is Guarded. If the instruction addressed by the current instruction address is in such storage, the system instruction storage error handler may be invoked (see Section 6.5.5 of Book III-S).

Programming Note

In some implementations, instructions may be executed before they are known to be required by the sequential execution model. Because the results of instructions executed in this manner are discarded if it is later determined that those instructions would not have been executed in the sequential execution model, this behavior does not affect most programs.

This behavior does affect programs that access storage locations that are not "well-behaved" (e.g., a storage location that represents a control register on an I/O device that, when accessed, causes the device to perform an operation). To avoid unintended results, programs that access such storage locations should request that the storage be Guarded, and should prevent such storage locations from being in a cache (e.g., by requesting that the storage also be Caching Inhibited).

1.6.5 Endianness [Category: Embedded.Little-Endian]

The Endianness storage control attribute specifies the byte ordering (Big-Endian or Little-Endian) that is used when the storage location is accessed; see Section 1.10 of Book I.

1.6.6 Variable Length Encoded (VLE) Instructions

VLE storage is used to store VLE instructions. Instructions fetched from VLE storage are processed as VLE instructions. VLE storage must also be Big-Endian. Instructions fetched from VLE storage that is Little-Endian cause a Byte-ordering exception, and the system instruction storage error handler will be invoked.

The VLE attribute has no effect on data accesses. See Chapter 1 of Book VLE.

1.6.7 Strong Access Order [Category: SAO]

All accesses to storage with the Strong Access Order (SAO) attribute (referred to as *SAO storage*) will be performed using a set of ordering rules different from that of the weakly consistent model that is described in Section 1.7.1, “Storage Access Ordering”. These rules apply only to accesses that are caused by a *Load* or a *Store*, and not to accesses associated with those instructions. Furthermore, these rules do not apply to accesses that are caused by or associated with instructions that are stated in their descriptions to be “treated as a *Load*” or “treated as a *Store*.” The details are described below, from the programmer’s point of view. (The processor may deviate from these rules if the programmer cannot detect the deviation.) The SAO attribute is not intended to be used for general purpose programming. It is provided in a manner that is not fully independent of the other storage attributes. Specifically, it is only provided for storage that is Memory Coherence Required, but not Write Through Required, not Caching Inhibited, and not Guarded. See Section 5.8.2.1, “Storage Control Bit Restrictions”, in Book III-S for more details. Accesses to SAO storage are likely to be performed more slowly than similar accesses to non-SAO storage.

The order in which a processor performs storage accesses to SAO storage, the order in which those accesses are performed with respect to other processors and mechanisms, and the order in which those accesses are performed in main storage are the same except in the circumstances described in the following paragraph. The ordering rules for accesses performed by a single processor to SAO storage are as follows. Stores are performed in program order. When a store accesses data adjacent to that which is accessed by the next store in program order, the two storage accesses may be combined into a single larger access. Loads are performed in program order. When a load accesses data adjacent to that which is accessed by the next load in program order, the two storage accesses may be combined into a single larger access. Stores may not be performed before loads which precede them in program order. Loads may be performed before stores which precede them in program order, with the provision that a load which follows a store of the same datum (to the same address) must obtain a value which is no older (in consideration of the possibility of programs on other processors sharing the same storage) than the value stored by the preceding store.

When any given processor loads the datum it just stored, as described above, the load may be performed by the processor before the preceding store has been performed with respect to other processors and mechanisms, and in main storage. This may cause the processor to see its store earlier relative to stores performed by other processors than it is observed by other processors and mechanisms, and than it is performed in memory. A direct consequence of this consideration is that although programs running on each processor will see the same sequence of accesses from any individual processor to SAO storage, each may in general see a different interleaving of the individual sequences. The memory barrier instructions may be used to establish stronger ordering, as described in Section 1.7.1, “Storage Access Ordering”, beginning with the third major bullet.

1.7 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be *aliases*. Each application can be granted separate access privileges to aliased pages.

1.7.1 Storage Access Ordering

The Power ISA defines two models for the ordering of storage accesses: weakly consistent and strong access ordering. The predominant model is weakly consistent. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when storage is shared by two or more programs. Implementations which support Category SAO apply a stronger consistency model among accesses to SAO storage. The order between accesses to SAO storage and those performed using the weakly consistent model is characteristic of the weakly consistent model. The following description, through the second major bullet, applies only to the weakly consistent model. The corresponding description for SAO storage is found in Section 1.6.7, “Strong Access Order [Category: SAO]”. The rest of the description following the second bulletted item applies to both models.

The order in which the processor performs storage accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main storage may all be different. Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs, or with mechanisms such as I/O devices. These means are listed below. The phrase “to the extent required by the associated Memory Coherence Required attributes” refers to the Memory Coherence Required attribute, if any, associated with each access.

- If two *Store* instructions or two *Load* instructions specify storage locations that are both Caching Inhibited and Guarded, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.
- If a *Load* instruction depends on the value returned by a preceding *Load* instruction (because the value is used to compute the effective address specified by the second *Load*), the corresponding storage accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated Memory Coherence Required attributes. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first *Load* is ANDed with zero and then added to the effective address specified by the second *Load*).
- When a processor (P1) executes a *Synchronize*, ***eieio***<S>, or ***mbar***<E> instruction a *memory barrier* is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions following the barrier-creating instruction. For each applicable pair a_i,b_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism.

  The ordering done by a memory barrier is said to be “cumulative” if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.

  - A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
  - B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.

No ordering should be assumed among the storage accesses caused by a single instruction (i.e, by an instruction for which the access is not atomic), even if the accesses are to SAO storage, and no means are provided for controlling that order.

Programming Note

Because stores cannot be performed "out-of-order" (see Book III), if a *Store* instruction depends on the value returned by a preceding *Load* instruction (because the value returned by the *Load* is used to compute either the effective address specified by the *Store* or the value to be stored), the corresponding storage accesses are performed in program order. The same applies if *whether* the *Store* instruction is executed depends on a conditional *Branch* instruction that in turn depends on the value returned by a preceding *Load* instruction.

Because an ***isync*** instruction prevents the execution of instructions following the ***isync*** until instructions preceding the ***isync*** have completed, if an ***isync*** follows a conditional *Branch* instruction that depends on the value returned by a preceding *Load* instruction, the load on which the *Branch* depends is performed before any loads caused by instructions following the ***isync***. This applies even if the effects of the "dependency" are independent of the value loaded (e.g., the value is compared to itself and the *Branch* tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction.

With the exception of the cases described above and earlier in this section, data dependencies and control dependencies do not order storage accesses. Examples include the following.

- If a *Load* instruction specifies the same storage location as a preceding *Store* instruction and the location is in storage that is not Caching Inhibited, the load may be satisfied from a "store queue" (a buffer into which the processor places stored values before presenting them to the storage subsystem), and not be visible to other processors and mechanisms. A consequence is that if a subsequent *Store* depends on the value returned by the *Load*, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a *Store Conditional* instruction may complete before its store has been performed, a conditional *Branch* instruction that depends on the CR0 value set by a *Store Conditional* instruction does not order the *Store Conditional*'s store with respect to storage accesses caused by instructions that follow the *Branch*.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (e.g., branches) do not order storage accesses except as described above. For example, when a subroutine returns to its caller the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Because processors may implement nonarchitected duplicates of architected resources (e.g., GPRs, CR fields, and the Link Register), resource dependencies (e.g., specification of the same target register for two Load instructions) do not order storage accesses.

Examples of correct uses of dependencies, ***sync***, ***lwsync***, and ***eieio***<S> to order storage accesses can be found in Appendix B. "Programming Examples for Sharing Storage" on page 829.

Because the storage model is weakly consistent, the sequential execution model as applied to instructions that cause storage accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before storage accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same storage location, if the location is in storage that is Memory Coherence Required the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is in storage that is Caching Inhibited.

Because accesses to storage that is Caching Inhibited are performed in main storage, memory barriers and dependencies on *Load* instructions order such accesses with respect to any processor or mechanism even if the storage is not Memory Coherence Required.

Programming Note

The first example below illustrates cumulative ordering of storage accesses preceding a memory barrier, and the second illustrates cumulative ordering of storage accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

Processor A:
stores the value 1 to location X

Processor B:
loads from location X obtaining the value 1, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor C:
loads from location Y obtaining the value 2, executes a ***sync*** instruction, then loads from location X

Example 2:

Processor A:
stores the value 1 to location X, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor B:
loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z

Processor C:
loads from location Z obtaining the value 3, executes a ***sync*** instruction, then loads from location X

In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

1.7.2 Storage Ordering of I/O Accesses

A “coherence domain” consists of all processors and all interfaces to main storage. Memory reads and writes initiated by mechanisms outside the coherence domain are performed within the coherence domain in the order in which they enter the coherence domain and are performed as coherent accesses.

1.7.3 Atomic Update

The *Load And Reserve* and *Store Conditional* instructions together permit atomic update of a shared storage location. There are byte, halfword, word, doubleword, and quadword forms of each of these instructions. Described here is the operation of the word forms ***lwarx*** and ***stwcx.***; operation of the byte, halfword, doubleword, and quadword forms ***lbarx***, ***stbcx.***, ***lharx***, ***sthcx.***, ***ldarx, stdcx.***, ***lqarx,*** and ***stqcx.*** is the same except for obvious substitutions.

The ***lwarx*** instruction is a load from a word-aligned location that has two side effects. Both of these side effects occur at the same time that the load is performed.

1. A reservation for a subsequent ***stwcx.*** instruction is created.
2. The memory coherence mechanism is notified that a reservation exists for the storage location specified by the ***lwarx***.

The ***stwcx.*** instruction is a store to a word-aligned location that is conditioned on the existence of the reservation created by the ***lwarx*** and on whether the same storage location is specified by both instructions. To emulate an atomic operation with these instructions, it is necessary that both the ***lwarx*** and the ***stwcx***. specify the same storage location.

A ***stwcx.*** performs a store to the target storage location only if the storage location specified by the ***lwarx*** that established the reservation has not been stored into by another processor or mechanism since the reservation was created. If the storage locations specified by the two instructions differ, the store is not necessarily performed except that if the Store Conditional Page Mobility category is supported and the storage locations are in different naturally aligned blocks of real storage whose size is the smallest real page size supported by the implementation, the store is not performed.

A ***stwcx.*** that performs its store is said to "succeed".

Examples of the use of ***lwarx*** and ***stwcx.*** are given in Appendix B. "Programming Examples for Sharing Storage" on page 829.

A successful ***stwcx.*** to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or ***lwarx*** from the given location by another processor may return a "stale" value. However, a subsequent ***lwarx*** from the given location by the other processor followed by a successful ***stwcx.*** by that processor is guaranteed to have returned the value stored by the first processor's ***stwcx***. (in the absence of other stores to the given location).

If a *Store Conditional* instruction is used with a preceding *Load and Reserve* instruction that has a different storage operand length (e.g., ***stwcx***. with ***ldarx***), the reservation is cleared and it is undefined whether the store is performed.

Programming Note

The store caused by a successful ***stwcx.*** is ordered, by a dependence on the reservation, with respect to the load caused by the ***lwarx*** that established the reservation, such that the two storage accesses are performed in program order with respect to any processor or mechanism.

Programming Note

Before reassigning a virtual address to a different real page, privileged software may need to clear all processors' reservations for the original real page in order to avoid a *Store Conditional* being successful only because the corresponding reservation for the original location is not cleared by a store to the new real page by some other processor or mechanism. This clearing of reservations is unnecessary on processors that support the Store Conditional Page Mobility category.

The Store Conditional Page Mobility category does not provide a mechanism for the *Store Conditional* instruction to detect that a virtual page has been moved to a new real page and back again to the original real page that was accessed by a *Load and Reserve* instruction. Privileged software that moves a virtual page could clear the reservation on the processor it is running on in order to ensure that a *Store Conditional* instruction executed by that processor does not succeed in this case. (The stores that occur naturally as part of moving the virtual page will cause any reservations, held by other processors, in the target real page to be lost.)

1.7.3.1 Reservations

The ability to emulate an atomic operation using ***lwarx*** and ***stwcx.*** is based on the conditional behavior of ***stwcx.***, the reservation created by ***lwarx***, and the clearing of that reservation if the target storage location is modified by another processor or mechanism before the ***stwcx.*** performs its store.

A reservation is held on an aligned unit of real storage called a reservation granule. The size of the reservation granule is 2^n bytes, where n is implementation-dependent but is always at least 4 (thus the minimum reservation granule size is a quadword) and, if the Store Conditional Page Mobility category is supported, where 2^n is not larger than the smallest real page size supported by the implementation. The reservation granule associated with effective address EA contains the real address to which EA maps. ("real_addr(EA)" in the RTL for the *Load And Reserve* and *Store Conditional* instructions stands for "real address to which EA maps".) The reservation also has an associated length, which is equal to the storage operand length, in bytes, of the *Load and Reserve* instruction that established the reservation.

A processor has at most one reservation at any time. A reservation is established by executing a ***lbarx***, ***lharx***, ***lwarx, ldarx***, or ***lqarx*** instruction, as described in item 1 below, and is lost or may be lost, depending on the item, if any of the following occur. Items 1-8 apply only if the relevant access is performed. (For example, an access that would ordinarily be caused by an instruc-

tion might not be performed if the instruction causes the system error handler to be invoked.)

1. The processor holding the reservation executes another ***lbarx***, ***lharx***, ***lwarx,*** or ***ldarx***: this clears the first reservation and establishes a new one.
2. The processor holding the reservation executes any ***stbcx.***, ***sthcx.***, ***stwcx., stdcx.***, or ***stqcx.***, regardless of whether the specified address matches the address specified by the ***lbarx***, ***lharx***, ***lwarx, ldarx***, or ***lqarx*** that established the reservation, and regardless of whether the storage operand lengths of the two instructions are the same.
3. <TM> Any of the following occurs on the processor holding the reservation.
   a. The transaction state changes (from Non-transactional, Transactional, or Suspended state to one of the other two states; see Section 5.2, “Transactional Memory Facility States”), except in the following cases
      - If the change is from Transactional state to Suspended state, the reservation is not lost.
      - If the change is from Suspended state to Transactional state, the reservation is not lost if it was established in Transactional state.
      - If the change is caused by a ***treclaim.*** or ***trechkpt.*** instruction, whether the reservation is lost is undefined.

   b. The transaction nesting depth (see Section 5.4, “Transactional Memory Facility Registers”) changes; whether the reservation is lost is undefined. (This item applies only if the processor is in Transactional state both before and after the change.)

   c. The processor is in Suspended state and executes a *Store Conditional* instruction (***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.***) or a ***waitrsv*** instruction; the reservation is lost if it was established in Transactional state. In this case the *Store Conditional* instruction’s store is not performed, and the ***waitrsv*** does not wait. (For *Store Conditional*, the reservation is also lost if it was established in Suspended state; see item 2.)
4. Some other processor executes a *Store,* ***dcbz,*** or ***dcbzep***<E> that specifies a location in the same reservation granule.
5. Some other processor executes a ***dcbtst***, ***dcbtstep***<E>, or ***dcbtstls***<E> that specifies a location in the same reservation granule: whether the reservation is lost is undefined. (For a ***dcbtst*** instruction that specifies a data stream, "location" in the preceding sentence includes all locations in the data stream.)
6. <E> Some other processor executes a ***dcba*** that specifies a location in the same reservation granule: the reservation is lost if the instruction causes the target block to be newly established in a data cache or to be modified; otherwise whether the reservation is lost is undefined.
7. <E> Some other processor executes a ***dcbi*** that specifies a location in the same reservation granule: the reservation may be lost if the instruction is treated as a *Store*.
8. <S> Any processor modifies a Reference or Change bit (see Book III-S) in the same reservation granule: whether the reservation is lost is undefined.
9. Some mechanism other than a processor modifies a storage location in the same reservation granule.
10. An interrupt (see Book III) occurs on the processor holding the reservation: for the Embedded environment, the reservation may be lost if the interrupt is asynchronous. (For the Server environment the reservation is not lost. However, for both environments, system software invoked by interrupts may clear the reservation.)
11. Implementation-specific characteristics of the coherence mechanism cause the reservation to be lost.

Virtualized Implementation Note

A reservation may be lost if:

- Software executes a privileged instruction or utilizes a privileged facility
- Software accesses storage not intended for general-purpose programming
- Software executes a Decorated Storage instruction <DS>
- Software accesses a Device Control Register

Programming Note

One use of ***lwarx*** and ***stwcx.*** is to emulate a "Compare and Swap" primitive like that provided by the IBM System/370 Compare and Swap instruction; see Section B.1, "Atomic Update Primitives" on page 829. A System/370-style Compare and Swap checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The combination of ***lwarx*** and ***stwcx.*** improves on such a Compare and Swap, because the reservation reliably binds the ***lwarx*** and ***stwcx.*** together. The reservation is always lost if the word is modified by another processor or mechanism between the ***lwarx*** and ***stwcx.***, so the ***stwcx.*** never succeeds unless the word has not been stored into (by another processor or mechanism) since the ***lwarx***.

Programming Note

In general, programming conventions must ensure that ***lwarx*** and ***stwcx.*** specify addresses that match; a ***stwcx.*** should be paired with a specific ***lwarx*** to the same storage location. Situations in which a ***stwcx.*** may erroneously be issued after some ***lwarx*** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a ***lwarx*** and before the paired ***stwcx.***. The ***stwcx.*** in the new context might succeed, which is not what was intended by the programmer. Such a situation must be prevented by executing a ***stbcx.***, ***sthcx***., ***stwcx.***, ***stdcx.***, or ***stqcx.*** that specifies a dummy writable aligned location as part of the context switch; see Section 6.4.3 of Book III-S and Section 7.5 of Book III-E.

Programming Note

Because the reservation is lost if another processor stores anywhere in the reservation granule, lock words (or bytes, halfwords, or doublewords) should be allocated such that few such stores occur, other than perhaps to the lock word itself. (Stores by other processors to the lock word result from contention for the lock, and are an expected consequence of using locks to control access to shared storage; stores to other locations in the reservation granule can cause needless reservation loss.) Such allocation can most easily be accomplished by allocating an entire reservation granule for the lock and wasting all but one word. Because reservation granule size is implementation-dependent, portable code must do such allocation dynamically.

Similar considerations apply to other data that are shared directly using ***lwarx*** and ***stwcx.*** (e.g., pointers in certain linked lists; see Section B.3, "List Insertion" on page 833).

1.7.3.2 Forward Progress

Forward progress in loops that use ***lwarx*** and ***stwcx.*** is achieved by a cooperative effort among hardware, system software, and application software.

The architecture guarantees that when a processor executes a ***lwarx*** to obtain a reservation for location X and then a ***stwcx.*** to store a value to location X, either

1. the ***stwcx.*** succeeds and the value is written to location X, or
2. the ***stwcx.*** fails because some other processor or mechanism modified location X, or
3. the ***stwcx.*** fails because the processor's reservation was lost for some other reason.

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor or mechanism writing elsewhere in the reservation granule, cancellation caused by the operating system in managing certain limited resources such as real storage, and cancellation caused by any of the other effects listed in see Section 1.7.3.1.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

Virtualized Implementation Note

On a virtualized implementation, Case 3 includes reservation loss caused by the virtualization software. Thus, on a virtualized implementation, a reservation may be lost at any time without apparent cause. The virtualization software participates in any forward progress assurances, as described above.

Programming Note

The architecture does not include a "fairness guarantee". In competing for a reservation, two processors can indefinitely lock out a third.

1.8 Transactions [Category: Transactional Memory]

A transaction is a group of instructions that collectively have unique storage access behavior intended to facilitate parallel programming. (It is possible to nest transactions within one another. The description in this chapter will ignore nesting because it does not have a significant impact on the properties of the memory model. Nesting and its consequences will be described elsewhere.) Sequences of instructions that are part of the transaction may be interleaved with sequences of Suspended state instructions that are not part of the transaction. A transaction is said to "succeed" or to "fail," and failure may happen before all of the instructions in the transaction have completed. If the transaction fails, it is as if the instructions that are part of the transaction were never executed. If the transaction succeeds, it appears to execute as an atomic unit as viewed by other processors and mechanisms. (Although the transaction appears to execute atomically, some knowledge of the inner workings will be necessary to avoid apparent paradoxes in the rest of the model. These details are described below.) The execution of Suspended state sequences have the same effect that the sequence would have in the absence of a transaction, independent of the success or failure of the transaction, including accessing storage according to the weakly consistent storage model or SAO, based on storage attributes. Upon failure, normal execution continues at the failure handler. Except for the rollback of the effects of transactional instructions upon transaction failure, as viewed by the executing thread, the interleaved sequences of Transactional and Suspended state instructions appear to execute according to the sequential execution model. See Chapter 5. "Transactional Memory Facility [Category: Transactional Memory]" on page 795 for more details. The unique attributes of the storage model for transactions are described below.

Transaction processing does not support the rollback of operations on the reservation mechanism. To prevent this possibility, a reservation is lost as a result of a state change from Transactional to Non-transactional or Non-transactional to Transactional. It is possible to successfully complete an atomic update in Transactional state, though such a sequence would have no benefit. It is also possible to complete an atomic update in Suspended state, or straddling an interval in Suspended state if Suspended state is entered via an interrupt or ***tsuspend.*** and exited via ***tresume.***, ***rfebb***, ***rfid***, ***hrfid***, or ***mtmsrd***. However, an atomic update will not succeed if only one of the *Load and Reserve* / *Store Conditional* instruction pair is executed in Suspended state.

Programming Note

Note that if a *Store Conditional* instruction within a transaction does not store, it may still be possible for the transaction to succeed. Software must not depend on the two operations having the same outcome. For example, software must not use success of an enclosing transaction as a replacement for checking the condition code from a transactional *Store Conditional* instruction.

Programming Note

Accessing storage locations in Suspended state that have been accessed transactionally has the potential to create apparent storage paradoxes. Consider, for example, a case where variable X has intial value zero, is updated transactionally to one, is read in Suspended state, subsequently the transaction fails, and variable X is read again. In the absence of external conflicts, the observed sequence of values will be zero, one, zero: old, new, old.

Performing an atomic update on X in Suspended state may be even more confusing. Suppose the atomic sequence increments X, but that the only way to have X=1 is via the transactional store that occurs before entering Suspended state. The store conditional, if it succeeds, will store X=2 and in so doing, kill the transaction. But with the transaction having failed, X was never equal to one.

The flexibility of the Suspended state programming model can create unintuitive results. It must be used with care.

Successful transactions are serialized in some order, and no processor or mechanism is able to observe the accesses caused by any subset of these transactions as occurring in an order that conflicts with this order. Specifically, let processor i execute transactions 0, 1,…, j, j+1, …, where only successful transactions are numbered, and the numbering reflects program order. Let T_{ij} be transaction j on processor i. Then there is an ordering of the T_{ij} such that no processor or mechanism is able to observe the accesses caused by the transactions T_{ij} in an order that conflicts with this order-

ing. Note that Suspended state storage accesses are not included in the serialization property.

Programming Note

The ordering of the T_{ij} for a given i is consistent with program order for processor i.

Because of the difference between a transaction's instantaneous appearance and the finite time required to execute it in an implementation, it is exposed to changes in memory management state in a way that is not true for individual accesses. A change to the translation or protection state that would prevent any access from taking place at any time during its processing for the transaction compromises the integrity of the transaction. Any such change must either be prevented or must cause the transaction to fail. The architecture will automatically fail a transaction if the memory management state change is accomplished using ***tlbie***. An implementation may overdetect such conflicts between the ***tlbie*** and the transaction footprint. (Overdetection may result from the technique used to detect the conflict. A bloom filter may be used, as an example. Subsequent references to translation invalidation conflicts implicitly include any cases of spurious overdetection.) Changes made in some other manner must be managed by software, for example by explicitly terminating any affected transactions. Examples of instructions that require software management are ***tlbiel***, ***slbie***, ***slbia***, and ***tlbia***.

The atomic nature of a transaction, together with the cumulative memory barrier created by the transaction and the memory barriers created by ***tbegin.*** and ***tend.*** described below, has the potential to eliminate the need for explicit memory barriers within the transaction, and before and after the transaction as well. However, since there may be a desire to preserve existing algorithms while exploiting transactions, the interaction of memory barriers and transactions is defined. In the presence of transactions, storage access ordering is the same as if no transactions are present, with the following exceptions. Memory barriers that are created while the transaction is running (other than the integrated cumulative memory barrier of the transaction described below), data dependencies, and SAO do not order transactional stores. Instead, transactional stores are grouped together into an "aggregate store," which is performed as an atomic unit with respect to other processors and mechanisms when the transaction succeeds, after all the transactional loads have been performed. With this store behavior, the appearance of transactional atomicity is created in a manner similarly to that for a *Load and Reserve* / *Store Conditional* pair. Success of the transaction is conditional on the storage locations specified by the loads not having been stored into by a more recent Suspended state store or by any store by another processor or mechanism since the load was performed. (There are additional conditions for the success of transactions.)

A ***tbegin.*** instruction that begins a successful transaction creates a memory barrier that immediately precedes the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism. Set A contains all data accesses caused by instructions preceding the ***tbegin.*** that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the ***tbegin.***, including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited.

Programming Note

The reason the creation of the memory barrier by ***tbegin.*** is specified to be contingent on the transaction succeeding is that delaying the creation may improve performance, and does not seriously inconvenience software.

A successful transaction has an integrated memory barrier behavior. When a processor (P1) executes a ***tend.*** instruction and ***tend.*** processing determines that the transaction will succeed, a memory barrier is created, which orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism. Set A contains all non-transactional data accesses by other processors and mechanisms that have been performed with respect to P1 before the memory barrier is created and are neither Write Through Required nor Caching Inhibited. Set B contains the aggregate store and all non-transactional data accesses by other processors and mechanisms that are performed after a *Load* instruction executed by that processor or mechanism has returned the value stored by a store that is in set B. Note that the cumulative memory barrier does not order Suspended state storage accesses interleaved with the transaction.

A ***tend.*** instruction that ends a successful transaction creates a memory barrier that immediately follows the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or

mechanism. Set A contains all data accesses caused by instructions preceding the ***tend.,*** including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the ***tend.*** that are neither Write Through Required nor Caching Inhibited.

Programming Note

The barriers that are created by the execution of a successful transaction (those associated with ***tbegin.***, ***tend.***, and the integrated cumulative barrier) render most explicit barriers in and around transactions redundant. An exception is when there is a need to establish order among Suspended state accesses.

1.8.1 Rollback-Only Transactions

A Rollback-Only Transaction (ROT) is a sequence of instructions that is executed, or not, as a unit. The purpose of the ROT is to enable bulk speculation of instructions with minimum overhead. It leverages the rollback mechanism that is invoked as part of transaction failure handling, but has reduced overhead in that it does not have the full atomic nature of the transaction and its synchronization and serialization properties. The absence of a (normal) transaction's atomic quality means that a ROT must not be used to manipulate shared data.

More specifically, a ROT differs from a normal transaction as follows.

- ROTs are not serialized.
- There are no memory barriers created by ***tbegin.*** and ***tend.***
- A ROT has no integrated cumulative memory barrier.
- There is no monitoring of storage locations specified by loads for modification by other processors and mechanisms between the performing of the loads and the completion of the ROT.
- The stores that are included in the ROT need not appear to be performed as an aggregate store. (Implementations are likely to provide an aggregate store appearance, but the correctness of the program must not depend on the aggregate store appearance.)

1.9 Instruction Storage

The instruction execution properties and requirements described in this section, including its subsections, apply only to instruction execution that is required by the sequential execution model.

In this section, including its subsections, it is assumed that all instructions for which execution is attempted are in storage that is not Caching Inhibited and (unless instruction address translation is disabled; see Book III-S) is not Guarded, and from which instruction fetching does not cause the system error handler to be invoked (e.g., from which instruction fetching is not prohibited by the "address translation mechanism" or the "storage protection mechanism"; see Book III).

Programming Note

The results of attempting to execute instructions from storage that does not satisfy this assumption are described in Section 1.6.2 and Section 1.6.4 of this Book and in Book III.

For each instance of executing an instruction from location X, the instruction may be fetched multiple times.

The instruction cache is not necessarily kept consistent with the data cache or with main storage. It is the responsibility of software to ensure that instruction storage is consistent with data storage when such consistency is required for program correctness.

After one or more bytes of a storage location have been modified and before an instruction located in that storage location is executed, software must execute the appropriate sequence of instructions to make instruction storage consistent with data storage. Otherwise the result of attempting to execute the instruction is boundedly undefined except as described in Section 1.9.1, "Concurrent Modification and Execution of Instructions" on page 752.

Programming Note

Following are examples of how to make instruction storage consistent with data storage. Because the optimal instruction sequence to make instruction storage consistent with data storage may vary between systems, many operating systems will provide a system service to perform this function.

Case 1: The given program does not modify instructions executed by another program nor does another program modify the instructions executed by the given program.

Assume that location X previously contained the instruction A0; the program modified one of more bytes of that location such that, in data storage, the location contains the instruction A1; and location X is wholly contained in a single cache block. The following instruction sequence will make instruction storage consistent with data storage such that if the ***isync*** was in location X-4, the instruction A1 in location X would be executed immediately after the ***isync***.

```
dcbst  X       #copy the block to main storage
sync           #order copy before invalidation
icbi   X       #invalidate copy in instr cache
isync          #discard prefetched instructions
```

Case 2: One or more programs execute the instructions that are concurrently being modified by another program.

Assume program A has modified the instruction at location X and other programs are waiting for program A to signal that the new instruction is ready to execute. The following instruction sequence will make instruction storage consistent with data storage and then set a flag to indicate to the waiting programs that the new instruction can be executed.

```
li     r0,1    #put a 1 value in r0
dcbst  X       #copy the block in main storage
sync           #order copy before invalidation
icbi   X       #invalidate copy in instr cache
sync           #order invalidation before store
               #  to flag
stw r0,flag    #set flag indicating instruction
               #  storage is now consistent
```

The following instruction sequence, executed by the waiting program, will prevent the waiting programs from executing the instruction at location X until location X in instruction storage is consistent with data storage, and then will cause any prefetched instructions to be discarded.

```
lwz    r0,flag #loop until flag = 1 (when 1 is
cmpwi  r0,1    #   loaded, location X in inst'n
bne    $-8     #   storage is consistent with
               #   location X in data storage)
isync          #discard any prefetched inst'ns
```

In the preceding instruction sequence any context synchronizing instruction (e.g., ***rfid***) can be used instead of ***isync***. (For Case 1 only ***isync*** can be used.)

For both cases, if two or more instructions in separate data cache blocks have been modified, the ***dcbst*** instruction in the examples must be replaced by a sequence of ***dcbst*** instructions such that each block containing the modified instructions is copied back to main storage. Similarly, for ***icbi*** the sequence must invalidate each instruction cache block containing a location of an instruction that was modified. The ***sync*** instruction that appears above between "***dcbst*** X" and "***icbi*** X" would be placed between the sequence of ***dcbst*** instructions and the sequence of ***icbi*** instructions.

1.9.1 Concurrent Modification and Execution of Instructions

The phrase “concurrent modification and execution of instructions” (CMODX) refers to the case in which a processor fetches and executes an instruction from instruction storage which is not consistent with data storage or which becomes inconsistent with data storage prior to the completion of its processing. This section describes the only case in which executing this instruction under these conditions produces defined results.

In the remainder of this section the following terminology is used.

- Location X is an arbitrary word-aligned storage location.
- X_0 is the value of the contents of location X for which software has made the location X in instruction storage consistent with data storage.
- X_1, X_2, ..., X_n are the sequence of the first n values occupying location X after X_0.
- X_n is the first value of X subsequent to X_0 for which software has again made instruction storage consistent with data storage.
- The “patch class” of instructions consists of the I-form *Branch* instruction (***b***[***l***][***a***]) and the preferred no-op instruction (***ori*** 0,0,0).

If the instruction from location X is executed after the copy of location X in instruction storage is made consistent for the value X_0 and before it is made consistent for the value X_n, the results of executing the instruction are defined if and only if the following conditions are satisfied.

1. The stores that place the values X_1, ..., X_n into location X are atomic stores that modify all four bytes of location X.
2. Each X_i, $0 \le i \le n$, is a patch class instruction.
3. Location X is in storage that is Memory Coherence Required.

If these conditions are satisfied, the result of each execution of an instruction from location X will be the execution of some X_i, $0 \le i \le n$. The value of the ordinate i associated with each value executed may be different and the sequence of ordinates i associated with a sequence of values executed is not constrained, (e.g., a valid sequence of executions of the instruction at location X could be the sequence X_i, X_{i+2}, then X_{i-1}). If these conditions are not satisfied, the results of each such execution of an instruction from location X are boundedly undefined, and may include causing inconsistent information to be presented to the system error handler.

Programming Note

An example of how failure to satisfy the requirements given above can cause inconsistent information to be presented to the system error handler is as follows. If the value X_0 (an illegal instruction) is executed, causing the system illegal instruction handler to be invoked, and before the error handler can load X_0 into a register, X_0 is replaced with X_1, an *Add Immediate* instruction, it will appear that a legal instruction caused an illegal instruction exception.

Programming Note

It is possible to apply a patch or to instrument a given program without the need to suspend or halt the program. This can be accomplished by modifying the example shown in the Programming Note at the end of Section 1.9 where one program is creating instructions to be executed by one or more other programs.

In place of the *Store* to a flag to indicate to the other programs that the code is ready to be executed, the program that is applying the patch would replace a patch class instruction in the original program with a *Branch* instruction that would cause any program executing the *Branch* to branch to the newly created code. The first instruction in the newly created code must be an ***isync***, which will cause any prefetched instructions to be discarded, ensuring that the execution is consistent with the newly created code. The instruction storage location containing the ***isync*** instruction in the patch area must be consistent with data storage with respect to the processor that will execute the patched code before the *Store* which stores the new *Branch* instruction is performed.

Programming Note

It is believed that all processors that comply with versions of the architecture that precede Version 2.01 support concurrent modification and execution of instructions as described in this section if the requirements given above are satisfied, and that most such processors yield boundedly undefined results if the requirements given above are not satisfied. However, in general such support has not been verified by processor testing. Also, one such processor is known to yield undefined results in certain cases if the requirements given above are not satisfied.

Chapter 2. Effect of Operand Placement on Performance

The placement (location and alignment) of operands in storage affects relative performance of storage accesses, and may affect it significantly. The best performance is guaranteed if storage operands are aligned to the natural boundary of the data accessed. Performance of storage accesses varies depending on the following:

1. Operand Size
2. Operand Alignment
3. Crossing no boundary
4. Crossing a cache block boundary
5. Crossing a virtual page boundary

Figure 1 on page 754 specifies which storage accesses that are permitted, but not necessarily required, to invoke the system alignment error handler, resulting in significantly degraded performance.

Specific platforms may set more restrictive requirements on which storage accesses an implementation is allowed to invoke the system alignment error handler.

The *Move Assist* instructions have no alignment requirements.

Operand			Access may cause invocation of the system alignment error handler
Class	**Size**	**Byte Align[1]**	
Integer	1-byte[2]	any	No
	2-byte	2	No
		< 2	Yes[3]
	4-byte	4	No
		< 4	Yes[2]
	lmw ***stmw***	4	No
		< 4	Yes
	eciwx ***ecowx***	4	No
		< 4	Yes
	8-byte	8	No
		< 8	Yes[2]
	lq ***stq***	16	No
		< 16	Yes[5]
Float	4-byte	4	No
		< 4	Yes
	8-byte	8	No
		odd 4	Yes[2]
		< 4	Yes
	lfdp[***x***] ***stfdp***[***x***]	16	No
		< 16	Yes
VSX	4-byte	4	No
		< 4	Yes
	8-byte	8	No
		odd 4	Yes[2]
		< 4	Yes
	16-byte	16	No
		odd 8 or odd 4	Yes[2]
		< 4	Yes
Vector[4]	any	any	No

1. "N" means any N-byte alignment

   "odd N" means any N-byte alignment but not 2×N-byte alignment

   "< N" means any non-N-byte alignment
2. Including *Move Assist* instructions (***lswi***, ***lswx***, ***stswi***, and ***stswx***).
3. For Category:Server, may be invoked when the storage access uses Little-Endian byte ordering or when the storage access spans a virtual page boundary.

   For Category: Embedded, may be invoked for any unaligned access.
4. Vector storage access instructions are always quadword-aligned.
5. May be invoked for any unaligned access.

Figure 1. Unaligned storage accesses

2.1 Instruction Restart

In this section, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

The following instructions are never restarted after having accessed any portion of the storage operand (unless the instruction causes a "Data Address Watchpoint match", for which the corresponding rules are given in Book III).

1. A *Store* instruction that causes an atomic access and, for the Embedded environment, accesses storage that is Guarded
2. A *Load* instructionthat causes an atomic access to storage that is Guarded and, for the Server environment, is also Caching Inhibited.

Any other *Load* or *Store* instruction may be partially executed and then aborted after having accessed a portion of the storage operand, and then re-executed (i.e., restarted, by the processor or the operating system). If an instruction is partially executed, the contents of registers are preserved to the extent that the correct result will be produced when the instruction is re-executed. Additional restrictions on the partial execution of instructions are described in Section 6.6 of Book III-S and Section 7.7 of Book III-E.

Programming Note

In order to ensure that the contents of registers are preserved to the extent that a partially executed instruction can be re-executed correctly, the registers that are preserved must satisfy the following conditions. For any given instruction, zero or more of the conditions applies.

- For a fixed-point *Load* instruction that is not a multiple or string form, or for an ***eciwx*** instruction, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an update form *Load* or *Store* instruction, the contents of register RA are not altered.

Programming Note

There are many events that might cause a *Load* or *Store* instruction to be restarted. For example, a hardware error may cause execution of the instruction to be aborted after part of the access has been performed, and the recovery operation could then cause the aborted instruction to be re-executed.

When an instruction is aborted after being partially executed, the contents of the instruction pointer indicate that the instruction has not been executed, however, the contents of some registers may have been altered and some bytes within the storage operand may have been accessed. The following are examples of an instruction being partially executed and altering the program state even though it appears that the instruction has not been executed.

1. *Load Multiple*, *Load String*: Some registers in the range of registers to be loaded may have been altered.
2. Any *Store* instruction, ***dcbz***: Some bytes of the storage operand may have been altered.

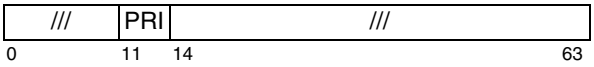
Chapter 3. Management of Shared Resources

The facilities described in this section provide the means to control the use of resources that are shared with other processors.

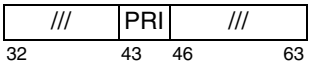
3.1 Program Priority Registers

The Program Priority Register (PPR) is a 64-bit register that controls the program's priority. The PPR provides access to the full 64-bit PPR, and the Program Priority Register 32-bit (PPR32) provides access to the upper 32 bits of the PPR. The Embedded environment only provides access to PPR32. The layouts of the PPR and PPR32 are shown in Figure 2.

PPR [Category: Server]:



PPR32



Bit(s) Description

11:13 ***Program Priority (PRI) ($PPR32_{43:45}$)***

001 very low
010 low
011 medium low
100 medium
101 medium high

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 4.3.6.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

If other values are written to this field, the PRI field is not changed. (See Section 4.3.5 of Book III-S for additional information.)

All other fields are reserved.

Figure 2. Program Priority Register

Programming Note

The ability to access the low-order half of the PPR (and thus the use of ***mfppr*** and ***mtppr***) might be phased out in a future version of the architecture.

Programming Note

By setting the PRI field, a programmer may be able to improve system throughput by causing system resources to be used more efficiently.

E.g., if a program is waiting on a lock (see Section B.2), it could set low priority, with the result that more processor resources would be diverted to the program that holds the lock. This diversion of resources may enable the lock-holding program to complete the operation under the lock more quickly, and then relinquish the lock to the waiting program.

Programming Note

or Rx,Rx,Rx can be used to modify the PRI field; see Section 3.2.

Programming Note

When the system error handler is invoked, the PRI field may be set to an undefined value.

3.2 "or" Instruction

Setting the PPR

The *or Rx,Rx,Rx* (see Book I) instruction can be used to set PPR_{PRI} as shown in Figure . *or. Rx,Rx,Rx* does not set PPR_{PRI}.

Rx	PPR_{PRI}	Priority
31	001	very low
1	010	low
6	011	medium low
2	100	medium
5	101	medium high

Priority levels for *or Rx,Rx,Rx*

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 4.3.6 of Book III-S.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

The following forms of *or Rx,Rx,Rx* provide hints about usage of shared processor resources.

"or" Shared Resource Hints

or 27,27,27

This form of ***or*** provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released for use by other processors.

or 29,29,29

This form of ***or*** provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released until all outstanding storage accesses to caching-inhibited storage have been completed.

or 30,30,30

This form of ***or*** provides a hint that performance will probably be improved if shared resources dedicated to the executing processor are released until all outstanding storage accesses to cacheable storage for which the data is not in the cache have been completed.

Extended Mnemonics:

Additional extended mnemonics for the ***or*** hints*:*

Extended:	**Equivalent to:**	
yield	or	27,27,27
mdoio	or	29,29,29
mdoom	or	30,30,30

Programming Note

Warning: Other forms of *or Rx,Rx,Rx* that are not described in this section and in Section 4.3.3, ""or" Instruction", in Book II may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (*ori 0,0,0*) should be used.

Chapter 4. Storage Control Instructions

4.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. The virtual page sizes
2. Coherence block size
3. Reservation granule size
4. An indication of the cache model implemented (e.g., Harvard-style cache, combined cache)
5. Instruction cache size
6. Data cache size
7. Instruction cache block size
8. Data cache block size
9. Instruction cache associativity
10. Data cache associativity
11. Number of stream IDs supported for the stream variant of ***dcbt***
12. Factors for converting the Time Base to seconds
13. Maximum transaction level

If the caches are combined, the same value should be given for an instruction cache attribute and the corresponding data cache attribute.

4.2 Data Stream Control Register (DSCR) [Category: Stream]

The layout of the Data Stream Control Register (DSCR) is shown in Figure 3 below.

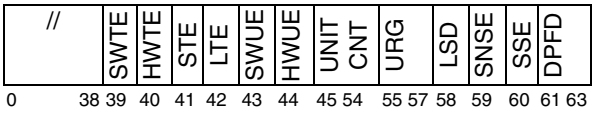


Figure 3. Data Stream Control Register

Bit(s) Description

39 ***Software Transient Enable*** (SWTE)

0 SWTE is disabled.

1 Applies the transient attribute to software-defined streams.

40 ***Hardware Transient Enable*** (HWTE)

0 HWTE is disabled.

1 Applies the transient attribute to hardware-detected streams.

41 ***Store Transient Enable*** (STE)

0 STE is disabled.

1 Applies the transient attribute to store streams.

42 ***Load Transient Enable*** (LTE)

0 LTE is disabled.

1 Applies the transient attribute to load streams.

43 ***Software Unit count Enable*** (SWUE)

0 SWUE is disabled.

1 Applies the unit count to software-defined streams.

44 ***Hardware Unit count Enable*** (HWUE)

0 HWUE is disabled.

1 Applies the unit count to hardware-detected streams.

45:54 ***Unit Count*** (UNITCNT)

Number of units in data stream.

55:57 ***Depth Attainment Urgency*** (URG)

This field indicates how quickly the prefetch depth should be reached for hardware-detected streams. Values and their meanings are as follows.

0 default
1 not urgent
2 least urgent
3 less urgent
4 medium
5 urgent
6 more urgent
7 most urgent

58 ***Load Stream Disable*** (LSD)

0 No effect.

1 Disables hardware detection and initiation of load streams.

59 ***Stride-N Stream Enable*** (SNSE)

0 No effect.
1 Enables the hardware detection and initiation of load and store streams that have a stride greater than a single cache block. Such load streams are detected only when LSD is also zero. Such store streams are detected only when SSE is also one.

61:63 ***Default Prefetch Depth*** (DPFD)

This field supplies a prefetch depth for hardware-detected streams and for software-defined streams for which a depth of zero is specified or for which ***dcbt/dcbtst*** with TH=1010 is *not* used in their description. Values and their meanings are as follows.

0 default ($LPCR_{DPFD}$)
1 none
2 shallowest
3 shallow
4 medium
5 deep
6 deeper
7 deepest

The contents of the DSCR affect how a processor handles hardware-detected and software-defined data streams. The DSCR provides the only means by which software can control or supply information for hardware-detected data streams. The DPFD, UNITCNT, and transient fields may also be used instead of the TH=01010 variant of ***dcbt*** for software-defined data streams, especially when multiple streams have these attributes in common. See Section 4.3.2, "Data Cache Instructions" on page 763, for information on streams and how software may specify them.

Programming Note

The URG, LSD, SNSE and SSE fields do not affect the initiation of streams specified using the ***dcbt*** and ***dcbtst*** instructions.

Note that even when SNSE is not set, hardware may detect Stride-N streams in intervals when they access elements that map to sequential cache blocks.

Programming Note

In order for the DSCR to apply the transient attribute to streams, at least two of the four enable bits must be set: one to choose a type of access (load or store), and one to choose a kind of prefetching (software-defined or hardware-detected).

Programming Note

The purpose of Depth Attainment Urgency is to regulate the rate of prefetch generation from the cycle at which the hardware first detects an incipient stream until the cycle when the prefetch Depth is reached. A more urgent setting will benefit applications that are dominated by short to medium length streams, because otherwise prefetching does not occur rapidly enough to benefit them. In contrast, applications that frequently cause unproductive prefetches due to stream mispredicts will benefit from a less urgent setting.

Unlike the Depth, the Depth Attainment Urgency applies only to hardware-detected streams. Furthermore, the DSCR provides the only point of control for this parameter. Software-defined streams are assumed not to have the correctness risk associated with hardware streams, and therefore are set to reach their depth relatively quickly.

Programming Note

In versions of the architecture that precede Version 2.07, ***mtspr*** specifying the DSCR caused all active and nascent data streams to cease to exist. In those versions of the architecture, the DSCR was used as an overall control mechanism to specify a single global profile for all streams. Beginning with Version 2.07, the DSCR is intended to control and accelerate the creation of new streams without disturbing existing streams.

4.3 Cache Management Instructions

The *Cache Management* instructions obey the sequential execution model except as described in Section 4.3.1.

In the instruction descriptions the statements “this instruction is treated as a *Load*” and “this instruction is treated as a *Store*” mean that the instruction is treated as a *Load* (*Store*) from (to) the addressed byte with respect to address translation, the definition of program order on page 735, storage protection, reference and change recording<S>, and the storage access ordering described in Section 1.7.1 and is treated as a *Read* (*Write*) from (to) the addressed byte with respect to debug events unless otherwise specified. (See Book III-E.)

Programming Note

Accesses that are caused by or associated with *Cache Management* instructions that are “treated as a *Load*” or “treated as a *Store*” are not subject to the special ordering rules described for SAO storage. These accesses are always performed in accordance with the weakly consistent storage model.

Some *Cache Management* instructions contain a CT field that is used to specify a cache level within a cache hierarchy or a portion of a cache structure to which the instruction is to be applied. The correspondence between the CT value specified and the cache level is shown below.

CT Field Value	Cache Level
0	Primary Cache
2	Secondary Cache

CT values not shown above may be used to specify implementation-dependent cache levels or implementation-dependent portions of a cache structure.

4.3.1 Instruction Cache Instructions

Instruction Cache Block Invalidate X-form

icbi RA,RB

31	///	RA	RB	982	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the instruction cache of this processor, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording<S> need not be done.

Special Registers Altered:

None

Programming Note

Because the instruction is treated as a *Load*, the effective address is translated using translation resources that are used for data accesses, even though the block being invalidated was copied into the instruction cache based on translation resources used for instruction fetches (see Book III).

Programming Note

The invalidation of the specified block need not have been performed with respect to the processor executing the ***icbi*** instruction until a subsequent ***isync*** instruction has been executed by that processor. No other instruction or event has the corresponding effect.

Instruction Cache Block Touch X-form

icbt CT, RA, RB

31	/	CT	RA	RB	22	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***icbt*** instruction provides a hint that the program will probably soon execute code from the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

The hint is ignored if the block is Caching Inhibited.

This instruction treated as a *Load* (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording<S> need not be done.

Special Registers Altered:

None

4.3.2 Data Cache Instructions

The Data Cache instructions control various aspects of the data cache.

TH field in the *dcbt* and *dcbtst* instructions

Described below are the TH field values for the ***dcbt*** and ***dcbtst*** instructions. For all TH field values which are not listed, the hint provided by the instruction is undefined.

TH=0b00000

If TH=0b00000, the ***dcbt/dcbtst*** instruction provides a hint that the program will probably soon access the block containing the byte addressed by EA.

TH=0b00000 - 0b00111
[Category: Cache Specification]

In addition to the hint specified above for the TH field value of 0b00000, an additional hint is provided indicating that placement of the block in the cache specified by the TH field might also improve performance. The correspondence between each value of the TH field and the cache to be specified is the same as the correspondence between each value the CT field and the cache to be specified as defined in Section 4.3. The hints corresponding to values of the TH field not supported by the implementation are undefined.

TH=0b01000 - 0b01111 [Category: Stream]

The ***dcbt/dcbtst*** instructions provide hints regarding a sequence of accesses to data elements, or indicate the expected use thereof. Such a sequence is called a "data stream", and a ***dcbt/dcbtst*** instruction in which TH is set to one of these values is said to be a "data stream variant" of ***dcbt/dcbtst***. In the remainder of this section, "data stream" may be abbreviated to "stream".

A data stream to which a program may perform *Load* accesses is said to be a "load data stream", and is described using the data stream variants of the ***dcbt*** instruction. A data stream to which a program may perform *Store* accesses is said to be a "store data stream", and is described using the data stream variants of the ***dcbtst*** instruction.

When, and how often, effective addresses for a data stream are translated is implementation-dependent.

Each data element is associated with a *unit* of storage, which is the aligned 128-byte location in storage that contains the first byte of the element. The data stream variants may be used to specify the address of the beginning of the data stream, the displacement (stride) between the first byte of successive elements, and the number of unique units of storage that are associated with all of the data elements. If the stride is specified, both the stride and the address of the first element are specified at 4 byte granularity. If the stride is not specified, the address of the first element is the address of the first unit.

Programming Note

The architecture does not provide a way to specify the size of the data elements that compose a stream. An implementation may assume some fixed size for all data elements. As a result, depending on the offset, stride, and size (and in particular whether the elements are aligned), the implementation may reduce the latency for accessing only a portion of some of the elements. A future version of the architecture may enable the specification of element size to avoid this limitation.

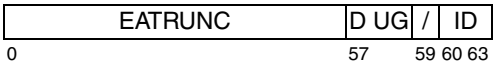
Each such data stream is associated, by software, with a stream ID, which is a resource that the processor uses to distinguish the data stream from other such data streams. The number of stream IDs is an implementation-dependent value in the range 1:16. Stream IDs are numbered sequentially starting from 0.

The encodings of the TH field and of the corresponding EA values are as follows. In the EA layout diagrams, fields shown as "/"s are reserved. These reserved fields are treated in the same manner as the corresponding case for instruction fields (see Section 1.3.3 of Book I). If a reserved value is specified for a defined EA field, or if a TH value is specified that is not explicitly defined below, the hint provided by the instruction is undefined.

TH **Description**

01000 The ***dcbt/dcbtst*** instruction provides a hint that describes certain attributes of a data stream, and may indicate that the program will probably soon access the stream.

The EA is interpreted as follows.



Bit(s) Description

0:56 ***EATRUNC***

High-order 57 bits of the effective address of the first element of the data stream. (i.e., the effective address of the first unit of the stream is EATRUNC || $^{7}0$)

57 ***Direction*** (D)

0 Subsequent elements have increasing addresses.
1 Subsequent elements have decreasing addresses.

58 ***Unlimited/GO*** (UG)

0 No information is provided by the UG field.
1 The number of elements in the data stream is unlimited, the elements are adjacent to each other, the program's need for each element of the stream is not likely to be transient, and the program will probably soon access the stream.

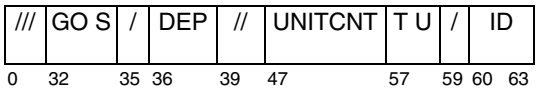
59 Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream.

01010 The ***dcbt/dcbtst*** instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon access data streams that have been described using data stream variants of the ***dcbt/dcbtst*** instruction, or will probably no longer access such data streams.

The EA is interpreted as follows. If GO=1 and S≠0b00 the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.



Bit(s) Description

0:31 Reserved

32 ***GO***

0 No information is provided by the GO field.
1 For ***dcbt***, the program will probably soon access all nascent load and store data streams that have been completely described, and will probably no longer access all other nascent load and store data streams. All other fields of the EA are ignored. ("Nascent" and "completely described" are defined below.) For ***dcbtst***, this field value holds no meaning and is treated as though it were zero.

33:34 ***Stop*** (S)

00 No information is provided by the S field.
01 Reserved
10 The program will probably no longer access the data stream (if any) associated with the specified stream ID. (All other fields of the EA except the ID field are ignored.)
11 For ***dcbt***, the program will probably no longer access the load and store data streams associated with all stream IDs. (All other fields of the EA are ignored.) For ***dcbtst***, this field value holds no meaning, and is treated as though it were 0b00.

35 Reserved

36:38 ***Depth*** (DEP)

The DEP field provides a relative estimate of how many elements ahead of the point of stream use the latency-reducing actions should go. This value reflects a comparison of the rate of consumption of the elements of the data stream and the latency to bring an arbitrary element of the stream into cache. The values are as follows.

0 default = $DSCR_{DPFD}$
1 none
2 shallowest
3 shallow
4 medium
5 deep
6 deeper
7 deepest

39:46 Reserved

47:56 ***UNITCNT***

Number of units in data stream.

57 ***Transient*** (T)

If T=1, the program's need for each element of the data stream is likely to be transient (i.e., the time interval during which the program accesses the element is likely to be short).

58 ***Unlimited*** (U)

If U=1, the number of units in the data stream is unlimited (and the UNITCNT field is ignored).

59 Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream (GO=0 and S=0b00), or stream ID associated with the data stream which the program will probably no longer access(S=0b10).

Programming Note

To maximize the utility of the Depth control mechanism, the architecture provides a hierarchy of three ways to program it. In the Server environment, the DPFD field in the LPCR is used by the provisory/ firmware to set a safe or appropriate default depth for unaware operating systems and applications. (The corresponding default in the Embedded environment is implementation specific.) The DPFD field in the DSCR may be initialized by the aware OS and overwritten by an application via the OS-provided service when per stream control is unnecessary or unaffordable. The DEP field in the EA specification when TH=0b01010 may be used by the application to specify the depth on a per-stream basis.

The number of elements ahead of the point of stream use indicated by a given depth value may differ across implementations, as may the latency to bring a given element into the cache. To achieve optimum performance, some experimentation with different depth values may be necessary.

01011 The ***dcbt/dcbtst*** instruction provides a hint that describes certain attributes of a data stream.

The EA is interpreted as follows.

Bit(s) Description

0:31 Reserved

32:49 ***Stride***

The displacement, in words, between the first byte of successive elements in the stream. The effective address of the N^{th} element in the stream is

$$(N-1)\times STRIDE\times 4$$

greater than or less than the effective address of the first element of the stream, depending on the direction specified for the stream.

50 Reserved

51:55 ***Offset***

The word-offset of the first element of the stream in its unit (i.e., the effective address of the first element of the stream is (EATRUNC || OFFSET || 0b00)).56:59Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream.

Programming Note

A program should use a ***dcbt/dcbtst*** instruction with TH=0b01011 only when the stride is larger than 128 bytes. Otherwise, consecutive units will be accessed, so the additional stream information has no benefit.

If the specified stream ID value is greater than m-1, where m is the number of stream IDs provided by the implementation, and either (a) TH=0b01000 or TH=0b01011, or (b) TH=0b01010 with GO=0 and S≠0b11, no hint is provided by the instruction.

The following terminology is used to describe the state of a data stream. Except as described in the paragraph after the next paragraph, the state of a data stream at a given time is determined by the most recently provided hint(s) for the stream.

- A data stream for which only descriptive hints have been provided (by ***dcbt/dcbtst*** instructions with TH=0b01000 and UG=0, TH=0b01010 and GO=0 and S=0b00, and/or with TH=0b01011) is said to be “nascent”. A nascent data stream for which all relevant descriptive hints have been provided (by the ***dcbt/dcbtst*** usages listed in the preceding sentence) is considered to be “completely described”. The order of descriptive hints with respect to one another is unimportant.
- A data stream for which a hint has been provided (by a ***dcbt/dcbtst*** instruction with TH=0b01000 and UG=1 or ***dcbt*** with TH=0b01010 and GO=1) that the program will probably soon access it is said to be “active”.
- A data stream that is either nascent or active is considered to “exist”.
- A data stream for which a hint has been provided (e.g., by a ***dcbt*** instruction with TH=0b01010 and S≠0b00) that the program will probably no longer access it is considered no longer to exist.

The hint provided by a ***dcbt/dcbtst*** instruction with TH=0b01000 and UG=1 implicitly includes a hint that the program will probably no longer access the data stream (if any) previously associated with the specified stream ID. The hint provided by a ***dcbt/dcbtst*** instruction with TH=0b01000 and UG=0, or with TH=0b01010 and GO=0 and S=0b00, or with TH=0b01011 implicitly includes a hint that the program will probably no longer access the *active* data stream (if any) previously associated with the specified stream ID.

If a data stream is specified without using a ***dcbt/ dcbtst*** instruction with TH=0b01010 and GO=0 and S=0b00, then the number of elements in the stream is unlimited, and the program’s need for each element of the stream is not likely to be transient. If a data stream is specified without using a ***dcbt/dcbtst*** instruction with

TH=0b01011, then the stream will access consecutive units of storage.

Interrupts (see Book III) cause all existing data streams to cease to exist. In addition, depending on the implementation, certain conditions and events may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a page.

Programming Note

To obtain the best performance across the widest range of implementations that support the data stream variants of ***dcbt/dcbtst***, the programmer should assume the following model when using those variants.

- The processor's response to a hint that the program will probably soon access a given data stream is to take actions that reduce the latency of accesses to the first few elements of the stream. (Such actions may include prefetching cache blocks into levels of the storage hierarchy that are "near" the processor.) Thereafter, as the program accesses each successive element of the stream, the processor takes latency-reducing actions for additional elements of the stream, pacing these actions with the program's accesses (i.e., taking the actions for only a limited number of elements ahead of the element that the program is currently accessing).

  The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, is to stop taking latency-reducing actions for the stream.
- A data stream having finite length ceases to exist when the latency-reducing actions have been taken for all elements of the stream.
- If the program ceases to need a given data stream before having accessed all elements of the stream (always the case for streams having unlimited length), performance may be improved if the program then provides a hint that it will no longer access the stream (e.g., by executing the appropriate ***dcbt*** instruction with TH=0b01010 and S≠0b00).
- At each level of the storage hierarchy that is "near" the processor, elements of a data stream that is specified as transient are most likely to be replaced. As a result, it may be desirable to stagger addresses of streams (choose addresses that map to different cache congruence classes) to reduce the likelihood that an element of a transient stream will be replaced prior to being accessed by the program.
- Processors that comply with versions of the architecture that do not support the TH field at all treat TH = 0b01000, 0b01010, and 0b01011 as if TH = 0b00000.
- A single set of stream IDs is shared between the ***dcbt*** and ***dcbtst*** instructions.
- On some implementations, data streams that are not specified by software may be detected by the processor. Such data streams are called "hardware-detected data streams". On some such implementations, data stream resources (resources that are used primarily to support data streams) are shared between software-specified data streams and hardware-detected data streams. On these latter implementations, the programming model includes the following.
  - Software-specified data streams take precedence over hardware-detected data streams in use of data stream resources.
  - The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, includes releasing the associated data stream resources, so that they can be used by hardware-detected data streams.

Programming Note

The latency-reducing actions taken in response to a program's hints about access to a data stream, including the depth and urgency parameters, may vary based on its behavior and on the behavior of other programs sharing platform resources, as well as on the design of the platform resources they use. Without actually changing the stream specification or DSCR parameters, the processor may adjust its actions (e.g. slow down prefetches or be more selective choosing them) based on their effectiveness and on the availability of storage bandwidth. In general, the goal of this variation is to improve overall system performance and fairness across the set of programs that share resources. There often will be a performance benefit, however, from adjusting stream specifications to the platform and co-resident programs to adjust for these actions by the processor.

Programming Note

This Programming Note describes several aspects of using the data stream variants of the ***dcbt*** and ***dcbtst*** instructions.

- A non-transient data stream having unlimited length and which will access consecutive units in storage can be completely specified, including providing the hint that the program will probably soon access it, using one ***dcbt*** instruction. The corresponding specification for a data stream having other attributes requires two or three ***dcbt/dcbtst*** instructions to describe the stream and one additional ***dcbt*** instruction to start the stream. However, one ***dcbt*** instruction with TH=0b01010 and GO=1 can apply to a set of the data streams described in the preceding sentence, so the corresponding specification for n such data streams requires 2×n to 3×n ***dcbt/dcbtst*** instructions plus one ***dcbt*** instruction. (There is no need to execute a ***dcbt/dcbtst*** instruction with TH=0b01010 and S=0b10 for a given stream ID before using the stream ID for a new data stream; the implicit portion of the hint provided by ***dcbt/dcbtst*** instructions that describe data streams suffices.)
- If it is desired that the hint provided by a given ***dcbt/dcbtst*** instruction be provided in program order with respect to the hint provided by another ***dcbt/dcbtst*** instruction, the two instructions must be separated by an ***eieio***<S> or ***mbar***<E> instruction. For example, if a ***dcbt*** instruction with TH=0b01010 and GO=1 is intended to indicate that the program will probably soon access nascent data streams described (completely) by preceding ***dcbt/dcbtst*** instructions, and is intended *not* to indicate that the program will probably soon access nascent data streams described (completely) by following ***dcbt/dcbtst*** instructions, an ***eieio***<S> or ***mbar***<E> instruction must separate the ***dcbt*** instruction with GO=1 from the preceding ***dcbt/dcbtst*** instructions, and another ***eieio***<S> or ***mbar***<E> instruction must separate that ***dcbt*** instruction from the following ***dcbt/dcbtst*** instructions.
- In practice, the second ***eieio***<S> or ***mbar***<E> described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the ***dcbt/dcbtst*** instructions and an inner loop that contains the *Load* or *Store* instructions that access the data streams, the characteristics of the inner loop and of the implementation's branch prediction mechanisms may make it highly unlikely that hints corresponding to a given iteration of the outer loop will be provided out of program order with respect to hints corresponding to the previous iteration of the outer loop. (Also, any providing of hints out of program order affects only performance, not program correctness.)
- To mitigate the effects of interrupts on data streams, it may be desirable to specify a given "logical" data stream as a sequence of shorter, component data streams. Similar considerations apply to conditions and events that, depending on the implementation, may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a virtual page.
- If it is desired to specify data streams without regard to the number of stream IDs provided by the implementation, stream IDs should be assigned to data streams in order of decreasing stream importance (stream ID 0 to the most important stream, stream ID 1 to the next most important stream, etc.). This order ensures that the hints for the most important data streams will be provided.

TH=0b10000

If TH=0b10000, the ***dcbt*** instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the program's need for the block will be transient (i.e. the time interval during which the program accesses the block is likely to be short).

Programming Note

The processor's response to the hint that access to the block will be transient is to prefetch data into the cache hierarchy in a way that minimizes the displacement of data that has not been identified as transient.

TH=0b10001

If TH=0b10001, the ***dcbt*** instruction provides a hint that the program will probably not access the block containing the byte addressed by EA for a relatively long period of time.

Data Cache Block Allocate *X-form*

dcba RA,RB
[Category: Embedded]

31	///	RA	RB	758	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

This instruction provides a hint that the program will probably soon store into a portion of the block and the contents of the rest of the block are not meaningful to the program. The contents of the block are undefined when the instruction completes. The hint is ignored if the block is Caching Inhibited.

This instruction is treated as a *Store* (see Section 4.3) except that the instruction is treated as a no-op if execution of the instruction would cause the system data storage error handler to be invoked.

Special Registers Altered:

None

Data Cache Block Touch *X-form*

dcbt RA,RB,TH [Category: Server]
dcbt TH,RA,RB [Category: Embedded]

31	TH	RA	RB	278	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbt*** instruction provides a hint that describes a block or data stream to which the program may perform a *Load* access. The instruction is also used to indicate imminent access or end of access to described load and store data streams. A hint that the program will probably soon load from a given storage location is ignored if the location is Caching Inhibited or Guarded<S>.

The only operation that is "caused" by the ***dcbt*** instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbt*** instruction (e.g., ***dcbt*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

The ***dcbt*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH≠0b01010 and TH≠0b01011, this instruction is treated as a *Load* (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording<S> need not be done.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch* instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

Extended:	**Equivalent to:**
dcbtct RA,RB,TH	dcbt for TH values of 0b00000 - 0b00111; other TH values are invalid.
dcbtds RA,RB,TH	dcbt for TH values of 0b00000 or 0b01000 - 0b01111; other TH values are invalid.
dcbtt RA,RB	dcbt for TH value of 0b10000
dcbna RA,RB	dcbt for TH value of 0b10001

Programming Notes

New programs should avoid using the ***dcbt*** and ***dcbtst*** mnemonics; one of the extended mnemonics should be used exclusively.

<S> If the ***dcbt*** mnemonic is used with only two operands, the TH operand is assumed to be 0b00000.

Processors that comply with versions of the architecture that precede Version 2.01 do not necessarily ignore the hint provided by ***dcbt*** and ***dcbtst*** if the specified block is in storage that is Guarded <S> and not Caching Inhibited.

Programming Note

See the Programming Notes at the beginning of this section.

Data Cache Block Touch for Store X-form

dcbtst RA,RB,TH [Category: Server]
dcbtst TH,RA,RB [Category: Embedded]

31	TH	RA	RB	246	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbtst*** instruction provides a hint that describes a block or data stream to which the program may perform a *Store* access, or indicates the expected use thereof. A hint that the program will soon store to a given storage location is ignored if the location is Caching Inhibited or Guarded<S>.

The only operation that is "caused by" the ***dcbtst*** instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbtst*** instruction (e.g., ***dcbtst*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

The ***dcbtst*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH≠0b01010 and TH≠0b01011, this instruction is treated as a *Store* (see Section 4.3), except that the system data storage error handler is not invoked, reference recording<S> need not be done, and change recording<S> is not done.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch for Store* instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

Extended:	**Equivalent to:**
dcbtstct RA,RB,TH	dcbtst for TH values of 0b00000 or 0b00000 - 0b00111; other TH values are invalid.
dcbtstds RA,RB,TH	dcbtst for TH values of 0b00000 or 0b01000 - 0b01111; other TH values are invalid.
dcbtstt RA,RB	dcbtst for TH value of 0b10000.

Programming Note

See the Programming Notes at the beginning of this section.

Data Cache Block set to Zero *X-form*

dcbz RA,RB

31	///	RA	RB	1014	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← block size (bytes)
m ← log2(n)
ea ← EA0:63-m || m0
MEM(ea, n) ← n0x00
```

Let the effective address (EA) be the sum (RAl0)+(RB).

All bytes in the block containing the byte addressed by EA are set to zero.

This instruction is treated as a Store (see Section 4.3).

Special Registers Altered:

None

Programming Note

dcbz does not cause the block to exist in the data cache if the block is in storage that is Caching Inhibited.

For storage that is neither Write Through Required nor Caching Inhibited, ***dcbz*** provides an efficient means of setting blocks of storage to zero. It can be used to initialize large areas of such storage, in a manner that is likely to consume less memory bandwidth than an equivalent sequence of *Store* instructions.

For storage that is either Write Through Required or Caching Inhibited, ***dcbz*** is likely to take significantly longer to execute than an equivalent sequence of *Store* instructions. For example, on some implementations dcbz for such storage may cause the system alignment error handler to be invoked; on such implementations the system alignment error handler sets the specified block to zero using Store instructions.

See Section 5.9.1 of Book III-S and Section 6.11.1 of Book III-E for additional information about ***dcbz***.

Data Cache Block Store *X-form*

dcbst RA,RB

31	///	RA	RB	54	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording<S> need not be done, and it is treated as a *Write* with respect to debug events.

Special Registers Altered:

None

Data Cache Block Flush *X-form*

dcbf RA,RB,L

31	///	L	RA	RB	86	/
0	6	9	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

L=0

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

L=1 (“dcbf local”) [Category: Server, Embedded.Phased-In]

The L=1 form of the ***dcbf*** instruction permits a program to limit the scope of the “flush” operation to the data cache of this processor. If the block containing the byte addressed by EA is in the data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 3 (“dcbf local primary”) [Category: Server, Embedded.Phased-In]

The L=3 form of the ***dcbf*** instruction permits a program to limit the scope of the “flush” operation to the primary data cache of this processor. If the block containing the byte addressed by EA is in the primary data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

For the L operand, the value 2 is reserved. The results of executing a ***dcbf*** instruction with L=2 are boundedly undefined.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording<S> need not be done, and it is treated as a *Write* with respect to debug events.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Flush* instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A. “Assembler Extended Mnemonics” on page 827. The extended mnemonics are shown below.

Extended:	Equivalent to:
dcbf RA,RB	dcbf RA,RB,0
dcbfl RA,RB	dcbf RA,RB,1
dcbflp RA,RB	dcbf RA,RB,3

Except in the ***dcbf*** instruction description in this section, references to “***dcbf***” in Books I-III imply L=0 unless otherwise stated or obvious from context; “***dcbfl***” is used for L=1 and “***dcbflp***” is used for L=3.

Programming Note

dcbf serves as both a basic and an extended mnemonic. The Assembler will recognize a ***dcbf*** mnemonic with three operands as the basic form, and a ***dcbf*** mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

dcbf with L=1 can be used to provide a hint that a block in this processor’s data cache will not be reused soon.

dcbf with L=3 can be used to flush a block from the processor’s primary data cache but reduce the latency of a subsequent access. For example, the block may be evicted from the primary data cache but a copy retained in a lower level of the cache hierarchy.

Programs which manage coherence in software must use ***dcbf*** with L=0.

4.3.3 “or” Instruction

“or” Cache Control Hint

or 26,26,26

This form of ***or*** provides a hint that stores caused by preceding *Store* and ***dcbz*** instructions should be performed with respect to other processors and mechanisms as soon as is feasible.

4.3.2.1 Obsolete Data Cache Instructions [Category: Vector]

The *Data Stream Touch* (***dst***), *Data Stream Touch for Store* (***dststst***), and *Data Stream Stop* (***dss***) instructions (primary opcode 31, extended opcodes 342, 374, and 822 respectively), which were proposed for addition to the Power ISA and were implemented by some processors, must be treated as no-ops (rather than as illegal instructions).

The treatment of these instructions is independent of whether other Vector instructions are available (i.e., is independent of the contents of MSR_{VEC}<S> (see Book III-S) or MSR_{SPV} (see Book III-E).

Programming Note

These instructions merely provided hints, and thus were permitted to be treated as no-ops even on processors that implemented them.

The treatment of these instructions is independent of whether other Vector instructions are available because, on processors that implemented the instructions, the instructions were available even when other Vector instructions were not.

The extended mnemonics for these instructions were ***dstt, dststt,*** and ***dssall***.

Programming Note

This form of the ***or*** instruction can be used to reduce latency in producer-consumer applications by requesting that modified data be made visible to other processors quickly.

Producer:

```
addi r1,r1,0x1234
sth r1,0x1000(r0) # store data value 0x1234

addi r2,r2,0x0001
stb r2,0x1002(r0) # store nonzero flag byte

or r26,r26,r26    # miso

p_loop:
lbz r2,0x1002(r0)
andi. r2,r2,0x00FF
bne p_loop        # wait for clear of flag byte
```

Consumer:

```
c_loop:
lbz r2,0x1002(r0)
andi. r2,r2,0x00FF
beq c_loop        # check for valid flag

lhz r1,0x1000(r0) # load data value

addi r2,r2,0x0000
stb r2,0x1002(r0) # clear valid flag

or r26,r26,r26    # miso
```

Engineering Note

Implementation of ***miso*** should attempt to order the hint relative to stores caused by preceding accesses. Implementations may elect to treat ***miso*** as a no-op.

Ways to implement ***miso*** include the following.

- In a store queue that has potentially different priorities for forwarding stores, the ***miso*** instruction could increase the priority of stores caused by preceding instructions.
- In a store queue that holds stores to allow for gathering, the ***miso*** instruction could release the hold and allow the stores to proceed to be performed with respect to other processors.

Extended Mnemonics:

Additional extended mnemonic for the ***or*** hint*:*

Extended:	**Equivalent to:**
miso	or 26,26,26

Programming Note

Warning: Other forms of *or Rx,Rx,Rx* that are not described in this section and in may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (*ori 0,0,0*) should be used.

4.4 Synchronization Instructions

The synchronization instructions are used to ensure that certain instructions have completed before other instructions are initiated, or to control storage access ordering, or to support debug operations.

4.4.1 Instruction Synchronize Instruction

Instruction Synchronize ***XL-form***

isync

19	///	///	///	150	/
0	6	11	16	21	31

Executing an ***isync*** instruction ensures that all instructions preceding the ***isync*** instruction have completed before the ***isync*** instruction completes, and that no subsequent instructions are initiated until after the ***isync*** instruction completes. It also ensures that all instruction cache block invalidations caused by ***icbi*** instructions preceding the ***isync*** instruction have been performed with respect to the processor executing the ***isync*** instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the ***isync*** instruction may complete before storage accesses associated with instructions preceding the ***isync*** instruction have been performed.

This instruction is context synchronizing (see Book III).

Special Registers Altered:

None

4.4.2 Load and Reserve and Store Conditional Instructions

The *Load And Reserve* and *Store Conditional* instructions can be used to construct a sequence of instructions that appears to perform an atomic update operation on an aligned storage location. See Section 1.7.3, “Atomic Update” for additional information about these instructions.

The *Load And Reserve* and *Store Conditional* instructions are fixed-point *Storage Access* instructions; see Section 3.3.1, “Fixed-Point Storage Access Instructions”, in Book I.

The storage location specified by the *Load And Reserve* and *Store Conditional* instructions must be in storage that is Memory Coherence Required if the location may be modified by another processor or mechanism. If the specified location is in storage that is Write Through Required or Caching Inhibited, the system data storage error handler is invoked for the Server environment and may be invoked for the Embedded environment.

The *Load and Reserve* instructions include an Exclusive Access hint (EH), which can be used to indicate that the instruction sequence being executed is implementing one of two types of algorithms:

Atomic Update (EH=0)
This hint indicates that the program is using a fetch and operate (e.g., fetch and add) or some similar algorithm and that all programs accessing the shared variable are likely to use a similar operation to access the shared variable for some time.

Exclusive Access (EH=1)
This hint indicates that the program is attempting to acquire a lock and if it succeeds, will perform another store to the lock variable (releasing the lock) before another program attempts to modify the lock variable.

Programming Note

The Memory Coherence Required attribute on other processors and mechanisms ensures that their stores to the reservation granule will cause the reservation created by the *Load And Reserve* instruction to be lost.

Programming Note

Because the *Load And Reserve* and *Store Conditional* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, locking, etc.; see Appendix B) that are needed by application programs. Application programs should use these library programs, rather than use the *Load And Reserve* and *Store Conditional* instructions directly.

Programming Note

EH = 1 should be used when the program is obtaining a lock variable which it will subsequently release before another program attempts to perform a store to it. When contention for a lock is significant, using this hint may reduce the number of times a cache block is transferred between processor caches.

EH = 0 should be used when all accesses to a mutex variable are performed using an instruction sequence with *Load and Reserve* followed by *Store Conditional* (e.g., emulating atomic update primitives such as "Fetch and Add;" see Appendix B). The processor may use this hint to optimize the cache to cache transfer of the block containing the mutex variable, thus reducing the latency of performing an operation such as 'Fetch and Add'.

Programming Note

Either value of the EH field is appropriate for a *Load and Reserve* instruction that is intended to establish a reservation for a subsequent ***waitrsv*** and not a subsequent *Store Conditional* instruction.

Programming Note

Warning: On some processors that comply with versions of the architecture that precede Version 2.00, executing a *Load And Reserve* instruction in which EH = 1 will cause the illegal instruction error handler to be invoked.

Load Byte And Reserve Indexed X-form [Category: Phased-In]

lbarx RT,RA,RB,EH

31	RT	RA	RB	52	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← 56^0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RAl0)+(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

This instruction creates a reservation for use by a ***stbcx.*** instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 1 byte is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the byte in storage addressed by EA regardless of the result of the corresponding ***stbcx.*** instruction.
- 1 Other programs will not attempt to modify the byte in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

Special Registers Altered:

None

Programming Note

lbarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lbarx*** mnemonic with four operands as the basic form, and a ***lbarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Halfword And Reserve Indexed X-form [Category: Phased-In]

lharx RT,RA,RB,EH

31	RT	RA	RB	116	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 2
RESERVE_ADDR ← real_addr(EA)
RT ← 48^0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RAl0)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

This instruction creates a reservation for use by a ***sthcx.*** instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 2 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the halfword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the halfword in storage addressed by EA regardless of the result of the corresponding ***sthcx.*** instruction.
- 1 Other programs will not attempt to modify the halfword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Programming Note

lharx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lharx*** mnemonic with four operands as the basic form, and a ***lharx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Word And Reserve Indexed X-form

lwarx RT,RA,RB,EH

31	RT	RA	RB	20	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 4
RESERVE_ADDR ← real_addr(EA)
RT ← 32^0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RAl0)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

This instruction creates a reservation for use by a ***stwcx.*** instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 4 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the word in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the word in storage addressed by EA regardless of the result of the corresponding ***stwcx.*** instruction.
- 1 Other programs will not attempt to modify the word in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Programming Note

lwarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lwarx*** mnemonic with four operands as the basic form, and a ***lwarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Store Byte Conditional Indexed X-form [Category: Phased-In]

stbcx. RS,RA,RB

31	RS	RA	RB	694	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 1 then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 1) ← (RS)56:63
      undefined_case  ← 0
      store_performed ← 1
    else
      if SCPM category supported then
        z ← smallest real page size supported by
            implementation
        if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
          undefined_case  ← 1
        else
          undefined_case  ← 0
          store_performed ← 0
      else
        undefined_case  ← 1
  else
    undefined_case ← 1
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 1) ← (RS)56:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 1 byte, and the real storage location specified by the ***stbcx.*** is the same as the real storage location specified by the ***lbarx*** instruction that established the reservation, $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 1 byte, and the real storage location specified by the ***stbcx.*** is not the same as the real storage location specified by the ***lbarx*** instruction that established the reservation, the following applies.

- If the Store Conditional Page Mobility category is supported, the following applies. Let z denote a naturally aligned block of real storage whose size is the smallest real page size supported by the implementation. If the real storage location specified by the ***stbcx.*** is in the same z as the real storage location specified by the ***lbarx*** instruction that established the reservation, it is undefined whether $(RS)_{56:63}$ are stored into the byte in storage addressed by EA. Otherwise, no store is performed.
- If the Store Conditional Page Mobility category is not supported, it is undefined whether $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 1 byte, it is undefined whether $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid n \mid\mid XER_{SO}$

The reservation is cleared.

Special Registers Altered:

CR0

Store Halfword Conditional Indexed X-form [Category: Phased-In]

sthcx. RS,RA,RB

31	RS	RA	RB	726	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 2 then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 2) ← (RS)48:63
      undefined_case  ← 0
      store_performed ← 1
    else
      if SCPM category supported then
        z ← smallest real page size supported by
            implementation
       if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
          undefined_case  ← 1
        else
          undefined_case  ← 0
          store_performed ← 0
      else
        undefined_case  ← 1
  else
    undefined_case ← 1
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 2) ← (RS)48:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 2 bytes, and the real storage location specified by the ***sthcx.*** is the same as the real storage location specified by the ***lharx*** instruction that established the reservation, $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 2 bytes, and the real storage location specified by the ***sthcx.*** is not the same as the real storage location specified by the ***lharx*** instruction that established the reservation, the following applies.

- If the Store Conditional Page Mobility category is supported, the following applies. Let z denote a naturally aligned block of real storage whose size is the smallest real page size supported by the implementation. If the real storage location specified by the ***sthcx.*** is in the same z as the real storage location specified by the ***lharx*** instruction that established the reservation, it is undefined whether $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA. Otherwise, no store is performed.
- If the Store Conditional Page Mobility category is not supported, it is undefined whether $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 2 bytes, it is undefined whether $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \,||\, n \,||\, XER_{SO}$

The reservation is cleared.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0

Store Word Conditional Indexed X-form

stwcx. RS,RA,RB

31	RS	RA	RB	150	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 4 then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 4) ← (RS)32:63
      undefined_case  ← 0
      store_performed ← 1
    else
      if SCPM category supported then
        z ← smallest real page size supported by
            implementation
       if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
          undefined_case  ← 1
        else
          undefined_case  ← 0
          store_performed ← 0
      else
        undefined_case  ← 1
  else
    undefined_case ← 1
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 4) ← (RS)32:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 4 bytes, and the real storage location specified by the ***stwcx.*** is the same as the real storage location specified by the ***lwarx*** instruction that established the reservation, $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 4 bytes, and the real storage location specified by the ***stwcx.*** is not the same as the real storage location specified by the ***lwarx*** instruction that established the reservation, the following applies.

- If the Store Conditional Page Mobility category is supported, the following applies. Let z denote a naturally aligned block of real storage whose size is the smallest real page size supported by the implementation. If the real storage location specified by the ***stwcx.*** is in the same z as the real storage location specified by the ***lwarx*** instruction that established the reservation, it is undefined whether $(RS)_{32:63}$ are stored into the word in storage addressed by EA. Otherwise, no store is performed.
- If the Store Conditional Page Mobility category is not supported, it is undefined whether $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 4 bytes, it is undefined whether $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid n \mid\mid XER_{SO}$

The reservation is cleared.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0

4.4.2.1 64-Bit Load and Reserve and Store Conditional Instructions [Category: 64-Bit]

Load Doubleword And Reserve Indexed X-form

ldarx RT,RA,RB,EH

31	RT	RA	RB	84	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 8
RESERVE_ADDR ← real_addr(EA)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

This instruction creates a reservation for use by a ***stdcx.*** instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 8 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding ***stdcx.*** instruction.
- 1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Programming Note

ldarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***ldarx*** mnemonic with four operands as the basic form, and a ***ldarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Store Doubleword Conditional Indexed X-form

stdcx. RS,RA,RB

31	RS	RA	RB	214	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 8 then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 8) ← (RS)
      undefined_case  ← 0
      store_performed ← 1
    else
      if SCPM category supported then
        z ← smallest real page size supported by
            implementation
       if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
          undefined_case  ← 1
        else
          undefined_case  ← 0
          store_performed ← 0
      else
        undefined_case  ← 1
  else
    undefined_case ← 1
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 8) ← (RS)
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XER_SO
else
  CR0 ← 0b00 || store_performed || XER_SO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 8 bytes, and the real storage location specified by the ***stdcx.*** is the same as the real storage location specified by the ***ldarx*** instruction that established the reservation, (RS) is stored into the doubleword in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 8 bytes, and the real storage location specified by the ***stdcx.*** is not the same as the real storage location specified by the ***ldarx*** instruction that established the reservation, the following applies.

- If the Store Conditional Page Mobility category is supported, the following applies. Let z denote a naturally aligned block of real storage whose size

is the smallest real page size supported by the implementation. If the real storage location specified by the ***stdcx.*** is in the same z as the real storage location specified by the ***ldarx*** instruction that established the reservation, it is undefined whether (RS) is stored into the doubleword in storage addressed by EA. Otherwise, no store is performed.

- If the Store Conditional Page Mobility category is not supported, it is undefined whether (RS) is stored into the doubleword in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 8 bytes, it is undefined whether (RS) is stored into the doubleword in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid n \mid\mid XER_{SO}$

The reservation is cleared.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

4.4.2.2 CR0128-bit Load and Reserve Store Conditional Instructions [Category: Load/ Store Quadword]

For ***lqarx***, the quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In big-endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

In the preferred form of the *Load Quadword* instruction RA ≠ RTp+1 and RB ≠ RTp+1.

For ***stqcx.***, the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In big-endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In little-endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

Load Quadword And Reserve Indexed X-form

lqarx RTp,RA,RB,EH

31	RTp	RA	RB	276	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 16
RESERVE_ADDR ← real_addr(EA)
RTp ← MEM(EA, 16)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by EA is loaded into RTp.

This instruction creates a reservation for use by a ***stqcx.*** instruction. A real address computed from the EA as described in Section 1.7.3.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 16 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding ***stqcx.*** instruction.
- 1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 16. If it is not, the system alignment error handler is invoked.

If RTp is odd, RTp=RA, or RTp=RB the instruction form is invalid. If RTp=RA or RTp=RB, an attempt to execute this instruction will invoke the system illegal instruction error handler. (The RTp=RA case includes the case of RTp=RA=0.)

Special Registers Altered:

None

Programming Note

lqarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lqarx*** mnemonic with four operands as the basic form, and a ***lqarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Store Quadword Conditional Indexed X-form

stqcx. RSp,RA,RB

31	RSp	RA	RB	182	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 16 then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 16) ← (RSp)
      undefined_case  ← 0
      store_performed ← 1
    else
      if SCPM category supported then
        z ← smallest real page size supported by
            implementation
        if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
          undefined_case  ← 1
        else
          undefined_case  ← 0
          store_performed ← 0
      else
        undefined_case  ← 1
  else
    undefined_case ← 1
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 16) ← (RSp)
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XER_SO
else
  CR0 ← 0b00 || store_performed || XER_SO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 16 bytes, and the real storage location specified by the ***stqcx.*** is the same as the real storage location specified by the ***lqarx*** instruction that established the reservation, (RSp) is stored into the quadword in storage addressed by EA.

If a reservation exists, the length associated with the reservation is 16 bytes, and the real storage location specified by the ***stqcx.*** is not the same as the real storage location specified by the ***lqarx*** instruction that established the reservation, the following applies.

- If the Store Conditional Page Mobility category is supported, the following applies. Let z denote a naturally aligned block of real storage whose size is the smallest real page size supported by the implementation. If the real storage location specified by the ***stqcx.*** is in the same z as the real storage location specified by the ***lqarx*** instruction that established the reservation, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA. Otherwise, no store is performed.
- If the Store Conditional Page Mobility category is not supported, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA.

If a reservation exists and the length associated with the reservation is not 16 bytes, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00\ ||\ n\ ||\ XER_{SO}$

The reservation is cleared.

EA must be a multiple of 16. If it is not, the system alignment error handler is invoked.

If RSp is odd, the instruction form is invalid.

Special Registers Altered:

CR0

4.4.3 Memory Barrier Instructions

The *Memory Barrier* instructions can be used to control the order in which storage accesses are performed. See Section 1.8, "Transactions [Category: Transactional Memory]" for a description of how the *Memory Barrier* instructions interact with transactions. Additional information about these instructions and about related aspects of storage management can be found in Book III.

Extended mnemonics for Synchronize

Extended mnemonics are provided for the *Synchronize* instruction so that it can be supported by assemblers that recognize only the ***msync***<E> mnemonic and so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A. "Assembler Extended Mnemonics" on page 827. The only reason for the ***msync***<E> mnemonic is compatibility with Book E assembler code.

Synchronize ***X-form***

sync L, E [Category: Elemental Memory Barriers]

31	///	L	/	E	///	598	/
0	6	9	11	12	16	21	31

```
if E ≠ 0b0000 then     <EMB>
  if E = 0b1xxx then enforce_barrier(mbll)
  if E = 0bx1xx then enforce_barrier(mbls)
  if E = 0bxx1x then enforce_barrier(mbsl)
  if E = 0bxxx1 then enforce_barrier(mbss)
else
   switch(L)
     case(0): hwsync
     case(1): lwsync
     case(2): ptesync<S> or hwsync<E>
```

The ***sync*** instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses that is ordered by the memory barrier depends on the value of the E and L fields. If the implementation supports the Elemental Memory Barriers category and the E field is nonzero, the L field is ignored as long as the E and L values do not create an invalid form (see Figure 4).

E≠0 [Category: Elemental Barriers]

The memory barrier provides an ordering function for one or more distinct pairings of accesses to storage that is memory Coherence Required and is neither Write Through Required nor Caching Inhibited. Each bit in the E field that has a value of 1 selects pairings, as described below.

- **E_0=1 (mbll)**
  The "load load" memory barrier is provided. Applicable pairs are all pairs a_i,b_j of such accesses in which both a_i and b_j are accesses caused by a *Load* instruction.
- **E_1=1 (mbls)**
  The "load store" memory barrier is provided. Applicable pairs are all pairs a_i,b_j of such accesses in which a_i is an access caused by a *Load* instruction and b_j is an access caused by a *Store* or ***dcbz*** instruction.
- **E_2=1 (mbsl)**
  The "store load" memory barrier is provided. Applicable pairs are all pairs a_i,b_j of such accesses in which a_i is an access caused by a *Store* or ***dcbz*** instruction and b_j is an access caused by a *Load* instruction.
- **E_3=1 (mbss)**
  The "store store" memory barrier is provided. Applicable pairs are all pairs a_i,b_j of such accesses in which both a_i and b_j are accesses caused by a *Store* or ***dcbz*** instruction.

L=0 ("heavyweight sync")

The memory barrier provides an ordering function for the storage accesses associated wth all instructions that are executed by the procesor executing the ***sync*** instruction. The applicable pairs are all pairs a_i,b_j of storage accesses in which b_j is a data access, except that if a_i is the storage access caused by an ***icbi*** instruction then b_j may be performed with respect to the processor executing the ***sync*** instruction before a_i is performed with respect to that processor.

L=1 ("lightweight sync")

The memory barrier provides an ordering function for the storage accesses caused by *Load*, *Store*, and ***dcbz*** instructions that are executed by the processor executing the ***sync*** instruction and for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a_i,b_j of storage accesses except those in which a_i is an access caused by a *Store* or ***dcbz*** instruction and b_j is an access caused by a *Load* instruction.

L=2<S> ("ptesync")

The set of storage accesses that is ordered by the memory barrier is described in Section 5.9.2 of

Book III-S, as are additional properties of the ***sync*** instruction with L=2.

The ordering done by the memory barrier is cumulative (regardless of the E and L values).

If L=0 (or L=2<S>), the ***sync*** instruction has the following additional properties.

- Executing the ***sync*** instruction ensures that all instructions preceding the ***sync*** instruction have completed before the ***sync*** instruction completes, and that no subsequent instructions are initiated until after the ***sync*** instruction completes.
- The ***sync*** instruction is execution synchronizing (see Book III). However, address translation and reference and change recording<S> (see Book III) associated with subsequent instructions may be performed before the ***sync*** instruction completes.
- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a store in set B is executed, all applicable storage accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coherence properties. The single exception is that any storage access in set A that is caused by an ***icbi*** instruction executed by the processor executing the ***sync*** instruction (P1) may not have been performed with respect to P1 (see the description of the ***icbi*** instruction on page 762).

  The cumulative properties of the barrier apply to the execution of the given instruction as they would to a load that returned a value that was the result of a store in set B.

The value L=3 is reserved.

Figure 4 shows the valid encodings of the E and L fields, as well as the resulting memory barrier for systems that support the Elemental Memory Barriers category and those that do not.

Programming Note

In Figure 4, encodings in which E_2=1 also have L=0 (***hwsync***) to ensure backward compatibility.

Assembler Note

Combinations of E and L values other than those shown in Figure 4 should be flagged as errors.

E	L	Intended Barrier for systems in which category Elemental Barriers is supported	Intended barrier for systems in which category Elemental Barriers is not supported
0001	1	mbss	lwsync
0010	0	mbsl	hwsync
0011	0	mbsl+mbss	hwsync
0100	1	mbls	lwsync
0101	1	mbls+mbss	lwsync
0110	0	mbls+mbsl	hwsync
0111	0	mbls+mbsl+mbss	hwsync
1000	1	mbll	lwsync
1001	1	mbll+mbss	lwsync
1010	0	mbll+mbsl	hwsync
1011	0	mbll+mbsl+mbss	hwsync
1100	1	mbll+mbls	lwsync
1101	1	mbll+mbls+mbss	lwsync
1110	0	mbll+mbls+mbsl	hwsync
1111	0	mbll+mbls+mbsl+mbss	hwsync
0000	0	hwsync	hwsync
0000	1	lwsync	lwsync
0000	2	ptesync<S> or hwsync<E>	ptesync<S> or hwsync<E>
(Other combinations of E and L values are invalid.)			

Figure 4. Interpretation of the E and L fields

The ***sync*** instruction may complete before storage accesses associated with instructions preceding the ***sync*** instruction have been performed.

See Section 6.11.3 of Book III-E for additional information related to ***sync*** with L=0 for the Embedded environment.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Synchronize*:

Extended:	Equivalent to:
sync	sync 0
msync<E>	sync 0
lwsync	sync 1
ptesync<S>	sync 2

Except in the ***sync*** instruction description in this section, references to "***sync***" in Books I-III imply L=0 unless otherwise stated or obvious from context; the appropriate extended mnemonics are used when other L values are intended. Throughout Books I-III, references to the L field imply E=0b0000 unless otherwise stated or obvious from context; the E field is mentioned explicitly when other E values are intended. Some programming examples and recommendations assume a common programming model that does not include the Elemental Memory Barriers category. Improved performance may be achieved through the use of elemental memory barriers in many cases.

Programming Note

Section 1.9 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

Programming Note

sync serves as both a basic and an extended mnemonic. The Assembler will recognize a ***sync*** mnemonic with one operand as the basic form, and a ***sync*** mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

The ***sync*** instruction can be used to ensure that all stores into a data structure, caused by *Store* instructions executed in a "critical section" of a program, will be performed with respect to another processor before the store that releases the lock is performed with respect to that processor; see Section B.2, "Lock Acquisition and Release, and Related Techniques" on page 831.

The memory barrier created by a ***sync*** instruction with L=1 does not order implicit storage accesses or instruction fetches. The memory barrier created by a ***sync*** instruction with L=0 (or L=2) orders implicit storage accesses and instruction fetches associated with instructions preceding the ***sync*** instruction but not those associated with instructions following the ***sync*** instruction.

In order to obtain the best performance across the widest range of implementations, the programmer should use the ***sync*** instruction with L=1, or the ***eieio***<S> or ***mbar***<E> instruction, if any of these is sufficient for his needs; otherwise he should use ***sync*** with L=0. ***sync*** with L=2<S> should not be used by application programs.

Programming Note

The functions provided by ***sync*** with L=1 are a strict subset of those provided by ***sync*** with L=0. (The functions provided by ***sync*** with L=2<S> are a strict superset of those provided by ***sync*** with L=0; see Book III.)

Enforce In-order Execution of I/O X-form

eieio
[Category: Server]

31	///	///	///	854	/
0	6	11	16	21	31

The ***eieio*** instruction creates a memory barrier (see Section 1.7.1, “Storage Access Ordering”), which provides an ordering function for the storage accesses caused by *Load*, *Store*, ***dcbz***, ***eciwx***, and ***ecowx*** instructions executed by the processor executing the ***eieio*** instruction. These storage accesses are divided into the two sets listed below. The storage access caused by an ***eciwx*** instruction is ordered as a load, and the storage access caused by a ***dcbz*** or ***ecowx*** instruction is ordered as a store.

1. Loads and stores to storage that is both Caching Inhibited and Guarded, and stores to main storage caused by stores to storage that is Write Through Required.

   The applicable pairs are all pairs a_i,b_j of such accesses.

2. Stores to storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited.

   The applicable pairs are all pairs a_i,b_j of such accesses.

The operations caused by the stream variants of the ***dcbt*** and ***dcbtst*** instructions (i.e. the providing of hints) are ordered by ***eieio*** as a third set of operations, and the operations caused by ***tlbie***<S> and ***tlbsync*** instructions (see Book III-S) are ordered by ***eieio*** as a fourth set of operations.

Each of the four sets of storage accesses or operations is ordered independently of the other three sets. The ordering done by ***eieio***'s memory barrier for the second set is cumulative; the ordering done by ***eieio***'s memory barrier for the other three sets is not cumulative.

The ***eieio*** instruction may complete before storage accesses associated with instructions preceding the ***eieio*** instruction have been performed. The ***eieio*** instruction may complete before operations caused by ***dcbt*** and ***dcbtst*** instructions preceding the ***eieio*** instruction have been performed

Special Registers Altered:

None

Memory Barrier X-form

mbar MO
[Category: Embedded]

31	MO	///	///	854	/
0	6	11	16	21	31

When MO=0, the ***mbar*** instruction creates a cumulative memory barrier (see Section 1.7.1, “Storage Access Ordering”), which provides an ordering function for the storage accesses executed by the processor executing the ***mbar*** instruction.

When MO≠0, an implementation may support the ***mbar*** instruction ordering a particular subset of storage accesses. An implementation may also support multiple, non-zero values of MO that each specify a different subset of storage accesses that are ordered by the ***mbar*** instruction. Which subsets of storage accesses that are ordered and which values of MO that specify these subsets is implementation-dependent.

The ***mbar*** instruction may complete before storage accesses associated with instructions preceding the ***mbar*** instruction have been performed. The ***mbar*** instruction may complete before operations caused by ***dcbt*** and ***dcbtst*** instructions preceding the sync instruction have been performed.

Special Registers Altered:

None

Programming Note

The ***eieio***<S> and ***mbar***<E> instructions are intended for use in doing memory-mapped I/O). Because loads, and separately stores, to storage that is both Caching Inhibited and Guarded are performed in program order (see Section 1.7.1, “Storage Access Ordering” on page 742), ***eieio***<S> or ***mbar***<E> is needed for such storage only when loads must be ordered with respect to stores.

For the ***eieio***<S> instruction, accesses in set 1, a_i and b_j need not be the same kind of access or be to storage having the same storage control attributes. For example, a_i can be a load to Caching Inhibited, Guarded storage, and b_j a store to Write Through Required storage.

If stronger ordering is desired than that provided by ***eieio***<S> or ***mbar***<E>, the ***sync*** instruction must be used, with the appropriate value in the L field.

Programming Note

The functions provided by ***eieio***<S> for its second set are a strict subset of those provided by ***sync*** with L=1.

Since ***eieio***<S> and ***mbar***<E>share the same op-code, software designed for both Server and Embedded environments must assume that only the ***eieio***<S> functionality applies since the functions provided by ***eieio*** are a subset of those provided by ***mbar*** with MO=0.

4.4.4 Wait Instruction

Wait ***X-form***

wait WC
[Category: Wait.Phased-In]

31	///	WC	///	///	62	/
0	6	9	11	16	21	31

wait
[Category: Wait.Phased-Out]

31	///	///	///	62	/
0	6	11	16	21	31

The ***wait*** instruction allows instruction fetching and execution to be suspended under certain conditions, depending on the value of the WC field. A ***wait*** instruction without the WC field is treated as a ***wait*** instruction with WC=0.

The defined values for WC are as follows.

0b00 Resume instruction fetching and execution when an interrupt occurs.

0b01 Resume instruction fetching and execution when an interrupt occurs or when a reservation made by the processor does not exist (see Section 1.7.3). It is implementation-dependent whether this WC value is supported or ***wait*** with this WC value is treated as a no-op.

0b10 Resume instruction fetching and execution when an interrupt occurs or when an implementation-specific condition exists. It is implementation-dependent whether this WC value is supported or is treated as reserved.

0b11 This WC value is treated as a no-op.

If WC=0, or if WC=1 and a reservation made by the processor exists when the wait instruction is executed, or if WC=2 and the associated implementation-specific condition does not exist when the ***wait*** instruction is executed, the following applies.

- Instruction fetching and execution is suspended.
- Once the ***wait*** instruction has completed, the NIA will point to the next sequential instruction.
- Instruction fetching and execution resumes when any of the following conditions are met.
  - An interrupt occurs.
  - WC=1 and a reservation made by the processor does not exist.
  - WC=2 and the associated implementation-specific condition exists.

When the ***wait*** instruction is executed, if WC=1 and a reservation made by the processor does not exist, or if WC=2 and the associated implementation-specific condition exists, the instruction is treated as a no-op.

Programming Note

On implementations which do not support the wait instruction with WC=0b10, the behavior for non-support (treated as reserved) differs from the non-support of the other non-zero WC values (treated as no-ops). The possibility of boundedly undefined behavior such as causing the system illegal instruction error handler to be invoked is meant to discourage the use of WC=0b10 in programs that are intended to be portable.

Only programs that are implementation-aware should use WC=0b10.

Engineering Note

Causing the system illegal instruction error handler to be invoked if an attempt is made to execute ***wait*** with WC=0b10 on an implementation that does not support that form of ***wait*** facilitates the debugging of software.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Wait*:

Extended:	Equivalent to:
wait	wait 0
waitrsv	wait 1
waitimpl	wait 2

Programming Note

The ***wait*** instruction with WC=0b00 can be used in verification test cases to signal the end of a test case. The encoding for the instruction is the same in both Big-Endian and Little-Endian modes.

Programming Note

The ***wait*** instruction may be useful as the primary instruction of an “idle process” or the completion of processing for a cooperative processor. However, overall system performance may be better served if the ***wait*** instruction is used by applications only for idle times that are expected to be short.

Programming Note

Execution of a ***wait*** instruction indicates that no further instruction fetching will occur until the condition(s) associated with the WC field value for the instruction take place. The main purpose of the ***wait*** instruction is to enable power savings. ***wait*** frees computational resources which might be allocated to another program or converted into power savings.

If an interrupt causes resumption of instruction execution, the interrupt handler will return to the instruction after the ***wait***.

Engineering Note

In previous versions of the architecture the ***wait*** instruction was context synchronizing.

Programming Note

The ***wait*** instruction is not execution synchronizing and does not cause a memory barrier. ***waitrsv*** behavior relative to a preceding *Load and Reserve* instruction or *Store Conditional* instruction has a data dependency on the reservation. When execution proceeds past ***waitrsv*** as the result of another processor storing to the reservation granule, a subsequent load from the same storage location may return stale data. It is also possible that execution could proceed past the ***waitrsv*** for other reasons such as the occurrence of an interrupt. There are no architecturally defined means to determine what terminated the ***wait***. Moreover, even if software were to attempt to determine what caused the ***wait*** to terminate, by the time the check occurred, both causes (interrupt and storage modification) might be true. Software must be designed to deal with the various causes of ***wait*** termination. In general, if the program that performed ***wait*** does not see the new value of the storage location for which the reservation was held, it should re-establish the reservation by repeating the *Load and Reserve* instruction, and then perform another ***waitrsv***.

The following code waits for a device to update a memory location and assumes that r3 contains the address of the word to be updated. This assumes that software has already set this word to zero and is waiting for the device to update the word to a non-zero value.

```
loop:
   lwarx  r4,0,r3 # load and reserve
   cmpwi  r4,0    # exit if nonzero
   bne-   exit
   waitrsv        # wait for reservation loss
   b      loop
exit: ...
```

The ***b*** instruction results in re-execution of the ***waitrsv*** if instruction execution had resumed for some reason other than loss of the reservation made by the processor. This branch instruction is also necessary because the reservation might have been lost for reasons other than the device updating the memory location addressed by r3. Also, even if the device updated this memory location, the ***lwarx*** and ***waitrsv*** instructions may need to be re-executed until the ***lwarx*** returns the current data.

Programming Note

A ***wait*** instruction without the WC field is treated as a ***wait*** instruction with WC=0b00 because older processors that comply with Power ISA 2.06 do not support the WC field.

Chapter 5. Transactional Memory Facility [Category: Transactional Memory]

5.1 Transactional Memory Facility Overview

This chapter describes the registers and instructions that make up the transactional memory (TM) facility. Transactional memory is a shared-memory synchronization construct allowing an application to perform a sequence of storage accesses that appear to occur atomically with respect to other threads.

A set of instructions, special-purpose registers, and state bits in the MSR (see Book III) are used to control a transactional facility that is associated with each hardware thread. A ***tbegin.*** instruction is used to initiate transactional execution, and a ***tend.*** instruction is used to terminate transactional execution. Loads and stores that occur between the ***tbegin.*** and ***tend.*** instructions appear to occur atomically. An implementation may prematurely terminate transactional execution for a variety of reasons, rolling back all transactional storage updates that have been made by the thread since the ***tbegin.*** was executed, and rolling back the contents of a subset of the thread's Book I registers to their contents before the ***tbegin.*** was executed. In the event of such premature termination, control is transferred to a software failure handler associated with the transaction, which may then retry the transaction or choose an alternate path depending on the cause of transaction failure. A transaction can be explicitly aborted via a set of *conditional abort* instructions and an *unconditional abort* instruction, ***tabort.***. A ***tsr.*** instruction is used to suspend or resume transactional execution, while allowing the transaction to remain active.

Programming Note

A ***tbegin.*** should always be followed immediately by a ***beq*** as the first instruction of the failure handler, that branches to the main body of the failure handler. The failure handler should always either retry the transaction or use non-transactional code to perform the same operation. (The number of retries should be limited to avoid the possibility of an infinite loop. The limit could be based on the perceived permanence / transience of the failure.) A failure handler policy which includes trying a different transaction before returning to the one that failed may fail to make forward progress.

Programming Note

In code that may be executed transactionally, conditional branches should hint in favor of successful transactional execution where such a distinction exists. For example, the branch immediately following ***tbegin.*** should hint that the branch is very likely not to be taken. As another example, consider a method of coding a failure handler that executes the body of a transaction non-transactionally by branching past the TM control instructions (e.g. ***tsuspend.***). Branches that bypass the TM control instructions should also hint that the branch is very likely not to be taken. These predictions will improve the efficiency of transactional execution, and may also help prevent the addition of spurious accesses to the transactional footprint.

Programming Note

The architecture does not include a “fairness guarantee” or a “forward progress” guarantee for transactions. If two processors repeatedly conflict with one another in an attempt to complete a transaction, one of the two may always succeed while the other may always fail. If two processors repeatedly conflict with one another in an attempt to complete a transaction, both may always fail, depending on the details of the transaction. This is different from the behavior of a typical locking routine, in which one or the other of the competitors will generally get the lock.

Transactions performed using this facility are “strongly atomic”, meaning that they appear atomic with respect to both transactional and non-transactional accesses performed by other threads. Transactions are isolated from reads and writes performed by other threads; i.e. transactional reads and writes will not appear to be interleaved with the reads and writes of other threads.

Nesting of transactions is supported using a form of nesting called “flattened nesting,” in which transactions that are initiated during transactional execution are subsumed by the pre-existing transaction. Consequently, the effects of a nested transaction do not become visible until the outer transaction commits, and if a nested transaction fails, the entire set of transactions (outer as well as nested) is rolled back, and control is transferred to the outer transaction’s failure handler. The memory barriers created by ***tbegin.*** and ***tend.*** and the integrated cumulative memory barrier that are described in Section 1.8, “Transactions [Category: Transactional Memory]” are only created for outer transactions and not for any transactions nested within them.

References to *Store* instructions, and stores, include ***dcbz*** and the storage accesses that it causes.

Rollback-Only Transactions

Rollback-Only Transactions (ROTs) differ from nornal transactions in that they are speculative but not atomic. They are initated by a unique variant of ***tbegin.*** They may be nested with other ROTs or with normal transactions. When a normal transaction is nested within a ROT, the behavior from the normal ***tbegin.*** until the end of the outer transaction is characteristic of a normal transaction. Although subject to failure from storage conflicts, the typical cause of ROT failure is via a ***Tabort*** variant that is executed after the program detects an error in its (software) speculation. Except where specifically differentiated or where differences follow from specific differentiation, the following description applies to ROTs as well as normal transactions.

5.1.1 Definitions

Commit: A transaction is said to commit when it successfully completes execution. When a transaction is committed, its transactional accesses become irrevocable, and are made visible to other threads. A transaction completes by either commiting or failing.

Checkpointed registers: The set of registers that are saved to the “checkpoint area” when a transaction is initiated, and restored upon transaction failure, is a subset of the architected register state, consisting of the General Purpose Registers, Floating-Point Registers, Vector Registers, Vector-Scalar Registers, and the following Special Registers and fields: CR fields other than CR0, LR, CTR, FPSCR, AMR, PPR, VRSAVE, VSCR, DSCR, and TAR. The checkpointed registers include all problem-state writable registers with the exception of CR0, EBBHR, EBBRR, BESCR, the performance monitor registers, and the Transactional Memory registers. With the exception of updates of CR0, and the Transactional Memory registers, explicit updates of registers that are not included in the set of checkpointed registers are disallowed in Transactional state (i.e. will cause the transaction to fail), but are permitted in Suspended state. Suspended state modifications of these registers will not be rolled back in the event of transaction failure. (Modifications of Transactional Memory registers are only permitted in Non-transactional state. Attempts to modify Transactional Memory registers in other than Non-transactional state will cause a TM Bad Thing type Program interrupt.)

Programming Note

CR0, and the Transactional Memory registers (TFHAR, TEXASR, TFIAR) are not saved, or restored when the transaction fails, because they are modified as a side effect of transaction failure (so restoring them would lose information needed by the failure handler). The performance monitor registers, and the event-based branching registers (BESCR, EBBHR, EBBRR) are not saved or restored because saving and restoring them would add significant implementation complexity and is not needed by software. Also, these registers, except EBBHR, can be modified asynchronously by the processor, so restoring them when the transaction fails could cause loss of information.

Transactional accesses: Data accesses that are caused by an instruction that is executed when the thread is in the Transactional state (see Section 5.2) are said to be “transactional,” or to have been “performed transactionally.” The set of accesses caused by a committed normal transaction is performed as if it were a single atomic access. That is, it is always performed in its entirety with no visible fragmentation. The sets performed by normal transactions are thus serialized: each happens in its entirety in some order, even

when that order is not specified in the program or enforced between processors. Until a transaction commits, its set of transactional accesses is provisional, and will be discarded should the transaction fail. The set of transactional accesses is also referred to as the "transactional footprint."

Non-transactional accesses: Storage accesses performed in the existing Power storage model are said to be "non-transactional." In contrast to transactional storage accesses, there is no provision of atomicity across multiple non-transactional accesses. Non-transactional storage updates are not discarded in the event of a transaction failure.

Outer transaction: A transaction that is initiated from the Non-transactional state is said to be an outer transaction. A ***tbegin.*** instruction that initiates an outer transaction is sometimes referred to as an "outer ***tbegin.***." Similarly, a ***tend.*** instruction with A=0 that ends an outer transaction is sometimes referred to as an "outer ***tend.***"

Nested Transaction: A transaction that is initiated while already executing a transaction is said to be "nested" within the pre-existing transaction. The set of active nested transactions forms a stack growing from the outer transaction. A ***tend.*** with A=0 will remove the most recently nested transaction from the stack.

Failure: A transaction failure is an exceptional condition causing the transactional footprint to be discarded, the checkpointed registers to be reverted to their pre-transactional values, and the failure handler to get control.

Failure handler: A failure handler is a software component responsible for handling transaction failure. On transaction failure, hardware redirects control to the failure handler associated with the outer transaction.

Conflict: A transactional memory access is said to conflict with another transactional or non-transactional access if both accesses reference the same cache block, and at least one of them is a store. If two transactions make conflicting accesses, at least one of them will fail. If a transaction fails as a result of a conflict with a store, the store may have been executed by another processor or may have been executed in Suspended state by the processor with the failing transaction. For a ROT, no conflict is caused if the ROT performs a load and another program performs a non-transactional store to the same block.

A transactional memory access is said to conflict with a ***tlbie*** if the storage location being accessed is in the page the translation for which is being invalidated by the tlbie. For a ROT, no conflict is caused if the access is a load.

A Suspended state cache control instruction is said to cause a conflict if it would cause the destruction of a transactional update or if it would make a transactional update visible to another thread.

5.2 Transactional Memory Facility States

The transactional memory facility supports several modes of operation, referred to in this document as the "transaction state." These states control the behavior of storage accesses made during the transaction and the handling of transaction failure. Changes to transaction state affect all transactions currently using the transactional facility on the affected thread: the outer transaction as well as any nested transactions, should they exist.

Non-transactional: The default, initial state of execution; no transaction is executing. The transactional facility is available for the initiation of a new transaction.

Transactional: This state is initiated by the execution of a ***tbegin.*** instruction in the Non-transactional state. Storage accesses (data accesses) caused by instructions executed in the Transactional state are performed transactionally. Other storage accesses associated with instructions executed in the Transactional state (instruction fetches, implicit accesses) are performed non-transactionally. In the event of transaction failure, failure is recorded as defined in Section 5.3.2, and control is transferred to the failure handler as described in Section 5.3.3.

Suspended: The Suspended execution state is explicitly entered with the execution of a ***tsuspend.*** form of ***tsr.*** instruction during a transaction, the execution of a ***trechkpt.*** instruction from Non-transactional state, or as a side-effect of an interrupt while in the Transactional state. Storage accesses and accesses to SPRs that are not part of the checkpointed registers are performed non-transactionally; they will be performed independently of the outcome of the transaction. The initiation of a new transaction is prevented in this state. In the event of transaction failure, failure recording is performed as defined in Section 5.3.2, but failure handling is usually deferred until transactional execution is resumed (see Section 5.3.3 for details).

Until failure occurs, *Load* instructions that access storage locations that were transactionally written by the same thread will return the transactionally written data. After failure is detected, but before failure handling is performed, such loads may return either the transactionally written data, or the current non-transactional contents of the accessed location. The ***tcheck*** instruction can be used to determine whether any previous such loads may have returned non-transactional contents.

Suspended state *Store* instructions that access cache blocks that have been accessed transactionally (due to load or store) by the same thread may cause the transaction to fail.

Programming Note

The intent of the Suspended execution state is to temporarily escape from transactional handling when transactional semantics are undesirable. Examples of such cases include storage updates that should be retained in the event of transactional failure, which is useful for debugging, interthread communication, the access of Caching-Inhibited storage, and the handling of interrupts. In the event of transaction failure during the Suspended execution state, failure handling is deferred until transactional execution is resumed, allowing the block of Suspended state code to complete its activities.

Programming Note

During Suspended state execution, accessing cache blocks that have been transactionally accessed by the same thread prior to entering Suspended state requires special care, because failure may occur due to uncontrollable events such as interactions with other threads or the operating system. Up until a transaction fails, loads from transactionally modified cache blocks will return the transactionally modified data. However once the transaction fails, the loads may return either the transactionally updated version of storage, or the most recent non-transactional version. Stores to transactionally modified blocks may or may not cause the thread's transaction to fail.

Table 1 enumerates the set of Transactional Memory instructions and events that can cause changes to the transaction state. Transaction states are abbreviated N (Non-transactional), T (Transactional), and S (Suspended). (Interrupts, and the ***rfebb***, ***rfid***, ***hrfid***, and ***mtmsrd*** instructions, can also cause changes to the transaction state; see Book III.)

Programming Note

tbegin. in Suspended state merely updates CR0. When ***tbegin.*** is followed by ***beq***, this will result in a transfer to the failure handler. Nothing more severe (e.g. an interrupt) is required. The failure handler for a transaction for which initiation may be attempted in Suspended state should test CR0 to determine whether tbegin. was executed in Suspended state. If so, it should attempt to emulate the transaction non-transactionally. (This case can arise, for example, if a transaction enters Suspended state and then calls a library routine that independently attempts to use transactions.)

Notice that, although a failure handler runs in Non-transactional state when reached because the transaction has failed, it runs in Suspended state for the case discussed in this Programmng Note.)

Instr/ Event State	*tbegin.*	*tend.*	Abort caused by *tabort.* and conditional *tabort.* variants	*tsuspend.*	*tresume.*	Failure	treclaim.	trechkpt.
N	T	N[2]	N[2]	N[2]	N[2]	Not applicable	N[6]	S[7]
T	T	N, if outer transaction or A=1 form; otherwise T	N[3,4]	S	T	N[3,4]	N[3]	S[6]
S	S[1]	S[6]	S[3]	S[2]	T[5]	S[3]	N[3]	S[6]

Notes
1. **CR0 updated indicating transactional initiation was unsuccessful, due to a pre-existing transaction occupying the transactional facility.**
2. **Execution of these operations does not affect transaction state, allowing for the instructions to be used in software modules called from Non-transactional, Transactional, and Suspended paths.**
3. **If failure recording has not previously occurred, failure recording is performed as defined in Section 5.3.2.**
4. **Failure handling is performed as defined in Section 5.3.3.**
5. **If failure has occurred during Suspended execution, failure handling will be performed sometime after the execution of *tresume*, and no later than the set of events listed in Section 5.3.3.**
6. **Generate TM Bad Thing type Program interrupt.**
7. **If $TEXASR_{FS}$=0, generate a TM Bad Thing type Program interrupt.**

Table 1: Transaction state transitions caused by TM instructions and transaction failure

5.2.1 The TDOOMED Bit

The status of an active transaction is summarized by a transaction doomed bit (TDOOMED) that resides in an implementation-dependent location. When 0, it indicates that the active transaction is valid, meaning that it remains possible for the transaction to commit successfully, if failure does not occur before committing. When 1 it indicates that transaction failure has already occurred for the transaction.

The TDOOMED bit is set to 0 upon the successful initiation of an outer transaction by ***tbegin.***. It is set to 1 when failure occurs or as a result of executing ***trechkpt.***. When failure occurs, TDOOMED is set to 1 before any other effects of the transaction failure (recording the failure in TEXASR, rollback of transactional stores, over-writing of the transactionally accessed locations by a conflicting store, etc.) are visible to software executing on the processor that executed the transaction. In Non-transactional state, the value of TDOOMED is undefined.

5.3 Transaction Failure

5.3.1 Causes of Transaction Failure

A transaction failure is said to be “externally-induced” if the failure is caused by a thread other than the transactional thread. Likewise, a transaction failure is said to be “self-induced” if the failure is caused by the transactional thread itself.

For self-induced failure as a result of attempting to execute an instruction that is forbidden in the Transactional state, a Privileged Instruction type of Program Interrupt takes precedence over transaction failure. (For example, an attempt to execute ***stdcix*** in Transactional state and problem state will result in a Privileged Instruction type of Program interrupt.) Transaction failure takes precedence over all other interrupt types. The relevant instructions are listed in the fourth bullet of the second set of bullets below and the first bullet in the third set of bullets below.

In general, a ROT will not fail in the following scenarios when the failure is specified as a conflict on a transactional access and the access is a load.

Transactions will fail for the following externally-induced causes

- Conflict with transactional access by another thread
- Conflict with non-transactional access by another thread

- In either of the previous two cases, if a successful *Store Conditional* would have conflicted, but the *Store Conditional* is not successful, it is implementation-dependent whether a conflict is detected
- Conflict with a translation invalidation caused by a ***tlbie*** performed by another thread

Transactions will fail for the following self-induced causes

- Termination caused by the execution of ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction.
- Transaction level overflow, defined as an attempt to execute ***tbegin.*** when the transaction level is already at its maximum value
- Footprint overflow, defined as an attempt to perform a storage access in Transactional state which exceeds the capacity for tracking transactional accesses.
- Execution of the following instructions while in the Transactional state: ***doze, sleep, nap, rvwinkle, icbi, dcbf, dcbi, dcbst, [h]rfid, rfebb, mtmsr[d], mtsle, mtsr, mtsrin, msgsnd, msgsndp, msgclr, msgclrp, slbie, slbia, slbmte, slbfee,*** and ***tlbie[l].*** (These instructions are considered to be *disallowed* in Transactional state.) The disallowed instruction is not executed; failure handling occurs before it has been executed.

  Programming Note

  Note that execution of a Power Saving instruction in Suspended state causes a TM Bad Thing type Program interrupt.

- Execution, while in Transactional state, of ***mtspr*** specifying an SPR that is not part of the checkpointed registers and is not a Transactional Memory SPR. The ***mtspr*** is not executed; failure handling occurs before it has been executed. (Modification of XER_{FXCC} and CR_{CR0} are allowed, but the changes will not be rolled back in the event of transaction failure.)
- Conflict caused by a Suspended state store to a block that was previously accessed transactionally. If the store would have been performed by a successful *Store Conditional* instruction, but the *Store Conditional* instruction does not succeed, it is implementation-dependent whether a conflict is detected.
- Conflict caused by a Suspended state ***tlbie*** that specifies a translation that was previously used transactionally. (This case will be recorded as a translation invalidation conflict because it may be hard to differentiate from a conflict caused by a ***tlbie*** performed by another thread and because it is highly likely to be a transient failure.)

For each of the following potential causes, the transaction will fail if the absence of failure would compromise transaction semantics; otherwise, whether the transaction fails is undefined.

- Execution of the following instructions while in the Transactional state: ***eciwx, ecowx, lbzcix, ldcix, lhzcix, lwzcix, stbcix, stdcix, sthcix, stwci***. The disallowed instruction is not executed; failure handling occurs before it has been executed. (These instructions are considered to be disallowed in Transactional state if they cause transaction failure in Transactional state.) Execution of these instructions in the Suspended state is allowed and does not cause transaction failure.
- Execution of the following instruction in the Transactional state: ***wait***. The disallowed instruction is not executed; failure handling occurs before it has been executed. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Execution of the following instruction in the Suspended state: ***wait***. The disallowed instruction is treated as a no-op; failure recording occurs. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Access of a disallowed type while in the Transactional state: Caching Inhibited, Write Through Required, and Memory Coherence not Required for data access; Caching-Inhibited for instruction fetch. The disallowed access is not performed; failure handling occurs such that the instruction that would cause (or be associated with, for instruction fetch) the disallowed access type appears not to have been executed. Accesses of this type in the Suspended state are allowed and do not cause transaction failure.
- Instruction fetch from a block that was previously written transactionally (reported as a unique cause that includes both self-induced and externally-induced instances)

- ***dcbf***, ***dcbi***, or ***icbi*** specifying a block that was previously accessed transactionally, in either of the following cases.

  Programming Note

  Note that ***dcbf*** with L=3 should never compromise transactional semantics, but it is still permitted to cause transaction failure in Suspended state and it is disallowed in Transactional state.

  - the instruction (***dcbf***, ***dcbi***, or ***icbi***) is executed in Suspended state on the processor executing the transaction (self-induced conflict)
  - the instruction is executed by another processor (externally-induced conflict)
- ***dcbst*** specifying a block that was previously written transactionally, in either of the following cases.
  - ***dcbst*** is executed in Suspended state on the processor executing the transaction (self-induced conflict)

- ***dcbst*** is executed by another processor (externally-induced conflict)
- Cache eviction of a block that was previously accessed transactionally

Transactions may also fail due to implementation-specific characteristics of the transactional memory mechanism.

Programming Note

WARNING: Software should not depend for its correct execution on the behavior (whether or not the relevant transaction fails) of the cases described in the preceding set of bullets. The behavior is likely to vary from design to design. Such a dependence would impact the software's portability without any tangible advantage.

Programming Note

Because the atomic nature of a transaction implies an apparent delay of its component accesses until they can be performed in unison, the use of cache control instructions to manage cache residency and/or the performing of storage accesses may have unexpected consequences. Although they may not cause transaction failure directly, their use in a transaction is strongly discouraged.

If an instruction or event does not cause transaction failure, it behaves as defined in the architecture.

The set of failure causes and events are further classified as "precise" and "imprecise" failure causes. All externally induced events are imprecise, and all self-induced events are precise with the exception of the following cases:

- Self-induced conflicts caused by instruction fetch
- Self-induced conflicts caused by footprint overflow
- Self-induced conflicts in Suspended state (because failure handling is deferred in Suspended state).

When failure recording and handling occur (as defined in Section 5.3.2 and 5.3.3) for a precise failure, they will occur precisely according to the sequential execution model, adhering to the following rules:

1. Effects of the failure occur such that all instructions preceding the instruction causing the failure appear to have completed with respect to the executing thread.
2. The instruction causing the failure may appear not to have begun execution (except for causing the failure), or may have completed, depending on the failure cause.
3. Architecturally, no subsequent instruction has begun execution.

Failure handling for imprecise failure types is guaranteed to occur no later than the execution of ***tend.*** with A=1 or $TEXASR_{TL}$ =1. Failure recording for imprecise failure types is guaranteed to occur no later than failure handling. Any operation that can cause imprecise failure if performed in-order can also cause imprecise failure if performed out-of-order.

Programming Note

Because instruction fetch from a transactionally written block may result in failure, it is recommended that transactionally accessed data and transactionally accessed instructions not be co-located within a single block.

Programming Note

The architecture does not detect and cause transaction failure for translation invalidations to transactionally accessed pages or segments, when the translation invalidation is caused by instructions other than ***tlbie*** (i.e. ***slbie, slbia, tlbiel, tlbia***). Consequently, software is responsible for terminating transactions in circumstances where such local translation invalidations may affect a local transaction.

5.3.2 Recording of Transaction Failure

When transaction failure occurs, information about the cause and circumstances of failure are recorded in SPRs associated with the transactional facility. Failure recording is performed a single time per transaction that fails, controlled by the state of the TEXASR failure summary (FS) bit; when 0, FS indicates that failure recording has not already been performed, and is therefore permissible.

The following RTL function specifies the actions taken during the recording of transaction failure:

```
TMRecordFailure(FailureCause)
                    #FailureCause is 32-bit cause
code
if TEXASR_FS = 0
   if failure IA known then
      TFIAR ← CIA
      TEXASR_37 ← 1
   else
      TFIAR ← approximate instruction address
      TEXASR_37 ← 0
   TEXASR_0:31 ← FailureCause
   if MSR_TS=0b01 then TEXASR_Suspended ← 1
   if environment = Embedded then
     TEXASR_PRIVILEGE ← ¬MSR_GS || MSR_PR
     TFIAR_PRIVILEGE ← ¬MSR_GS || MSR_PR
   else
     TEXASR_PRIVILEGE ← MSR_HV || MSR_PR
     TFIAR_PRIVILEGE ← MSR_HV || MSR_PR
  TEXASR_FS ← 1
  TDOOMED ← 1
```

When failure recording occurs, the TEXASR and TFIAR SPRs are set indicating the source of failure. When possible, TFIAR is set to the effective address of the instruction that caused the failure, and $TEXASR_{37}$ is set to 1 indicating that the contents of TFIAR are exact. When the instruction address is not known exactly, an approximate value is placed in TFIAR and $TEXASR_{37}$ is set to 0. TEXASR bits 0:31 are set indicating the cause of the failure, and the $TEXASR_{Suspended}$, $TEXASR_{Privilege}$, and $TFIAR_{Privilege}$ fields are set indicating the machine state in which the failure was recorded. $TEXASR_{TL}$ is unchanged. The TDOOMED bit is set to 1.

Programming Note

TFIAR is intended for use in the debugging of transactional programs by identifying the source of transaction failure. Because TFIAR may not always be set exactly, software should test $TEXASR_{37}$ before use; if zero, the contents of TFIAR are an approximation.

5.3.3 Handling of Transaction Failure

Discarding of the transactional footprint may begin immediately after detection of failure and, except in the case of an abort in Suspended state, may continue until the rest of failure handling is complete. However, the timing of the rest of failure handling is dependent on the state of the transactional facility. In the case of an abort in Suspended state, the transactional footprint is discarded immediately, despite that the rest of failure handling is deferred.

In Transactional state, failure handling may occur immediately, but an implementation is free to delay handling until one of the following failure synchronizing events occurs in Transactional state.

- An abort caused by the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, or ***tabortwci.*** instruction.
- The execution of a ***treclaim.*** instruction.
- An attempt, in Transactional state, to execute a disallowed instruction, perform an access of a disallowed type, or execute an ***mtspr*** instruction that specifies an SPR that is not part of the checkpointed registers and is not a Transactional Memory SPR.
- Nesting level overflow.
- An attempt to transition from Transactional to Suspended state caused by ***tsuspend.*** or by an interrupt or event.
- An attempt to commit a transaction, caused by the execution of ***tend.*** with A = 1 or when $TEXASR_{TL}$ = 1.

When a failure synchronizing event occurs in Transactional state, the processor waits until all preceding Transactional and Suspended state loads have been performed with respect to all processors and mechanisms and all failures that have occurred up to that point have been recorded. Then failure handling occurs if a failure has been recorded; otherwise, processing of the failure synchronizing event continues. If failure is caused by the failure synchronizing event, failure handling occurs immediately.

When failure handling occurs, checkpointed registers are reverted to their pre-transactional values, the transactional footprint is discarded if it has not previously been discarded, and any resources occupied by the transaction are discarded. If the failure is not caused by ***treclaim.***, the following things occur. CR0 is set to 0b101 || 0. The transaction state is set to Non-transactional, and control flow is redirected to the instruction address stored in TFHAR. If the failure is caused by ***treclaim.***, CR0 is not set to indicate failure and the transaction's failure handler is not invoked.

The following RTL function specifies the actions taken during the handling of transaction failure:

```
TMHandleFailure()
   If the transactional footprint has not previously been discarded
      Discard transactional footprint
   Revert checkpointed registers to pre-transactional values
   Discard all resources related to current transaction
   MSR_TS ← 0b00                    #Non-transactional
   If failure was not caused by treclaim.,
     NIA ← TFHAR
     CR0 ← 0b101 || 0
```

Upon failure detected in Suspended state from causes other than the execution of a ***treclaim.*** instruction, failure handling is deferred until the transaction is resumed. Once resumed, failure handling will occur no later than the set of failure synchronizing events listed above. Upon failure in Suspended state caused by ***treclaim.***, failure handling is immediate (but CR0 is not set to indicate failure and the transaction's failure handler is not invoked).

Programming Note

A *Load* instruction executed immediately after ***treclaim.*** or a conditional or unconditional *Abort* instruction is guaranteed not to load a transactional storage update.

5.4 Transactional Memory Facility Registers

The architecture is augmented with three Special Purpose Registers in support of transactional memory.

TFHAR stores the effective address of the software failure handler used in the event of transaction failure. TFIAR is used to inform software of the exact location of the transaction failure, when possible. TEXASR contains a transaction level indicating the nesting depth of an active transaction, as well as an indicator of the cause of transaction failure and some machine state when the transaction failed. These registers can be written only when in Non-transactional state.

5.4.1 Transaction Failure Handler Address Register (TFHAR)

The Transaction Failure Handler Address Register is a 64-bit SPR that records the effective address of a software failure handler used in the event of transaction failure. Bits 62:63 are reserved.

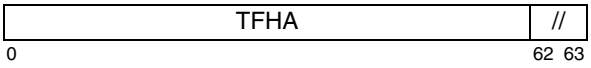


Figure 5. Transaction Failure Handler Address Register (TFHAR)

This register is written with the NIA for the ***tbegin.*** as a side-effect of the execution of an outer ***tbegin.*** instruction (***tbegin.*** executed in the Non-transactional state).

5.4.2 Transaction EXception And Summary Register (TEXASR)

The Transaction EXception And Summary Register is a 64-bit register, containing a transaction level ($TEXASR_{TL}$) and summary information for use by transaction failure handlers. Bits 0:31 are called the *failure cause* in the instruction descriptions.

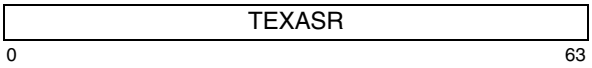


Figure 6. Transaction EXception And Summary Register (TEXASR)

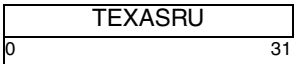


Figure 7. Transaction EXception And Summary Register Upper (TEXASRU)

Bit(s Description

0:6 ***Failure Code***
The Failure Code is copied from the ***tabort.*** or ***treclaim.*** source operand. When set, TFIAR is exact.

7 ***Failure Persistent***
The failure is likely to recur on each execution of the transaction. This bit is a hint. It is set to 1 for causes in bits 8:11, copied from the ***tabort.*** or ***treclaim.*** source operand when RA is nonzero, and set to 0 for all other failure causes.

Programming Note

The Failure Persistent bit may be viewed as an eighth bit in the failure code in that both fields are supplied by the least significant byte of RA and software may use all eight to differentiate among the cases for which it performs an abort or reclaim. However, software is expected to organize its cases so that bit 7 predicts the persistence of the case.

Programming Note

Warning: Software must not depend on the value of the Failure Persistent bit for correct execution. The number of retries for a transient failure should be counted, and a limit set after which the program will perform the operation non-transactionally. In the analysis of failures, consideration should be given to the fact that speculative execution can cause unexpected behavior.

8 ***Disallowed***
The instruction, SPR, or access type is not permitted. When set, TFIAR is exact. See Section 5.3.1, “Causes of Transaction Failure”.

Programming Note

An instruction fetch to storage that is Caching Inhibited, while nominally disallowed, will be reported as Implementation-specific (bit 15). This choice was made because it seems like a relatively unlikely programming error, and there is a significant chance that data from an external conflict (store by another thread) could indirectly cause a wild branch to storage that is Caching Inhibited.

9 ***Nesting Overflow***
The maximum transaction level was exceeded. When set, TFIAR is exact.

10 ***Footprint Overflow***
The tracking limit for transactional storage accesses was exceeded. When set, TFIAR is an approximation.

Programming Note

Note that transactional footprint tracking resources may be shared by multiple programs executing concurrently. Depending on the circumstances, this failure cause may or may not be persistent.

11 ***Self-Induced Conflict***
A self-induced conflict occurred in Suspended state, due to one of the following: a store to a block that was previously accessed transactionally; a ***dcbf***, ***dcbi,*** or ***icbi*** specifying a block that was previously accessed transactionally; a ***dcbst*** specifying a block that was previously written transactionally; or a ***tlbie*** that specifies a translation that was previously used transactionally. When set, TFIAR may be exact.

12 ***Non-Transactional Conflict***
A conflict occurred with a non-transactional access by another processor. When set, TFIAR is an approximation.

13 ***Transaction Conflict***
A conflict occurred with another transaction. When set, TFIAR may be exact.

14 ***Translation Invalidation Conflict***
A conflict occurred with a TLB invalidation. When set, TFIAR is an approximation.

15 ***Implementation-specific***
An implementation-specific condition caused the transaction to fail. Such conditions are transient and the value in the TFIAR may be exact.

16 ***Instruction Fetch Conflict***
An instruction fetch (by this or another thread) was performed from a block that was previously written transactionally. Such conditions are transient and the value in the TFIAR may be exact.

17:30 Reserved for future failure causes

31 ***Abort***
Termination was caused by the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction. When due to ***tabort.*** or ***treclaim.***, bits in $TEXASR_{0:7}$ are user-supplied. When set, TFIAR is exact.

32 ***Suspended***
When set to 1, the failure was recorded in Suspended state. When set to 0, the failure was recorded in Transactional state.

33 Reserved

34:35 ***Privilege***
The thread was in this privilege state when the failure was recorded. For the Embedded environment, this was the value $\neg MSR_{GS} \,||\, MSR_{PR}$ when the failure was recorded. For the Server environment, this was the value $MSR_{HV} \,||\, MSR_{PR}$ when the failure was recorded.

36 ***Failure Summary (FS)***
Set to 1 when a failure has been detected and failure recording has been performed.

37 ***TFIAR Exact***
Set to 1 when the value in the TFIAR is exact. Otherwise the value in the TFIAR is approximate.

38 ***ROT***
Set to 1 when a ROT is initiated. Set to zero when a non-ROT ***tbegin.*** is executed.

39 Reserved

40:51 Reserved

52:63 ***Transaction Level (TL)***
Transaction level (nesting depth + 1) for the active transaction, if any; otherwise 0 if the most recently executed transaction completed successfully, or the transaction level at which the most recently executed transaction failed if the most recently executed transaction did not complete successfully.

Programming Note

A value of 1 corresponds to an outer transaction. A value greater than 1 corresponds to a nested transaction.

The transaction level in $TEXASR_{TL}$ contains an unsigned integer indicating whether the current transaction is an outer transaction, or is nested, and if nested, its depth. The maximum transaction level supported by a given implementation is of the form $2^t - 1$. The value of *t* corresponding to the smallest maximum is 4; the value of *t* corresponding to the largest maximum is 12. This value is tied to the "Maximum transaction level" parameter useful for application programmers, as specified in Section 4.1. The high-order 12-*t* bits of $TEXASR_{TL}$ are treated as reserved.

Transaction failure information is contained in $TEXASR_{0:37}$. The fields of TEXASR are initialized upon the successful initiation of a transaction from the Non-transactional state, by setting $TEXASR_{TL}$ to 1, indicating an outer transaction, and all other fields to 0.

When transaction failure is recorded, the failure summary bit $TEXASR_{FS}$ is set to 1, indicating that failure has been detected for the active transaction and that failure recording has been performed. $TEXASR_{0:31}$ are set indicating the source of the failure. Exactly one of bits 8 through 31 will be set indicating the instruction or event that caused failure. In the event of failure due to

the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction, $TEXASR_{31}$ is set to 1, and, for ***tabort.*** and ***treclaim.***, a software defined failure code is copied from a register operand to $TEXASR_{0:7}$. $TEXASR_{Suspended}$ indicates whether the transaction was in the Suspended state at the time that failure occurred. The inverse of the value of MSR_{GS} and the value of MSR_{PR} for the Embedded environment or the values of MSR_{HV} and MSR_{PR} for the Server environment at the time that failure occurs are copied to $TEXASR_{34}$ and $TEXASR_{35}$, respectively. In some circumstances, the failure causing instruction address in TFIAR may not be exact. In such circumstances, $TEXASR_{37}$ is set to 0 indicating that the contents of TFIAR are not exact; otherwise $TEXASR_{37}$ is set to 1.

Programming Note

The transaction level contained in $TEXASR_{TL}$ should be interpreted by software as follows:

When in the Transactional or Suspended state, this field contains an unsigned integer representing the transaction level of the active transaction, with 1 indicating an outer transaction, and a number greater than 1 indicating a nested transaction. The nesting depth of the active transaction is $TEXASR_{TL} - 1$.

When in the Non-transactional state, $TEXASR_{TL}$ contains 0 if the last transaction committed successfully, otherwise it contains the transaction level at which the most recent transaction failed.

Programming Note

The Privilege bits in TEXASR represent the state of the machine at the point when failure occurs. This information may be used by problem-state software to determine whether an unexpected hypervisor or operating system interaction was responsible for transaction failure. This information may be useful to operating systems or hypervisors when restoring register state for failure handling after the transactional facility was reclaimed, to determine which of the operating system or the hypervisor has retained the pre-transactional version of the checkpointed registers.

5.4.3 Transaction Failure Instruction Address Register (TFIAR)

The Transaction Failure Instruction Address Register is a 64-bit SPR that is set to the exact effective address of the instruction causing the failure, when possible. Bits 62:63 contain the privilege state when the failure was recorded. For the Embedded environment, this was the value $\neg MSR_{GS} \parallel MSR_{PR}$ when the failure was recorded. For the Server environment, this was the value $MSR_{HV} \parallel MSR_{PR}$ when the failure was recorded.

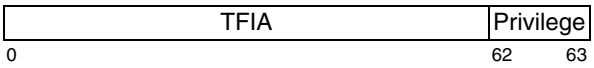


Figure 8. Transaction Failure Instruction Address Register (TFIAR)

In certain cases, the exact address may not be available, and therefore TFIAR will be an approximation. An approximate value will point to an instruction near the instruction that was executing at the time of the failure. TFIAR accuracy is recorded in an Exact bit residing in $TEXASR_{37}$.

5.5 Transactional Facility Instructions

Similar to the *Floating-Point Status and Control Register* instructions, modifications of transaction state caused by the execution of *Transactional Memory* instructions or by failure handling synchronize the effects of exception-causing floating-point instructions executed by a given processor. Executing a Transactional Memory instruction, or invocation of the failure handler, ensures that all floating-point instructions previously initiated by the given processor have completed before the transaction state is modified, and that no subsequent floating-point instructions are initiated by the given processor until the transaction state has been modified. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the transaction state is modified.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the transaction state is modified.
- No subsequent floating-point instruction that alters the settings of any FPSCR bits is initiated until the transaction state has been modified.

(Floating-point Storage Access instructions are not affected.)

Transaction Begin *X-form*

tbegin. R

31	A	//	R	///	///	654	1
0	6	7	10	11	16	21	31

```
ROT  ← R
CR0  ← 0 || MSR_TS || 0

if MSR_TS = 0b00 then                #Non-transactional
   TEXASR ← 0x000000000 || 0b00 || ROT || 0b0 ||
0x000001
   TFHAR ← CIA + 4
   TDOOMED ← 0
   MSR_TS ← 0b10
   checkpoint area ← (checkpointed registers)
   if not ROT and the transaction succeeds then
     enforce_barrier(mbll)
     enforce_barrier(mbls)
     enforce_barrier(mbsl)
     enforce_barrier(mbss)
else if MSR_TS = 0b10 then           #Transactional
   if TEXASR_TL=TL_max then
      cause ← 0x01400000
      TMRecordFailure(cause)
      TMHandleFailure()
   else
      TEXASR_TL ← TEXASR_TL + 1
      if (TEXASR_ROT=1) & (not ROT)
                      & the transaction succeeds
         enforce_barrier(mbll)
         enforce_barrier(mbls)
         enforce_barrier(mbsl)
         enforce_barrier(mbss)
         TEXASR_ROT ← 0
```

The ***tbegin.*** instruction initiates execution of a transaction, either an outer transaction or a nested transaction, as described below.

An outer transaction is initiated when ***tbegin.*** is executed in the Non-transactional state. If R=0 and the transaction is successful, a memory barrier is inserted equivalent to that produced by a *sync* instruction with E=0b1111. (See Section 4.4.3.) TEXASR and TFHAR are initialized, and the TDOOMED bit is set to 0. A nested transaction is initiated when ***tbegin.*** is executed in the Transactional state unless the transaction level is already at its maximum value, in which case failure recording is performed with a failure cause of 0x01400000 and failure handling is performed. When initiating a nested transaction, the transaction level held in $TEXASR_{TL}$ is incremented by 1, and if $TEXASR_{ROT}$ =1 but R=0, and the transaction succeeds, a memory barrier is inserted equivalent to that produced by a ***sync*** instruction with E=0b1111 and $TEXASR_{ROT}$ is turned off. The effects of a nested transaction will not be visible until the outer transaction commits, and in the event of failure, the checkpointed registers are reverted to the pre-transactional values of the outer transaction. Initiation of a transaction is unsuccessful when in the Suspended state.

When successfully initiated, transactional execution continues until the transaction is terminated using a ***tend.***, ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.***, or ***treclaim.*** instruction, suspended using a ***tsr*** instruction, or failure occurs. Upon transaction failure while in the Transactional state, transaction failure recording and failure handling are performed as defined in Section 5.3. Upon transaction failure while in the Suspended state, failure recording is performed as defined in Section 5.3.2, but failure handling is usually deferred.

CR0 is set as follows.

CR0	*Description*
000 || 0	Transaction initiation successful, unnested (Transaction state of Non-transactional prior to ***tbegin.***)
010 || 0	Transaction initiation successful, nested (Transaction state of Transactional prior to ***tbegin.***)
001 || 0	Transaction initiation unsuccessful, (Transaction state of Suspended prior to ***tbegin.***)

Other than the setting of CR0, ***tbegin.*** in the Suspended state is treated as a no-op.

The use of the A field is implementation specific.

Special Registers Altered

CR0 TEXASR TFHAR TS

Programming Note

When a transaction is successfully initiated, and failure subsequently occurs, control flow will be redirected to the instruction following the *tbegin.* instruction. When failure handling occurs, as described in Section 5.3.3, CR0 is set to 0b101 || 0. Consequently, instructions following ***tbegin.*** should also expect this value as an indication of transaction failure. Most applications will follow ***tbegin.*** with a conditional branch predicated on $CR0_2$; code at this target is responsible for handling the transaction failure.

Transaction End *X-form*

tend. A

31	A	//	/	///	///	686	1
0	6	7	10	11	16	21	31

```
CR0  ← 0b0 || MSR_TS || 0

if MSR_TS = 0b10 then           #Transactional
   if A = 1 | TEXASR_TL = 1 then
      if (TDOOMED) then
         TMHandleFailure()
      else
         if not TEXASR_ROT
           insert integrated cumulative
           memory barrier
         Commit transaction
         TEXASR_TL ← 0
          Discard all resources related to current
transaction
          MSR_TS ← 0b00          #Non-transactional
          if not TEXASR_ROT
           enforce_barrier(mbll)
           enforce_barrier(mbls)
           enforce_barrier(mbsl)
           enforce_barrier(mbss)
   else TEXASR_TL ← TEXASR_TL - 1  # nested
```

The A=0 variant of ***tend.*** supports nested transactions, in which the transaction is committed only if the execution of ***tend.*** completes an outer transaction. Execution of this variant by a nested transaction ($TEXASR_{TL} > 1$) causes $TEXASR_{TL}$ to be decremented by 1. The A=1 variant of ***tend.*** unconditionally completes the current outer transaction and all nested transactions.

When the ***tend.*** instruction completes an outer transaction, transaction commit is predicated on the TDOOMED bit. If TDOOMED is 1, failure handling occurs as defined in Section 5.3.3. If TDOOMED is 0, the transaction is committed, and $TEXASR_{TL}$ is set to 0. In both cases, the transaction state is set to Non-transactional.

When the ***tend.*** instruction commits a transaction, it atomically commits its writes to storage. If $TEXASR_{ROT}$=0, the integrated cumulative memory barrier is inserted prior to the creation of the aggregate store, and a memory barrier is inserted equivalent to that produced by a ***sync*** instruction with E=0b1111 after the aggregate store. (See Section 4.4.3.) If the transaction has failed prior to the execution of ***tend.,*** no storage updates are performed and no memory barrier is inserted. In either case (success or failure), all resources associated with the transaction are discarded.

If the transaction succeeds, Condition Register field 0 is set to 0 || MSR_{TS} || 0. If the transaction fails, CR0 is set to 0b101 || 0.

Other than the setting of CR0, ***tend.*** in Non-transactional state is treated as a no-op. If an attempt is made to execute ***tend.*** in Suspended state, a TM Bad Thing type Program interrupt occurs.

Special Registers Altered

CR0 TEXASR TS

Extended Mnemonics

Extended mnemonics for transaction end.

Extended:	***Equivalent To:***
tend.	***tend.*** 0
tendall.	***tend.*** 1

Programming Note

When an outer ***tend.*** or a ***tend.*** with A=1 is executed in the Transactional state, the CR0 value 0b101 || 0 will never be visible to the instruction that immediately follows ***tend.***, because in the event of failure the failure handler will have been invoked not later than the completion of the ***tend.*** instruction.

Transaction Abort *X-form*

tabort. RA

31	///	RA	///	910	1
0	6	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0

if MSR_TS = 0b10 | MSR_TS = 0b01 then
#Transactional, or Suspended
   if RA = 0 then cause ←0x00000001
   else           cause ← GPR(RA)_56:63 || 0x000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard the transactional footprint
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                 #Transactional
      TMHandleFailure()
```

The ***tabort.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. When in the Transactional state or the Suspended state the ***tabort.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to $GPR(RA)_{56:63}$ || 0x000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3(this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabort.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TEXASR TFIAR TS

Transaction Abort Word Conditional

X-form

tabortwc. TO,RA,RB

31	TO	RA	RB	782	1
0	6	11	16	21	31

```
a ← EXTS((RA)32:63)
b ← EXTS((RB)32:63)
abort ← 0

CR0 ←  0 || MSRTS || 0

if (a < b)  & TO0 then abort ← 1
if (a > b)  & TO1 then abort ← 1
if (a = b)  & TO2 then abort ← 1
if (a u< b) & TO3 then abort ← 1
if (a >u b) & TO4 then abort ← 1

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSRTS = 0b10 then                #Transactional
      TMHandleFailure()
```

The ***tabortwc.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register $RA_{32:63}$ are compared with the contents of register $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the ***tabortwc.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortwc.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TEXASR TFIAR TS

Transaction Abort Word Conditional Immediate

X-form

tabortwci. TO,RA,SI

31	TO	RA	SI	846	1
0	6	11	16	21	31

```
a ← EXTS((RA)32:63)
abort ← 0

CR0 ←  0 || MSRTS || 0


if a < EXTS(SI)  & TO0 then abort ← 1
if a > EXTS(SI)  & TO1 then abort ← 1
if a = EXTS(SI)  & TO2 then abort ← 1
if a u< EXTS(SI) & TO3 then abort ← 1
if a >u EXTS(SI) & TO4 then abort ← 1

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSRTS = 0b10 then                #Transactional
      TMHandleFailure()
```

The ***tabortwci.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register $RA_{32:63}$ are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended then the ***tabortwci.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortwci.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TEXASR TFIAR TS

Transaction Abort Doubleword Conditional

X-form

tabortdc. TO,RA,RB

31	TO	RA	RB	814	1

0	6	11	16	21	31

```
a ← ( RA )
b ← ( RB )
abort ← 0

CR0 ← 0 || MSR_TS || 0

if (a < b)  & TO_0 then abort ← 1
if (a > b)  & TO_1 then abort ← 1
if (a = b)  & TO_2 then abort ← 1
if (a u< b) & TO_3 then abort ← 1
if (a >u b) & TO_4 then abort ← 1

if abort & (MSR_TS = 0b10 | MSR_TS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                 #Transactional
      TMHandleFailure()
```

The ***tabortdc.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the ***tabortdc.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3(this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortdc.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TEXASR TFIAR TS

Transaction Abort Doubleword Conditional Immediate *X-form*

tabortdci. TO,RA, SI

31	TO	RA	SI	878	1
0	6	11	16	21	31

```
a ← (RA)
abort ← 0

CR0 ← 0 || MSR_TS || 0

if a < EXTS(SI)  & TO_0 then abort ← 1
if a > EXTS(SI)  & TO_1 then abort ← 1
if a = EXTS(SI)  & TO_2 then abort ← 1
if a u< EXTS(SI) & TO_3 then abort ← 1
if a >u EXTS(SI) & TO_4 then abort ← 1

if abort & (MSR_TS = 0b10 | MSR_TS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                 #Transactional
      TMHandleFailure()
```

The ***tabortdci.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended then the ***tabortdci.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortdci.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TEXASR TFIAR TS

Transaction Suspend or Resume *X-form*

tsr. L

31	///	L	///	///	750	1
0	6	10	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0
if L = 0 then
   if MSR_TS = 0b10 then              #Transactional
      MSR_TS ← 0b01                    #Suspended
else
   if MSR_TS = 0b01                    #Suspended
      MSR_TS ← 0b10                    #Transactional
```

The ***tsr.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. Based on the value of the L field, two variants of ***tsr.*** are used to change the transaction state.

If L = 0, and the transaction state is Transactional, the transaction state is set to Suspended.

If L = 1, and the transaction state is Suspended, the transaction state is set to Transactional.

Other than the setting of CR0, the execution of ***tsr.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TS

Programming Note

When resuming a transaction that has encountered failure while in the Suspended state, failure handling is performed after the execution of ***tresume.*** and no later than the next failure synchronizing event.

Extended Mnemonics

Extended mnemonics for Transaction Suspend or Resume.

Extended:	***Equivalent To:***
tsuspend.	***tsr***. 0
tresume.	***tsr***. 1

Transaction Check *X-form*

tcheck BF

31	BF	//	///	///	718	/
0	6	9	11	16	21	31

```
if MSR_TS = 0b10 | MSR_TS = 0b01 then   #Transactional
                                        #or Suspended
  for each load caused by an instruction following
  the outer tbegin and preceding this tcheck
    if (Load instruction was executed in T state
       with TEXASR_ROT=0 or accessing a location
       previously stored transactionally) |
       (Load instruction was executed in S state
        with TEXASR_ROT=0 and accessed a location
        previously accessed transactionally)|
       (Load instruction was executed in S state
        with TEXASR_ROT=1 and accessed a location
        previously stored transactionally)
    then wait until load has been performed with
    respect to all processors and mechanisms
CR field BF ← TDOOMED || MSR_TS || 0
```

If the transaction state is Transactional or Suspended, the ***tcheck*** instruction ensures that all loads that are caused by instructions that follow the outer *tbegin.* instruction and precede the ***tcheck*** instruction and satisfy one of the following properties, have been performed with respect to all processors and mechanisms.

- The load is caused by an instruction that was executed in Transactional state, either while $TEXASR_{ROT}$=0 or accessing a location previously stored transactionally.
- The load is caused by an instruction that was executed in Suspended state while $TEXASR_{ROT}$=0 and accesses a location that was accessed transactionally.
- The load is caused by an instruction that was executed in Suspended state while $TEXASR_{ROT}$=1 and accesses a location that was stored transactionally.

The ***tcheck*** instruction then copies the TDOOMED bit into bit 0 of CR field BF, copies MSR_{TS} to bits 1:2 of CR field BF, and sets bit 3 of CR field BF to 0.

Other than the setting of CR field BF, execution of ***tcheck*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR field BF

Programming Note

One use of the ***tcheck*** instruction in Suspended state is to determine whether preceding loads from transactionally modified locations have returned the data the transaction stored. (If the transaction has failed, some of the loads may have returned a more recent value that was stored by a conflicting store, or may have returned the pre-transaction contents of the location.). It is important to use tcheck. between any Suspended state loads that might access transactionally modified locations and subsequent computation using the Suspended-state-loaded data. Otherwise, corrupt data could cause problems such as wild branches or infinite loops.

Another use of ***tcheck*** in Suspended state is to determine whether the contents of storage, as seen in Suspended state, are consistent with the transaction succeeding -- e.g., whether no location that has been accessed transactionally (stored transactionally, for ROTs), and has been seen in Suspended state, has been subject to a conflict thus far. (A location is seen in Suspended state either by being loaded in Suspended state or by being loaded in Transactional state and the value (or a value derived therefrom) passed, in a register, into Suspended state.)

A use of ***tcheck*** in Transactional state is to determine whether the transaction still has the potential to succeed.

Note that ***tcheck*** provides an instantaneous check on the integrity of a subset of the accesses performed within a transaction. ***tcheck*** is not a failure synchronizing mechanism. Even if no accesses follow the ***tcheck***, there may still be latent failures that haven't been recorded, for example caused by accesses that ***tcheck*** does not wait for, by external conflicts that will happen in the future, or simply by time of flight to the failure detection mechanism for operations that have already been performed.

Programming Note

The ***tcheck*** instruction can return 1 in bit 0 of CR field BF before the failure has been recorded in TEXASR and TFIAR.

Programming Note

The ***tcheck*** instruction may cause pipeline synchronization. As a result, programs that use ***tcheck*** excessively may perform poorly.

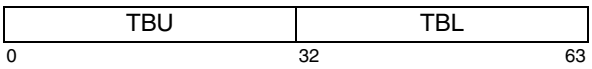
Chapter 6. Time Base

6.1 Time Base Overview

The time base facilities include a Time Base and an Alternate Time Base which is category: Alternate Time Base. The Alternate Time Base is analogous to the Time Base except that it may count at a different frequency and is not writable.

6.2 Time Base

The Time Base (TB) is a 64-bit register (see Figure 9) containing a 64-bit unsigned integer that is incremented periodically as described below.



Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 9. Time Base

The Time Base monotonically increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1); at the next increment its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The suggested frequency at which the time base increments is 512 MHz, however, variation from this rate is allowed provided the following requirements are met.

- The contents of the Time Base differ by no more than +/- four counts from what they would be if they incremented at the required frequency.
- Bit 63 of the Time Base is set to 1 between 30% and 70% of the time over any time interval of at least 16 counts.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Programming Note

If the operating system initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

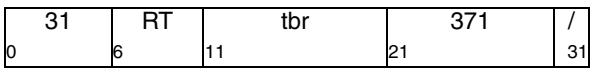
Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from 2^{64}-1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

6.2.1 Time Base Instructions

Move From Time Base ***XFX-form***

mftb RT,TBR
[Category: Phased-Out]



This instruction behaves as if it were an ***mfspr*** instruction; see the ***mfspr*** instruction description in Section 3.3.16 of Book I.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Move From Time Base*:

Extended:	Equivalent to:
mftb Rx	mftb Rx,268 mfspr Rx,268
mftbu Rx	mftb Rx,269 mfspr Rx,269

Programming Note

New programs should use ***mfspr*** instead of ***mftb*** to access the Time Base.

Programming Note

mftb serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

Programming Note

The ***mfspr*** instruction can be used to read the Time Base on all processors that comply with Version 2.01 of the architecture or with any subsequent version.

It is believed that the ***mfspr*** instruction can be used to read the Time Base on most processors that comply with versions of the architecture that precede Version 2.01. Processors for which ***mfspr*** cannot be used to read the Time Base include the following.

- 601
- POWER3

(601 implements neither the Time Base nor ***mftb***, but depends on software using ***mftb*** to read the Time Base, so that the attempt causes the Illegal Instruction error handler to be invoked and thereby permits the operating system to emulate the Time Base.)

Programming Note

Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base increments at the constant rate of 512 MHz. (Note, however, that programs should allow for the possibility that some implementations may not increment the least-significant 4 bits of the Time Base at a constant rate.) What is wanted is the pair of 32-bit values comprising a POSIX standard clock:[1] the number of whole seconds that have passed since 00:00:00 January 1, 1970, UTC, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- The integer constant *ticks_per_sec* contains the value 512,000,000, which is the number of times the Time Base is updated each second.
- The integer constant *ns_adj* contains the value

$$\frac{1{,}000{,}000{,}000}{512{,}000{,}000} \times 2^{32} / 2 = 4194304000$$

  which is the number of nanoseconds per tick of the Time Base, multiplied by 2^{32} for use in ***mulhwu*** (see below), and then divided by 2 in order to fit, as an unsigned integer, into 32 bits.

When the processor is in 64-bit mode, The POSIX clock can be computed with an instruction sequence such as this:

```
mfspr  Ry,268  # Ry = Time Base
lwz    Rx,ticks_per_sec
divdu  Rz,Ry,Rx   # Rz = whole seconds
stw    Rz,posix_sec
mulld  Rz,Rz,Rx   # Rz = quotient * divisor
sub    Rz,Ry,Rz   # Rz = excess ticks
lwz    Rx,ns_adj
slwi   Rz,Rz,1 # Rz = 2 * excess ticks
mulhwu Rz,Rz,Rx   # mul by (ns/tick)/2 * 2^32
stw    Rz,posix_ns# product[0:31] = excess ns
```

For the Embedded environment when the processor is in 32-bit mode, it is not possible to read the Time Base using a single instruction. Instead, two instructions must be used, one of which reads TBL and the other of which reads TBU. Because of the possibility of a carry from TBL to TBU occurring between the two reads, a sequence such as the following must be used to read the Time Base.

```
loop:
   mfspr  Rx,TBU  # load from TBU
   mfspr  Ry,TB   # load from TB
   mfspr  Rz,TBU  # load from TBU
   cmp    cr0,0,Rx,Rz# check if 'old'='new'
   bne    loop    #branch if carry occurred
```

Non-constant update frequency

In a system in which the update frequency of the Time Base may change over time, it is not possible to convert an isolated Time Base value into time of day. Instead, a Time Base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change via an interrupt (see Book III), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of *ticks_per_sec* for the new frequency, and save the time of day, Time Base value, and tick rate. Subsequent calls to compute Time of Day use the current Time Base Value and the saved value.

1. Described in POSIX Draft Standard P1003.4/D12, *Draft Standard for Information Technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API) - Amendment 1: Real-time Extension* [*C Language*]. Institute of Electrical and Electronics Engineers, Inc., Feb. 1992.

6.3 Alternate Time Base [Category: Alternate Time Base]

The Alternate Time Base (ATB) is a 64-bit register (see Figure 9) containing a 64-bit unsigned integer that is incremented periodically. The frequency at which the integer is updated is implementation-dependent.

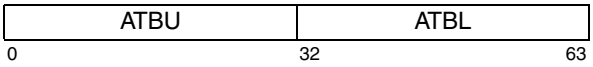


Figure 10. Alternate Time Base

The ATBL register is an aliased name for the ATB.

The Alternate Time Base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no explicit indication (such as an interrupt; see Book III) that this has occurred.

The Alternate Time Base is accessible in both user and supervisor mode. The counter can be read by executing a ***mfspr*** instruction specifying the ATB (or ATBL) register, but cannot be written. A second SPR register ATBU, is defined that accesses only the upper 32 bits of the counter. Thus the upper 32 bits of the counter may be read into a register by reading the ATBU register.

The effect of entering a power-savings mode or of processor frequency changes on counting in the Alternate Time Base is implementation-dependent.

Chapter 7. Event-Based Branch Facility [Category: Server]

7.1 Event-Based Branch Overview

The Event-Based Branch facility allows application programs to enable hardware to change the effective address of the next instruction to be executed when certain events occur to an effective address specified by the program.

The operation of the Event-Based Branch facility is summarized as follows:

- The Event-Based Branch facility is available only when the system program has made it available. See Section 9.5 of Book III-S for additional information.
- When the Event-Based Branch facility is available, event-based branches are caused by event-based exceptions. Event-based exceptions can be enabled to occur by setting bits in the Event Control field of the BESCR.
- When an event-based exception occurs, the bit in the BESCR control field corresponding to the event-based exception is set to 0 and the bit in the Event Status field in the BESCR corresponding to the event-based exception is set to 1.
- If the global enable bit in the BESCR is set to 1 when any of the bits in the status field are set to 1 (i.e. when an event-based exception exists), an event-based branch occurs.
- The event-based branch causes the global enable bit to be set to 0, causes instruction fetch and execution to continue at the effective address contained in the EBBHR, and causes the TS field of the BESCR to indicate the transactional state of the processor when the event-based branch occurred. If the processor was in transactional state when the event-based branch occurred, it is put into suspended state. The EBBRR is set to the effective address of the instruction that would have attempted to execute next if no event-based branch had occurred.
- The event-based branch handler performs the necessary processing in response to the event, and then executes an ***rfebb*** instruction in order to resume execution at the instruction that would have been executed next when the event-based branch occurred. The ***rfebb*** instruction also restores the processor to the transactional state indicated by $BESCR_{TS}$.

Additional information about the Event-Based Branch facility is given in Section 3.4 of Book III-S.

Programming Note

Since system software controls the availability of the Event-Based Branch facility (see Section 9.5 of Book III-S), an interface must be provided that enables applications to request access to the facility and determine when it is available.

Programming Note

In order to initialize the Event-Based Branch facility for Performance Monitor event-based exceptions, software performs the following operations.

- Software requests control of the Event-Based Branch facility from the system program.
- Software requests the System Program to initialize the Performance Monitor as desired.
- Software sets the EBBHR to the effective address of the event-based branch handler.
- Software enables Performance Monitor event-based exceptions in the BESCR Control Field by setting $BESCR_{PME}$ to 1. $BESCR_{PMEO}$ should also be set to 0.
- Software sets the GE bit in the BESCR to enable event-based branches.

7.2 Event-Based Branch Registers

7.2.1 Branch Event Status and Control Register

The Branch Event Status and Control Register (BESCR) is a 64-bit register that contains control and status information about the Event-Based Branch facility.

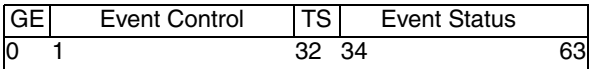


Figure 11. Branch Event Status and Control Register (BESCR)

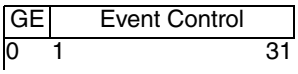


Figure 12. Branch Event Status and Control Register Upper (BESCRU)

System software controls whether or not event-based branches occur regardless of the contents of the BESCR. See Section 9.4.4 of Book III-S and Section 6.2.11 of Book III-S.

The entire BESCR can be read or written using SPR 806. Individual bits of the BESCR can be set or reset using two sets of additional SPR numbers.

- When ***mtspr*** indicates SPR 800 (Branch Event Status and Control Set, or BESCRS), the bits in BESCR which correspond to "1" bits in the source register are set to 1; all other bits in the BESCR are unaffected. SPR 801 (BESCRSU) provides the same capability to each of the upper 32 bits of the BESCR.
- When ***mtspr*** indicates SPR 802 (Branch Event Status and Control Reset, or BESCRR), the bits in BESCR which correspond to "1" bits in the source register are set to 0; all other bits in the BESCR are unaffected. SPR 803 (BESCRRU) provides the same capability to each of the upper 32 bits of the BESCR.

When ***mfspr*** indicates any of the above SPR numbers, the current value of the register is returned.

Programming Note

Event-based branch handlers typically reset event status bits upon entry, and enable event enable bits after processing an event. Execution of ***rfebb*** then re-enables the GE bit so that additional event-based branches can occur.

0 **Global Enable (GE)**

0 Event-based branches are disabled
1 Event-based branches are enabled.

When an event-based branch occurs, GE is set to 0 and is not altered by hardware until ***rfebb*** 1 is executed or software sets GE=1 and another event-based branch occurs.

1:31 **Event Control**

1:30 **Reserved**

31 **Performance Monitor Event-Based Exception Enable (PME)**

0 Performance Monitor event-based exceptions are disabled.
1 Performance Monitor event-based exceptions are enabled until a Performance Monitor event-based exception occurs, at which time:
- PME is set to 0
- PMEO is set to 1

See Chapter 9 of Book III-S for information about performance monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

32:33 **Transactional State [Category:TM]**

When an event-based branch occurs, hardware sets this field to indicate the transactional state of the processor when the event-based branch occurred.
The values and their associated meanings are as follows.

00 Non-transactional
01 Suspended
10 Transactional
11 Reserved

Programming Note

Event-based branch handlers should not modify this field since its value is used by the processor to determine the transactional state of the processor after the ***rfebb*** instruction is executed.

34:63 **Event Status**

34:62 Reserved

63 **Performance Monitor Event-Based Exception Occurred (PMEO)**

- 0 Performance Monitor event-based exception has not occurred since the last time software set this bit to 0.
- 1 A Performance Monitor event-based exception has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor event-based exception occurs. This bit can be set to 0 only by the ***mtspr*** instruction.

See Chapter 9 of Book III-S for information about performance monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

Programming Note

Software should set this bit to 0 after handling an event-based branch due to a Performance Monitor event-based exception.

7.2.2 Event-Based Branch Handler Register

The Event-Based Branch Handler Register (EBBHR) is a 64-bit register register that contains the 62 most significant bits of the effective address of the instruction that is executed next after an event-based branch occurs. Bits 62:32 must be available to be read and written by software.

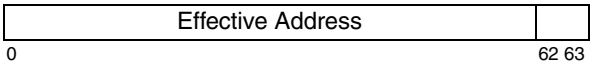


Figure 13. Event-Based Branch Handler Register (EBBHR)

Programming Note

The EBBHR can be used by software as a scratch-pad register after entry into an event-based branch handler, provided that its contents are restored prior to executing ***rfebb*** 1. An example of such usage is as follows, where SPRG3 is used to contain a pointer to a storage area where context information may be saved.

```
E:mtspr EBBHR, r1     // Save r1 in EBBHR
  mfspr r1, SPRG3     // Move SPRG3 to r1
  std r2, r1,offset1  // Store r2
  mfspr EBBHR,r2      // Copy original contents
                      // of r1 to r2
  std r2,offset2(r1)  // save original r1
  ..                  // Store rest of state
 ...                   // Process event
 ...                 // Restore all state except
                      // r1,r2
  r2 = &E            // Generate original value
                      // of EBBHR in r2
  mtspr EBBHR,r2      // Restore EBBHR
  ld r2 offset1(r1)   // restore r2
  ld r1 offset2(r1)   // restore r1
  rfebb 1             // Return from handler
```

7.2.3 Event-Based Branch Return Register

The Event-Based Branch Return Register (EBBRR) is a 64-bit register that contains the 62 most significant bits of an instruction effective address as specified below.

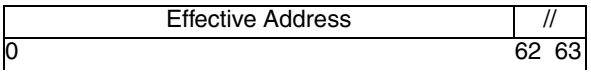


Figure 14. Event-Based Branch Return Register (EBBRR)

When an event-based branch occurs, bits 0:61 of the EBBRR are set to the effective address of the instruction that the processor would have attempted to execute next if no event-based branch had occurred. Bits 62:63 are reserved.

7.3 Event-Based Branch Instructions

Return from Event-Based Branch *XL-form*

rfebb S

19	///	///	///	S	146	/
0	6	11	16	20	21	31

$BESCR_{GE}$ ← S
MSR_{TS} ← $BESCR_{TS}$
NIA ←$_{iea}$ $EBBRR_{0:61}$ || 0b00

$BESCR_{GE}$ is set to S. The processor is placed in the transactional state indicated by $BESCR_{TS}$.

If there are no pending event-based exceptions, then the next instruction is fetched from the address $EBBRR_{0:61}$ || 0b00 (when MSR_{SF}=1) or $^{32}0$ || $EBBRR_{32:61}$ || 0b00 (when MSR_{SF}=0). If one or more pending event-based exceptions exist, an event-based branch is generated; in this case the value placed into EBBRR by the Event-Based Branch facility is the address of the instruction that would have been executed next had the event-based branch not occurred.

See Section 3.4 of Book III-S for additional information about this instruction.

Special Registers Altered:

BESCR
MSR (See Book III-S)

Extended Mnemonics:

Extended:	Equivalent to:
rfebb	rfebb 1

Programming Note

rfebb serves as both a basic and an extended mnemonic. The Assembler will recognize an ***rfebb*** mnemonic with one operand as the basic form, and an ***rfebb*** mnemonic with no operand as the extended form. In the extended form, the S operand is omitted and assumed to be 1.

Programming Note

If the $BESCR_{TS}$ has been modified by software after an event-based branch occurs, an illegal transaction state transition may occur. See Chapter 3.2.2 of Book III-S.

Programming Note

When an event-based branch occurs, the event-based branch handler executes the following sequence of operations. This sequence of operations assumes that the handler has access to a stack or other area in memory in which state information from the main program can be stored. Note also that in this example, the handler entry point is labeled “E,” r1 is used as a scratch register, and only Performance Monitor events are enabled.

```
E:Save  state          // This is the entry pt
 mfspr r1, BESCR      // Check event status
 Process event
 r1 ← 0x0000 0000 0000 0001
 mtspr BESCRR, r1
       //Reset PMEO event status bit
       //Note: The PMAO bit of MMCR0 must also
       //      be reset. See Book III-S.
 r1 ← 0x0000 0001 0000 0000
 mtspr BESCRS, r1
       //Enable PME bit
       //Note: The PMAE bit of MMCR0 must also
       //      be enabled. See Book III-S.
 Restore state
 rfebb  1            // return & global enable
```

Chapter 8. Decorated Storage Facility [Category: Decorated Storage]

The *Decorated Storage* facility provides *Load*, *Store*, and *Notify* operations to storage that have additional semantics other than the reading and writing of data values to the addressed storage locations. A decoration is specified that provides semantic information about how the operation is to be performed. A decorated device is a device that implements an address range of storage, and applies decorations to operations performed on the address range of storage.

A *Decorated Storage* instruction specifies the following attributes:

- The type of access, which is either a *Decorated Load*, *Decorated Store*, or a *Decorated Notify*.
- The EA in register RB, to which the operation is to be performed.
- The decoration in register RA, which further defines what operation should be performed by the decorated device.
- The data itself, either data provided by the processor to the decorated device (in the case of a *Decorated Store*), or the data provided by the decorated device to be consumed by the processor (in the case of a *Decorated Load*). *Decorated Notify* operations do not contain data.

The semantics of any *Decorated Storage* operation that is Caching Inhibited are defined by the decorated device depending on whether it is a *Decorated Load*, *Decorated Store*, or *Decorated Notify*, and the value supplied as a decoration. Such semantics may differ from decorated device to decorated device similar to how devices other than well-behaved memory may treat *Load* and *Store* operations. The semantics of any operation associated with a *Decorated Storage* operation that is not Caching Inhibited are the same as an analogous *Load* or *Store* instruction of the same data size.

The results of a *Decorated Storage* operation that is Caching Inhibited to a device that does not support decorations is boundedly undefined. The results of a *Load* or *Store* operation that is Caching Inhibited to a decorated device that requires a decoration is boundedly undefined.

For *Decorated Load* operations, a *Load* operation with the specified decoration is performed to the EA and the data provided by the decorated device is placed in the target register.

For *Decorated Store* operations, a *Store* operation using the data specified in the source register with the specified decoration is performed to the EA.

Decorated Load instructions are treated as *Load* instructions for address translation, access control, debug events, storage attributes, alignment, and memory access ordering. *Decorated Store* instructions are treated as *Store* instructions for address translation, access control, debug events, storage attributes, alignment, and memory access ordering. A *Decorated Notify* instruction is treated as a zero byte *Store* for address translation, access control, debug events, storage attributes, alignment, and memory access ordering.

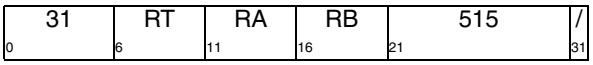
Programming Note

Software should be acutely aware of how transactions to a decorated device that implements Decorated Storage will occur. Not only does this imply knowing the particular decorated device's semantics, but also ensuring that the transactions are appropriately issued by the processor. This includes alignment, speculative accesses, and ordering. In general, Caching Inhibited accesses are required to be Guarded and properly aligned.

8.1 Decorated Load Instructions

Load Byte with Decoration Indexed X-form

lbdx RT,RA,RB




```
EA ← (RB)
RT ← 56 0 || MEM_DECORATED(EA,1,(RA))
```

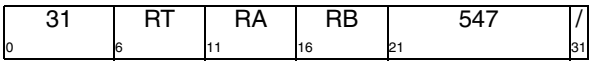
Let the effective address (EA) be the contents of RB. The byte in storage addressed by EA is loaded using the decoration supplied by (RA) into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Halfword with Decoration Indexed X-form

lhdx RT,RA,RB




```
EA ← (RB)
RT ← 48 0 || MEM_DECORATED(EA,2,(RA))
```

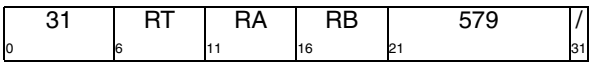
Let the effective address (EA) be the contents of RB. The halfword in storage addressed by EA is loaded using the decoration supplied by (RA) into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Word with Decoration Indexed X-form

lwdx RT,RA,RB




```
EA ← (RB)
RT ← 32 0 || MEM_DECORATED(EA,4,(RA))
```

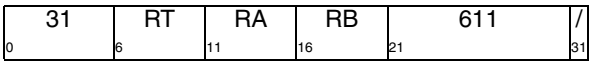
Let the effective address (EA) be the contents of RB. The word in storage addressed by EA is loaded using the decoration supplied by (RA) into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Doubleword with Decoration Indexed X-form

lddx RT,RA,RB [Co-requisite category: 64-Bit]




```
EA ← (RB)
RT ← MEM_DECORATED(EA,8,(RA))
```

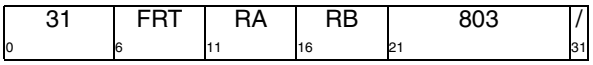
Let the effective address (EA) be the contents of RB. The doubleword in storage addressed by EA is loaded using the decoration supplied by (RA) into RT.

Special Registers Altered:

None

Load Floating Doubleword with Decoration Indexed X-form

lfddx FRT,RA,RB [Co-requisite category: FP]




```
EA ← (RB)
FRT ← MEM_DECORATED(EA,8,(RA))
```

Let the effective address (EA) be the contents of RB. The doubleword in storage addressed by EA is loaded using the decoration supplied by (RA) into FRT.

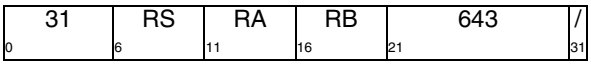
Special Registers Altered:

None

8.2 Decorated Store Instructions

Store Byte with Decoration Indexed X-form

stbdx RS,RA,RB




```
EA ← (RB)
MEM_DECORATED(EA,1,(RA)) ← (RS)56:63
```

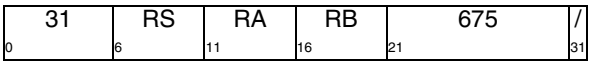
Let the effective address (EA) be the contents of RB. $(RS)_{56:63}$ are stored to the byte in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:

None

Store Halfword with Decoration Indexed X-form

sthdx RS,RA,RB




```
EA ← (RB)
MEM_DECORATED(EA,2,(RA)) ← (RS)48:63
```

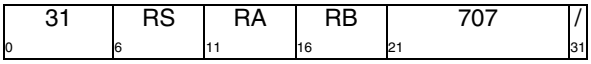
Let the effective address (EA) be the contents of RB. $(RS)_{48:63}$ are stored to the halfword in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:

None

Store Word with Decoration Indexed X-form

stwdx RS,RA,RB




```
EA ← (RB)
MEM_DECORATED(EA,4,(RA)) ← (RS)32:63
```

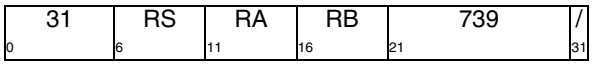
Let the effective address (EA) be the contents of RB. $(RS)_{32:63}$ are stored to the word in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:

None

Store Doubleword with Decoration Indexed X-form

stddx RS,RA,RB [Co-requisite category: 64-Bit]




```
EA ← (RB)
MEM_DECORATED(EA,8,(RA)) ← (RS)
```

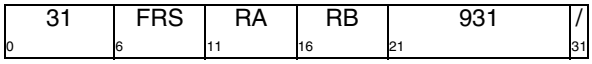
Let the effective address (EA) be the contents of RB. (RS) is stored to the doubleword in storage addressed by EA using the decoration supplied by (RA).

Special Registers Altered:

None

Store Floating Doubleword with Decoration Indexed X-form

stfddx FRS,RA,RB [Co-requisite category: FP]




```
EA ← (RB)
MEM_DECORATED(EA,8,(RA)) ← (FRS)
```

Let the effective address (EA) be the contents of RB. (FRS) is stored to the doubleword in storage addressed by EA using the decoration supplied by (RA).

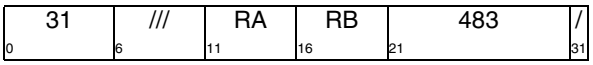
Special Registers Altered:

None

8.3 Decorated Notify Instructions

Decorated Storage Notify *X-form*

dsn RA,RB




```
EA ← (RB)
MEM_NOTIFY(EA,(RA))
```

Let the effective address (EA) be the contents of RB. The decoration supplied by (RA) is sent to the address in storage specified by EA.

Special Registers Altered:

None

Chapter 9. External Control [Category: External Control]

The External Control category of facilities and instructions permits a program to communicate with a special-purpose device. Two instructions are provided, both of which must be implemented if the facility is provided.

- *External Control In Word Indexed* (***eciwx***), which does the following:
  - Computes an effective address (EA) like most X-form instructions
  - Validates the EA as would be done for a load from that address
  - Translates the EA to a real address
  - Transmits the real address to the device
  - Accepts a word of data from the device and places it into a General Purpose Register
- *External Control Out Word Indexed* (***ecowx***), which does the following:
  - Computes an effective address (EA) like most X-form instructions
  - Validates the EA as would be done for a store to that address
  - Translates the EA to a real address
  - Transmits the real address and a word of data from a General Purpose Register to the device

Permission to execute these instructions and identification of the target device are controlled by two fields, called the E bit and the RID field respectively. If attempt is made to execute either of these instructions when E=0 the system data storage error handler is invoked. The location of these fields is described in Book III.

The storage access caused by ***eciwx*** and ***ecowx*** is performed as though the specified storage location is Caching Inhibited and Guarded, and is neither Write Through Required nor Memory Coherence Required.

Interpretation of the real address transmitted by ***eciwx*** and ***ecowx*** and of the 32-bit value transmitted by ***ecowx*** is up to the target device, and is not specified by the Power ISA. See the System Architecture documentation for a given Power ISA system for details on how the External Control facility can be used with devices on that system.

Example

An example of a device designed to be used with the External Control facility might be a graphics adapter. The ***ecowx*** instruction might be used to send the device the translated real address of a buffer containing graphics data, and the word transmitted from the General Purpose Register might be control information that tells the adapter what operation to perform on the data in the buffer. The ***eciwx*** instruction might be used to load status information from the adapter.

A device designed to be used with the External Control facility may also recognize events that indicate that the address translation being used by the processor has changed. In this case the operating system need not “pin” the area of storage identified by an ***eciwx*** or ***ecowx*** instruction (i.e., need not protect it from being paged out).

9.1 External Access Instructions

In the instruction descriptions the statements “this instruction is treated as a *Load*” and “this instruction is treated as a *Store*” have the same meanings as for the *Cache Management* instructions; see Section 4.3.

External Control In Word Indexed X-form

eciwx RT,RA,RB

31	RT	RA	RB	310	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
raddr ← address translation of EA
send load word request for raddr to
   device identified by RID
RT ← 320 || word from device
```

Let the effective address (EA) be the sum (RA|0)+(RB).

A load word request for the real address corresponding to EA is sent to the device identified by RID, bypassing the cache. The word returned by the device is placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

The E bit must be 1. If it is not, the data storage error handler is invoked.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

This instruction is treated as a *Load*.

See Book III-S for additional information about this instruction.

Special Registers Altered:

None

Programming Note

The ***eieio***<S> or ***mbar***<E> instruction can be used to ensure that the storage accesses caused by ***eciwx*** and ***ecowx*** are performed in program order with respect to other Caching Inhibited and Guarded storage accesses.

External Control Out Word Indexed X-form

ecowx RS,RA,RB

31	RS	RA	RB	438	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
raddr ← address translation of EA
send store word request for raddr to
  device identified by RID
send (RS)32:63 to device
```

Let the effective address (EA) be the sum (RA|0)+(RB).

A store word request for the real address corresponding to EA and the contents of $RS_{32:63}$ are sent to the device identified by RID, bypassing the cache.

The E bit must be 1. If it is not, the data storage error handler is invoked.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

This instruction is treated as a *Store*, except that its storage access is not performed in program order with respect to accesses to other Caching Inhibited and Guarded storage locations unless software explicitly imposes that order.

See Book III-S for additional information about this instruction.

Special Registers Altered:

None

Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book II. Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Data Cache Block Flush Mnemonics

The L field in the *Data Cache Block Flush* instruction controls the scope of the flush function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

Note: ***dcbf*** serves as both a basic and an extended mnemonic. The Assembler will recognize a ***dcbf*** mnemonic with three operands as the basic form, and a ***dcbf*** mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

dcbf RA,RB	(equivalent to: dcbf RA,RB,0)
dcbfl RA,RB	(equivalent to: dcbf RA,RB,1)

A.2 Load and Reserve Mnemonics

The EH field in the *Load and Reserve* instructions provides a hint regarding the type of algorithm implemented by the instruction sequence being executed. Extended mnemonics are provided that allow the EH value to be omitted and assumed to be 0b0.

Note: ***lbarx***, ***lharx***, ***lwarx, ldarx,*** and ***lqarx*** serve as both basic and extended mnemonics. The Assembler will recognize these mnemonics with four operands as the basic form, and these mnemonics with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

lbarx RT,RA,RB	(equivalent to: lbarx RT,RA,RB,0)
lharx RT,RA,RB	(equivalent to: lharx RT,RA,RB,0)
lwarx RT,RA,RB	(equivalent to: lwarx RT,RA,RB,0)
ldarx RT,RA,RB	(equivalent to: ldarx RT,RA,RB,0)
lqarx RT,RA,RB	(equivalent to: lqarx RT,RA,RB,0)

A.3 Synchronize Mnemonics

The L field in the *Synchronize* instruction controls the scope of the synchronization function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand. Two extended mnemonics are provided for the L=0 value in order to support Assemblers that do not recognize the ***sync*** mnemonic.

Note: ***sync*** serves as both a basic and an extended mnemonic. The Assembler will recognize a ***sync*** mnemonic with one operand as the basic form, and a ***sync*** mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

sync	(equivalent to:	sync	0)
msync<E>	(equivalent to:	sync	0)
lwsync	(equivalent to:	sync	1)
ptesync<S>	(equivalent to:	sync	2)

A.4 Wait Mnemonics

The WC field in the ***wait*** instruction determines the condition that causes instruction execution to resume. Extended mnemonics are provided that represent the WC value in the mnemonic rather than requiring it to be coded as a numeric operand.

Note: ***wait*** serves as both a basic and an extended mnemonic. The Assembler will recognize a ***wait*** mnemonic with one operand as the basic form, and a ***wait*** mnemonic with no operands as the extended form. In the extended form the WC operand is omitted and assumed to be 0.

wait	(equivalent to: wait 0)
waitrsv	(equivalent to: wait 1)
waitimpl	(equivalent to: wait 2)

Appendix B. Programming Examples for Sharing Storage

This appendix gives examples of how dependencies and the *Synchronization* instructions can be used to control storage access ordering when storage is shared between programs.

Many of the examples use extended mnemonics (e.g., ***bne***, ***bne-***, ***cmpw***) that are defined in Appendix E of Book I.

Many of the examples use the *Load And Reserve* and *Store Conditional* instructions, in a sequence that begins with a *Load And Reserve* instruction and ends with a *Store Conditional* instruction (specifying the same storage location as the *Load Conditional*) followed by a *Branch Conditional* instruction that tests whether the *Store Conditional* instruction succeeded.

In these examples it is assumed that contention for the shared resource is low; the conditional branches are optimized for this case by using "+" and "-" suffixes appropriately.

The examples deal with words; they can be used for doublewords by changing all word-specific mnemonics to the corresponding doubleword-specific mnemonics (e.g., ***lwarx*** to ***ldarx***, ***cmpw*** to ***cmpd***).

In this appendix it is assumed that all shared storage locations are in storage that is Memory Coherence Required, and that the storage locations specified by *Load And Reserve* and *Store Conditional* instructions are in storage that is neither Write Through Required nor Caching Inhibited.

B.1 Atomic Update Primitives

This section gives examples of how the *Load And Reserve* and *Store Conditional* instructions can be used to emulate atomic read/modify/write operations.

An atomic read/modify/write operation reads a storage location and writes its next value, which may be a function of its current value, all as a single atomic operation. The examples shown provide the effect of an atomic read/modify/write operation, but use several instructions rather than a single atomic instruction.

Fetch and No-op

The "Fetch and No-op" primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR 3 and the data loaded are returned in GPR 4.

```
loop:
   lwarx  r4,0,r3 #load and reserve
   stwcx. r4,0,r3 #store old value if
                  #  still reserved
   bne-   loop    #loop if lost reservation
```

Note:

1. The ***stwcx***., if it succeeds, stores to the target location the same value that was loaded by the preceding ***lwarx***. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the ***lwarx*** is still the current value at the time the ***stwcx.*** is executed.

Fetch and Store

The "Fetch and Store" primitive atomically loads and replaces a word in storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR 3, the new value is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   stwcx. r4,0,r3 #store new value if
                  #  still reserved
   bne-   loop    loop if lost reservation
```

Fetch and Add

The “Fetch and Add” primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR 3, the increment is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   add    r0,r4,r5#increment word
   stwcx. r0,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
```

Fetch and AND

The “Fetch and AND” primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR 3, the value to AND into it is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   and    r0,r4,r5#AND word
   stwcx. r0,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
```

Note:

1. The sequence given above can be changed to perform another Boolean operation atomically on a word in storage, simply by changing the ***and*** instruction to the desired Boolean instruction (***or***, ***xor***, etc.).

Test and Set

This version of the “Test and Set” primitive atomically loads a word from storage, sets the word in storage to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR 3, the new value (nonzero) is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   cmpwi  r5,0    #done if word not equal to 0
   bne-   exit
   stwcx. r4,0,r3 #try to store non-0
   bne-   loop    #loop if lost reservation
exit: ...
```

Compare and Swap

The “Compare and Swap” primitive atomically compares a value in a register with a word in storage, if they are equal stores the value from a second register into the word in storage, if they are unequal loads the word from storage into the first register, and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR 3, the comparand is in GPR 4 and the old value is returned there, and the new value is in GPR 5.

```
loop:
   lwarx  r6,0,r3 #load and reserve
   cmpw   r4,r6   #1st 2 operands equal?
   bne-   exit    #skip if not
   stwcx. r5,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
exit:
   mr     r4,r6   #return value from storage
```

Notes:

1. The semantics given for “Compare and Swap” above are based on those of the IBM System/370 Compare and Swap instruction. Other architectures may define a Compare and Swap instruction differently.
2. “Compare and Swap” is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by ***lwarx*** and ***stwcx.***. A major weakness of a System/370-style Compare and Swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.
3. In some applications the second ***bne-*** instruction and/or the ***mr*** instruction can be omitted. The ***bne-*** is needed only if the application requires that if the EQ bit of CR Field 0 on exit indicates “not equal” then (r4) and (r6) are in fact not equal. The ***mr*** is needed only if the application requires that if the comparands are not equal then the word from storage is loaded into the register with which it was compared (rather than into a third register). If either or both of these instructions is omitted, the resulting Compare and Swap does not obey System/370 semantics.

B.2 Lock Acquisition and Release, and Related Techniques

This section gives examples of how dependencies and the *Synchronization* instructions can be used to implement locks, import and export barriers, and similar constructs.

B.2.1 Lock Acquisition and Import Barriers

An "import barrier" is an instruction or sequence of instructions that prevents storage accesses caused by instructions following the barrier from being performed before storage accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A ***sync*** instruction can be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant storage accesses.

B.2.1.1 Acquire Lock and Import Shared Storage

If ***lwarx*** and ***stwcx.*** instructions are used to obtain the lock, an import barrier can be constructed by placing an ***isync*** instruction immediately following the loop containing the ***lwarx*** and ***stwcx.***. The following example uses the "Compare and Swap" primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:
   lwarx  r6,0,r3,1 #load lock and reserve
   cmpw   r4,r6     #skip ahead if
   bne-   wait      #  lock not free
   stwcx. r5,0,r3   #try to set lock
   bne-   loop      #loop if lost reservation
   isync            #import barrier
   lwz    r7,data1(r9)#load shared data
   .
   .
wait...           #wait for lock to free
```

The hint provided with ***lwarx*** indicates that after the program acquires the lock variable (i.e. ***stwcx***. is successful), it will release it (i.e. store to it) prior to another program attempting to modify it.

The second ***bne-*** does not complete until CR0 has been set by the ***stwcx.***. The ***stwcx.*** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the ***stwcx.*** completes successfully. Together, the second ***bne-*** and the subsequent ***isync*** create an import barrier that prevents the load from "data1" from being performed until the branch has been resolved not to be taken.

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an ***lwsync*** instruction can be used instead of the ***isync*** instruction. If ***lwsync*** is used, the load from "data1" may be performed before the ***stwcx.***. But if the ***stwcx.*** fails, the second branch is taken and the ***lwarx*** is re-executed. If the ***stwcx.*** succeeds, the value returned by the load from "data1" is valid even if the load is performed before the ***stwcx.***, because the ***lwsync*** ensures that the load is performed after the instance of the ***lwarx*** that created the reservation used by the successful ***stwcx.***.

B.2.1.2 Obtain Pointer and Import Shared Storage

If ***lwarx*** and ***stwcx.*** instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the "Fetch and Add" primitive to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load pointer and reserve
   add    r0,r4,r5#increment the pointer
   stwcx. r0,0,r3 #try to store new value
   bne-   loop    #loop if lost reservation
   lwz    r7,data1(r5) #load shared data
```

The load from "data1" cannot be performed until the pointer value has been loaded into GPR 5 by the ***lwarx***. The load from "data1" may be performed before the ***stwcx.***. But if the ***stwcx.*** fails, the branch is taken and the value returned by the load from "data1" is discarded. If the ***stwcx.*** succeeds, the value returned by the load from "data1" is valid even if the load is performed before the ***stwcx.***, because the load uses the pointer value returned by the instance of the ***lwarx*** that created the reservation used by the successful ***stwcx.***.

An ***isync*** instruction could be placed between the ***bne-*** and the subsequent ***lwz***, but no ***isync*** is needed if all accesses to the shared data structure depend on the value returned by the ***lwarx***.

B.2.2 Lock Release and Export Barriers

An “export barrier” is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock will be performed with respect to any other processor before the store that releases the lock is performed with respect to that processor.

B.2.2.1 Export Shared Storage and Release Lock

A ***sync*** instruction can be used as an export barrier independent of the storage control attributes (e.g., presence or absence of the Caching Inhibited attribute) of the storage containing the shared data structure. Because the lock must be in storage that is neither Write Through Required nor Caching Inhibited, if the shared data structure is in storage that is Write Through Required or Caching Inhibited a ***sync*** instruction *must* be used as the export barrier.

In this example it is assumed that the shared data structure is in storage that is Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw    r7,data1(r9)#store shared data (last)
sync           #export barrier
stw    r4,lock(r3)#release lock
```

The ***sync*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***sync*** have been performed with respect to that processor.

B.2.2.2 Export Shared Storage and Release Lock using lwsync

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an ***lwsync*** instruction can be used as the export barrier. Using ***lwsync*** rather than ***sync*** will yield better performance in most systems.

In this example it is assumed that the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw    r7,data1(r9)#store shared data (last)
lwsync         #export barrier
stw    r4,lock(r3)#release lock
```

The ***lwsync*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***lwsync*** have been performed with respect to that processor.

B.2.3 Safe Fetch

If a load must be performed before a subsequent store (e.g., the store that releases a lock protecting a shared data structure), a technique similar to the following can be used.

In this example it is assumed that the address of the storage operand to be loaded is in GPR 3, the contents of the storage operand are returned in GPR 4, and the address of the storage operand to be stored is in GPR 5.

```
lwz    r4,0(r3)#load shared data
cmpw   r4,r4   #set CR0 to "equal"
bne-   $-8     #branch never taken
stw    r7,0(r5)#store other shared data
```

An alternative is to use a technique similar to that described in Section B.2.1.2, by causing the ***stw*** to depend on the value returned by the ***lwz*** and omitting the ***cmpw*** and ***bne-***. The dependency could be created by ANDing the value returned by the ***lwz*** with zero and then adding the result to the value to be stored by the ***stw***. If both storage operands are in storage that is neither Write Through Required nor Caching Inhibited, another alternative is to replace the ***cmpw*** and ***bne-*** with an ***lwsync*** instruction.

B.3 List Insertion

This section shows how the ***lwarx*** and ***stwcx.*** instructions can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The "next element pointer" from the list element after which the new element is to be inserted, here called the "parent element", is stored into the new element, so that the new element points to the next element in the list; this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR 3, the address of the new element is in GPR 4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:
   lwarx  r2,0,r3 #get next pointer
   stw    r2,0(r4)#store in new element
   lwsync or sync #order stw before stwcx
   stwcx. r4,0,r3 #add new element to list
   bne-   loop    #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, "livelock" can occur. (Livelock is a state in which processors interact in a way such that no processor makes forward progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
   lwz    r2,0(r3)#get next pointer
loop1:
   mr     r5,r2   #keep a copy
   stw    r2,0(r4)#store in new element
   sync           #order stw before stwcx.
                  and before lwarx
loop2:
   lwarx  r2,0,r3 #get it again
   cmpw   r2,r5   #loop if changed (someone
   bne-   loop1   #  else progressed)
   stwcx. r4,0,r3 #add new element to list
   bne-   loop2   #loop if failed
```

In the preceding example, livelock is avoided by the fact that each processor re-executes the ***stw*** only if some other processor has made forward progress.

B.4 Notes

The following notes apply to Section B.1 through Section B.3.

1. To increase the likelihood that forward progress is made, it is important that looping on ***lwarx***/***stwcx.*** pairs be minimized. For example, in the "Test and Set" sequence shown in Section B.1, this is achieved by testing the old value before attempting the store; were the order reversed, more ***stwcx.*** instructions might be executed, and reservations might more often be lost between the ***lwarx*** and the ***stwcx.***

2. The manner in which ***lwarx*** and ***stwcx.*** are communicated to other processors and mechanisms, and between levels of the storage hierarchy within a given processor, is implementation-dependent. In some implementations performance may be improved by minimizing looping on a ***lwarx*** instruction that fails to return a desired value. For example, in the "Test and Set" sequence shown in Section B.1, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the "bne- exit" to "bne- loop". However, in some implementations better performance may be obtained by using an ordinary Load instruction to do the initial checking of the value, as follows.

```
loop:
 lwz   r5,0(r3)#load the word
 cmpwi r5,0    #loop back if word
 bne-  loop    #  not equal to 0
 lwarx r5,0,r3 #try again, reserving
 cmpwi r5,0    #  (likely to succeed)
 bne-  loop
 stwcx.r4,0,r3 #try to store non-0
 bne-  loop    #loop if lost reserv'n
```

3. In a multiprocessor, livelock is possible if there is a *Store* instruction (or any other instruction that can clear another processor's reservation; see Section 1.7.3.1) between the ***lwarx*** and the ***stwcx.*** of a ***lwarx***/***stwcx.*** loop and any byte of the storage location specified by the Store is in the reservation granule. For example, the first code sequence shown in Section B.3 can cause livelock if two list elements have next element pointers in the same reservation granule.

B.5 Transactional Lock Elision [Category: Transactional Memory]

This section illustrates the use of the Transactional Memory facility to implement transactional lock elision (TLE), in which lock-based critical sections are speculatively executed as a transaction without first acquiring a lock. This locking protocol is an alternative to the rou-

tines described above, yielding increased concurrency when the lock that guards a critical section is frequently unnecessary.

B.5.1 Enter Critical Section

The following example shows the entry point to a critical section using transactional lock elision. The entry code starts a transaction using the *tbegin.* instruction and checks whether the transaction was aborted or not. If not, it checks whether the lock is free or not. If the lock is found to be free, the thread proceeds to execute the critical section.

In this example it is assumed that the address of the lock is in GPR 3, and the value indicating that the lock is free is in GPR 4. The handling of cases of transaction abort and busy lock are described in subsequent examples.

```
tle_entry:
   tbegin.            #Start TLE transaction
   beq- tle_abort     #Handle TLE transaction abort
   lwz r6,0(r3)       #Read lock
   cmpw r6,r4         #Check if lock is free
   bne- busy_lock     #If not, handle lock busy case

critical_section1:
```

B.5.2 Handling Busy Lock

In the event that the lock is already held, by either another thread or the current thread, the transaction is aborted using the *tabort* instruction, using a software-defined code *TLE_BUSY_LOCK* indicating the cause of the abort. The abort returns control to the ***beq*** following ***tbegin.*** in the critical section entrance sequence, allowing for an abort handler to react appropriately.

```
busy_lock:
   li r3, TLE_BUSY_LOCK
   tabort r3             #Abort TLE transaction
```

B.5.3 Handling TLE Abort

A TLE transaction may fail for one of a variety of causes, persistent and transient. Persistent causes are certain—or at least highly likely—to cause future attempts to execute the same transaction to fail. However, for transient causes, it is possible that the failure cause may not be re-encountered in a subsequent attempt. Thus, persistent aborts are handled by taking a non-transactional path that involves the actual acquisition of the lock, while transient aborts retry the critical section using TLE.

The following example illustrates the handling of aborts in TLE. It is assumed that the address of the lock is in GPR 3. The immediate value of the *andis.* instruction selects the Failure Persistent bit in the upper half of TEXASR to be tested.

```
tle_abort:
   mfspr r4, TEXASRU    # Read high-order half
                        # of TEXASR
   andis. r5,r4,0x0100  # determine whether failure
                       # is likely to be persistent
   bne tle_acquire_lock   #Persistent, acquire lock
                          #enter critical sec
   b tle_entry            #Transient, try TLE again
```

This example can be extended to keep track of the number of transient aborts and fall back on the acquisition of the lock after the number of transient failures reaches some threshold. It can also be extended to handle reentrant locks. Acquisition of TLE locks is described in a subsequent example.

B.5.4 TLE Exit Section Critical Path

The following example illustrates the instruction sequence used to exit a TLE critical section. The CR0 value set by ***tend.*** indicates whether the current thread was in a transaction. If so, the exited critical section was entered speculatively, and the transaction is ended. If not, the execution takes a path to release the lock.

Release of an acquired TLE lock is described in a subsequent example.

```
tle_exit:
   tend.                   #End the current trans-
                           #action, if any
   bng- tle_release_lock   #Release lock, if was
                           #not in a transaction
```

B.5.5 Acquisition and Release of TLE Locks

The steps for acquiring and releasing a lock associated with a TLE critical section are nearly identical to those for acquiring and releasing conventional locks that are not elided, as described in Section B.2. The only difference is special care must be taken to prevent loads and stores inside the critical section protected by the lock from being performed before the *Store Conditional* instruction to the lock variable.

The ***isync*** from the *Acquire Lock and Import Shared Storage* sequence described in Section B.2.1.1 is insufficient for ordering subsequent instructions. Instead, a

stronger storage access ordering instruction is needed. The following example shows the resulting steps for acquiring a TLE lock, replacing the *isync* instruction with a *sync* instruction. In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5.

```
tle_acquire_lock:
   lwarx r10,0,r3,1       #Load lock and reserve
   cmpw r4,r10            #Skip ahead if
   bne- wait              #Lock not free
   stwcx. r5,0,r3         #Try to set lock
   bne- tle_acquire_lock  #Loop if lost reservation
   sync                   #Import barrier for TLE

critical_section1:
```

The instruction sequence necessary for the release of a TLE lock is identical to the conventional lock release sequence, where ***lwsync*** is sufficient to prevent the store that releases the lock from being performed before the loads and stores in the critical section protected by the lock.

Book III-S:

Power ISA Operating Environment Architecture - Server Environment [Category: Server]

Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I describes computation modes, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the Power ISA Operating Environment Architecture.

1.2 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt”, “Hypervisor Data Storage interrupt”, or “Data Segment interrupt”, as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Hypervisor Emulation Assistance interrupt”.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Hypervisor Instruction Storage interrupt”, or “Instruction Segment interrupt”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction type Program interrupt”.
- For “system service program” substitute “System Call interrupt”.
- For “system trap handler” substitute “Trap type Program interrupt”.
- For “system Facility Unavailable error handler” substitute “Facility Unavailable interrupt” or “Hypervisor Facility Unavailable interrupt.”

1.2.1 Definitions and Notation

The definitions and notation given in Book I are augmented by the following.

- **Threaded processor, single-threaded processor, thread**

  A threaded processor implements one or more “threads”, where a thread corresponds to the Book I/II concept of “processor”. That is, the definition of “thread” is the same as the Book I definition of “processor”, and “processor” as used in Books I and II can be thought of as either a single-threaded processor or as one thread of a multi-threaded processor. The only unqualified uses of “processor” in Book III are in resource names (e.g. Processor Identification Register); such uses should be regarded as meaning “threaded processor”. The threads of a multi-threaded processor typically share certain resources, such as the hardware components that execute certain kinds of instructions (e.g., Fixed-Point instructions), certain caches, the address translation mechanism, and certain hypervisor resources.

- **real page**
  A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size is 4KB.

- **context of a program**
  The state (e.g., privilege and relocation) in which the program executes. The context is controlled by the contents of certain System Registers, such as the MSR and SDR1, of certain lookaside buffers, such as the SLB and TLB, and of the Page Table.

- **exception**
  An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

- **interrupt**
  The act of changing the machine state in response to an exception, as described in Chapter 6. “Interrupts” on page 939.

- **trap interrupt**
  An interrupt that results from execution of a *Trap* instruction.
- Additional exceptions to the sequential execution model, beyond those described in the section entitled “Instruction Fetching” in Book I, are the following.
  - A System Reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)
  - A context-altering instruction is executed (Chapter 12. “Synchronization Requirements for Context Alterations” on page 1009). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.
  - A Reference and Change bit is updated by the thread. The update need not be performed with respect to that thread until the required subsequent synchronizing operation has occurred.
  - A *Branch* instruction is executed and the branch is taken. The update of the Come-From Address Register<S> (see Section 8.1.1 of Book III-S) need not occur until a subsequent context synchronizing operation has occurred.
- **“must”**
  If hypervisor software violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage in real addressing mode, the results are undefined, and may include altering resources belonging to other partitions, causing the system to “hang”, etc.
- **hardware**
  Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.
- **hypervisor privileged**
  A term used to describe an instruction or facility that is available only when the thread is in hypervisor state.
- **privileged state** and **supervisor mode**
  Used interchangeably to refer to a state in which privileged facilities are available.
- **problem state** and **user mode**
  Used interchangeably to refer to a state in which privileged facilities are not available.
- /, //, ///, ... denotes a field that is reserved in an instruction, in a register, or in an architected storage table.
- ?, ??, ???, ... denotes a field that is implementation-dependent in an instruction, in a register, or in an architected storage table.

1.2.2 Reserved Fields

Book I's description of the handling of reserved bits in System Registers, and of reserved values of defined fields of System Registers, applies also to the SLB. Book I's description of the handling of reserved values of defined fields of System Registers applies also to architected storage tables (e.g., the Page Table).

Some fields of certain architected storage tables may be written to automatically by the hardware, e.g., Reference and Change bits in the Page Table. When the hardware writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) specifically being updated are modified.
- Contents of reserved fields are either preserved or written as zero.

Programming Note

Software should set reserved fields in the SLB and in architected storage tables to zero, because these fields may be assigned a meaning in some future version of the architecture.

1.3 General Systems Overview

The hardware contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Most implementations also contain data and instruction caches. Instructions that the processing unit can execute fall into the following classes:

- instructions executed in the Branch Facility
- instructions executed in the Fixed-Point Facility
- instructions executed in the Floating-Point Facility
- instructions executed in the Vector Facility

Almost all instructions executed in the Branch Facility, Fixed-Point Facility, Floating-Point Facility, and Vector Facility are nonprivileged and are described in Book I. Book II may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged

instructions for cache management). Instructions related to the privileged state, control of hardware resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

1.4 Exceptions

The following augments the exceptions defined in Book I that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSR_{FP}=0 (Floating-Point Unavailable interrupt)
- an attempt to modify a hypervisor resource when the thread is in privileged but non-hypervisor state (see Chapter 2), or an attempt to execute a hypervisor-only instruction (e.g., ***tlbie***) when the thread is in privileged but non-hypervisor state
- the execution of a traced instruction (Trace interrupt)
- the execution of a Vector instruction when the vector facility is unavailable (Vector Unavailable interrupt)

1.5 Synchronization

The synchronization described in this section refers to the state of the thread that is performing the synchronization.

1.5.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. The context synchronizing operations are the ***isync*** instruction, the *System Linkage* instructions, the ***mtmsr***[***d***] instructions with L=0, and most interrupts (see Section 6.4).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of ***isync***, does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.
3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.
4. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 6.8).
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 5.5, “Performing Operations Out-of-Order”.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.5.2.

Unlike the *Synchronize* instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for ***isync*** (and ***sync*** and ***ptesync***; see Section 1.5.2) because all other execution synchronizing operations also alter context.

1.5.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.5.1). ***sync*** and ***ptesync*** are treated like ***isync*** with respect to item 2. The execution synchronizing instructions are ***sync***, ***ptesync***, the ***mtmsr***[***d***] instructions with L=1, and all context synchronizing instructions.

Programming Note

Unlike a context synchronizing operation, an execution synchronizing instruction does not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

Chapter 2. Logical Partitioning (LPAR) and Thread Control

2.1 Overview

The Logical Partitioning (LPAR) facility permits threads and portions of real storage to be assigned to logical collections called *partitions*, such that a program executing on a thread in one partition cannot interfere with any program executing on a thread in a different partition. This isolation can be provided for both problem state and privileged state programs, by using a layer of trusted software, called a *hypervisor* program (or simply a "hypervisor"), and the resources provided by this facility to manage system resources. (A hypervisor is a program that runs in hypervisor state; see below.)

The number of partitions supported is implementation-dependent.

A thread is assigned to one partition at any given time. A thread can be assigned to any given partition without consideration of the physical configuration of the system (e.g., shared registers, caches, organization of the storage hierarchy), except that threads that share certain hypervisor resources may need to be assigned to the same partition; see Section 2.7. The registers and facilities used to control Logical Partitioning are listed below and described in the following subsections.

Except in the following subsections, references to the "operating system" in this document include the hypervisor unless otherwise stated or obvious from context.

2.2 Logical Partitioning Control Register (LPCR)

The layout of the Logical Partitioning Control Register (LPCR) is shown in Figure 1 below.

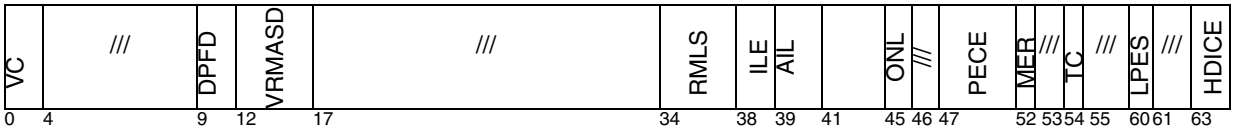


Figure 1. Logical Partitioning Control Register

The contents of the LPCR control a number of aspects of the operation of the thread with respect to a logical partition. Below are shown the bit definitions for the LPCR.

Bit **Description**

0:3 ***Virtualization Control*** (VC)

Controls the virtualization of partition memory. This field contains three subfields, VPM, ISL, and KBV. Accesses that are initiated in hypervisor state (i.e. $MSR_{HV\ PR}$=0b10) are performed as if VC=0b0000.

0:1 ***Virtualized Partition Memory*** (VPM)

This field controls whether VPM mode is enabled as specified below. (See Section 5.7.3.4 and Section 5.7.2, "Virtualized Partition Memory (VPM) Mode" for additional information on VPM mode.)

Bit **Description**

0 This bit controls whether VPM mode is enabled when address translation is disabled
0 - VPM mode disabled
1 - VPM mode enabled

1 This bit controls whether VPM mode is enabled when address translation is enabled
0 - VPM mode disabled
1 - VPM mode enabled

2 ***Ignore SLB Large Page Specification*** (ISL)

Controls whether ISL mode is enabled as specified below.

0 - ISL mode disabled
1 - ISL mode enabled

When ISL mode is enabled and address translation is enabled, address translation is performed as if the contents of $SLB_{L||LP}$ were 0b000. When address translation is disabled, the setting of the ISL bit has no effect. ISL mode has no effect on SLB, TLB, and ERAT entry invalidations caused by ***slbie***, ***slbia***, ***tlbia***, ***tlbie***, and ***slbie***.

3 ***Key-Based Virtualization*** (KBV)

Controls whether Key-Based Virtualization is enabled as specified below.

0 - KBV is disabled
1 - KBV is enabled

When KBV is enabled, Virtual Page Class Key Storage Protection exceptions that occur on operand accesses when VPM_1=0 cause Hypervisor Data Storage interrupts.

Programming Note

Key-Based Virtualization provides an efficient means for the hypervisor to intercept storage references, e.g. MMIO, that must be emulated. (The corresponding behavior for instruction fetching is not desired.) Virtual Page Class Key Storage Protection exceptions not handled by the hypervisor should be reflected to the operating system at its Data Storage interrupt vector with the hypervisor having set $DSISR_{42}$.

4:8 Reserved

9:11 ***Default Prefetch Depth*** (DPFD)

The DPFD field is used as the default prefetch depth for data stream prefetching when $DSCR_{DPFD}$=0; see page 764.

12:16 ***Virtual Real Mode Area Segment Descriptor*** (VRMASD)

When address translation is disabled and VPM_0=1, the contents of this field specify the L and LP fields of the segment descriptor that apply for storage references to the virtualized real mode area (VRMA). See Section 5.7.3.4 for additional information. The definitions and allowed values of the L and LP fields are the same as for the corresponding fields in the segment descriptor. (See Section 5.7.7.) If VPM_0=0 or address translation is enabled, the setting of the VRMASD has no effect.

Bit Description

0 ***Virtual Page Size Selector Bit 0*** (L)
1:2 Reserved
3:4 ***Virtual Page Size Selector Bits 1:2*** (LP)

Programming Note

Specifying that L||LP=0b000 in the VRMASD field when VPM mode is enabled has the same effect on address translation when translation is disabled as enabling ISL mode when translation is enabled.

ISL mode is needed when translation is enabled because translation uses the SLB, and the contents of the SLB are accessible to the operating system and should not be modified by the hypervisor. ISL mode is not needed when translation is disabled since translation uses the VRMASD, which is not visible to the operating system and is in complete control of the hypervisor.

17:33 Reserved

34:37 ***Real Mode Limit Selector*** (RMLS)

The RMLS field specifies the largest effective address that can be used by partition software when address translation is disabled. The valid RMLS values are implementation-dependent, and each value corresponds to a maximum effective address of 2^m, where m has a minimum value of 12 and a maximum value equal to the number of bits in the real address size supported by the implementation.

38 ***Interrupt Little-Endian*** (ILE)

The contents of the ILE bit are copied into MSR_{LE} by interrupts that set MSR_{HV} to 0 (see Section 6.5), to establish the Endian mode for the interrupt handler.

39:40 ***Alternate Interrupt Location*** (AIL)

Controls the effective address offset of the interrupt handler and the relocation mode in which it begins execution for all interrupts except Machine Check, System Reset, and Hypervisor Maintenance.

0 The interrupt is taken with $MSR_{IR\ DR}$ = 0b00 and no effective address offset.

1 Reserved

2 The interrupt is taken with $MSR_{IR\ DR}$ = 0b11 and an effective address offset of 0x0000_0000_0001_8000.

3 The interrupt is taken with $MSR_{IR\ DR}$ = 0b11 and an effective address offset of 0xc000_0000_0000_4000.

Interrupts that cause a transition from MSR_{HV}=0 to MSR_{HV}=1, or that occur when MSR_{IR}=0 or MSR_{DR}=0, are always taken as if $LPCR_{AIL}$=0.

Programming Note

One of the purposes of the AIL field is to provide relocation for interrupts that occur while an application is running with $MSR_{HV\ PR}$=0b11 under a "bare metal" operating system (i.e., an operating system that runs in hypervisor state), such as KVM.

41:44 Reserved

45 ***Online*** (ONL)

0 The PURR and SPURR do not increment.
1 The PURR and SPURR increment.

Programming Note

Typically, the hypervisor sets the ONL bit to 0 when the thread is not in a power saving mode, is not performing useful work, and is available for use. The hypervisor may take the state of the ONL bit into account when making course-grain load balancing and power management decisions.

46 Reserved

47:51 ***Power-saving mode Exit Cause Enable*** (PECE)

47 <S.PC> If $PECE_0$ = 1 when a *Power-Saving Mode* instruction is executed, Directed Privileged Doorbell exceptions are enabled to cause exit from power-saving mode; otherwise Directed Privileged Doorbell exceptions are disabled from causing exit from power-saving mode.

48 <S.PC> If $PECE_1$ = 1 when a *Power-Saving Mode* instruction is executed, Directed Hypervisor Doorbell exceptions are enabled to cause exit from power-saving mode; otherwise Directed Hypervisor Doorbell exceptions are disabled from causing exit from power-saving mode.

49 If $PECE_2$ = 1 when a *Power-Saving Mode* instruction is executed, External exceptions are enabled to cause exit from power-saving mode; otherwise External exceptions are disabled from causing exit from power-saving mode.

50 If $PECE_3$ = 1 when a *Power-Saving Mode* instruction is executed, Decrementer exceptions are enabled to cause exit from power-saving mode; otherwise Decrementer exceptions are disabled from causing exit from power-saving mode. (In sleep and rvwinkle power-saving levels, Decrementer exceptions do not occur if the state of the Decrementer is not maintained and updated as if the thread was not in power-saving mode.)

51 If $PECE_4$=1 when a *Power-Saving Mode* instruction is executed, Machine Check, Hypervisor Maintenance, and certain implementation-specific exceptions are enabled to cause exit from power-saving mode; otherwise Machine Check, Hypervisor Maintenance, and the same implementation-specific exceptions are disabled from causing exit from power-saving mode.

It is implementation-specific whether the exceptions enabled by the PECE field cause exit from sleep and rvwinkle power-saving levels. See Section 6.5.1 and Section 6.5.2 for additional information about exit from power-saving mode.

52 ***Mediated External Exception Request*** (MER)

0 A Mediated External exception is not requested.

1 A Mediated External exception is requested.

The exception effects of this bit are said to be consistent with the contents of this bit if one of the following statements is true.

- $LPCR_{MER}$ = 1 and a Mediated External exception exists.
- $LPCR_{MER}$ = 0 and a Mediated External exception does not exist.

A context synchronizing instruction or event that is executed or occurs when $LPCR_{MER}$ = 0 ensures that the exception effects of $LPCR_{MER}$ are consistent with the contents of $LPCR_{MER}$. Otherwise, when an instruction changes the contents of $LPCR_{MER}$, the exception effects of $LPCR_{MER}$ become consistent with the new contents of $LPCR_{MER}$ reasonably soon after the change.

Programming Note

$LPCR_{MER}$ provides a means for the hypervisor to direct an external exception to a partition independent of the partition's MSR_{EE} setting. (When MSR_{EE}=0, it is inappropriate for the hypervisor to deliver the exception.) Using $LPCR_{MER}$, the partition can be interrupted upon enabling external interrupts. Without using $LPCR_{MER}$, the hypervisor must check the state of MSR_{EE} whenever it gets control, which will result in less timely delivery of the exception to the partition.

53 Reserved

54 ***Translation Control*** (TC)

0 The secondary Page Table search is enabled.
1 The secondary Page Table search is disabled.

55:59 Reserved

60 ***Logical Partitioning Environment Selector*** (LPES)

0 External interrupts set the HSRRs, set MSR_{HV} to 1, and leave MSR_{RI} unchanged.
1 External interrupts set the SRRs, set MSR_{RI} to 0, and leave MSR_{HV} unchanged.

Programming Note

$LPES_0$ = 1 should be used by operating systems not running under a hypervisor, so that external interrupts are directed to the SRRs rather than to the HSRRs.

Programming Note

In versions of the architecture that precede Version 2.07, LPES was a two-bit field, in which the second bit controlled significant aspects of storage accessing and interrupt handling.

61:62 Reserved

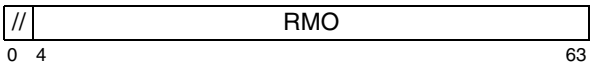
63 ***Hypervisor Decrementer Interrupt Conditionally Enable*** (HDICE)

0 Hypervisor Decrementer interrupts are disabled.
1 Hypervisor Decrementer interrupts are enabled if permitted by MSR_{EE}, MSR_{HV}, and MSR_{PR}; see Section 6.5.12 on page 961.

See Section 6.5 on page 950 for a description of how the setting of LPES affects the processing of interrupts.

2.3 Real Mode Offset Register (RMOR)

The layout of the Real Mode Offset Register (RMOR) is shown in Figure 2 below.



Bits	Name	Description
4:63	RMO	Real Mode Offset

Figure 2. Real Mode Offset Register

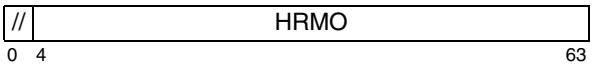
All other fields are reserved.

The supported RMO values are the non-negative multiples of 2^S, where 2^S is the smallest implementation-dependent limit value representable by the contents of the Real Mode Limit Selector field of the LPCR.

The contents of the RMOR affect how some storage accesses are performed as described in Section 5.7.3 on page 891 and Section 5.7.4 on page 895.

2.4 Hypervisor Real Mode Offset Register (HRMOR)

The layout of the Hypervisor Real Mode Offset Register (HRMOR) is shown in Figure 3 below.



Bits	Name	Description
4:63	HRMO	Real Mode Offset

Figure 3. Hypervisor Real Mode Offset Register

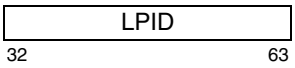
All other fields are reserved.

The supported HRMO values are the non-negative multiples of 2^r, where r is an implementation-dependent value and $12 \le r \le 26$.

The contents of the HRMOR affect how some storage accesses are performed as described in Section 5.7.3 on page 891 and Section 5.7.4 on page 895.

2.5 Logical Partition Identification Register (LPIDR)

The layout of the Logical Partition Identification Register (LPIDR) is shown in Figure 4 below.



Bits	Name	Description
32:63	LPID	Logical Partition Identifier

Figure 4. Logical Partition Identification Register

The contents of the LPIDR identify the partition to which the thread is assigned, affecting operations necessary to manage the coherency of some translation lookaside buffers (see Section 5.10.1 and Chapter 12.). The number of LPIDR bits supported is implementation-dependent.

Programming Note

On some implementations, software must prevent the execution of a ***tlbie*** instruction with an LPID operand value which matches the contents of another thread's LPIDR that is being modified or is the same as the new value being written to the LPIDR. This restriction can be met with less effort if one partition identity is used only on threads on which no ***tlbie*** instruction is ever executed. This partition can be thought of as the transfer partition used exclusively to move a thread from one partition to another.

2.6 Processor Compatibility Register (PCR) [Category: Processor Compatibility]

The layout of the Processor Compatibility Register (PCR) is shown in Figure 5 below.

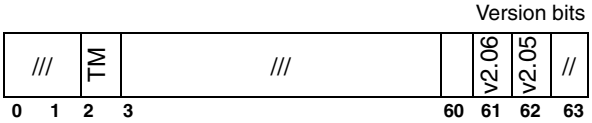


Figure 5. Processor Compatibility Register

Starting from bit 0, high-order PCR bits are assigned to control the availability of certain categories. Starting from bit 63, low-order PCR bits are assigned to control the availability of resources that are new in a specified version of the Architecture. These low-order bits, referred to as the version bits, can change the set of resources provided by a category. For example, since new function is added to VSX category in V 2.07, the VSX, V 2.06, and V 2.05 bits can be set to 0,1,0, respectively, to enable a version of the VSX category that was available in V 2.06.

Each defined bit in the PCR controls whether certain instructions, SPRs, and other related facilities are available in problem state. Except as specified elsewhere in this section, the PCR has no effect on facilities when the thread is not in problem state. Facilities that are made unavailable by the PCR are treated as follows when the thread is in problem state.

- Instructions are treated as illegal instructions,
- SPRs are treated as if they were not defined for the implementation,
- The "reserved SPRs" (see Section 1.3.3 of Book I) are treated as not defined for the implementation,
- Fields in instructions are treated as if they were 0s,
- bits in system registers read back 0s, and ***mtspr*** operations have no effect on their values.
- ***rfebb*** instructions have the same effect on bits in system registers that they would if the bits were available.

Architecture Note

When a bit in a system register is made unavailable by the PCR, ***mtspr*** operations performed on the register in problem state have no effect on the value of the bit regardless of the privilege state in which the register may subsequently be read. When transactional memory is made unavailable by the PCR, however, ***rfebb*** instructions executed in problem state have the same effect on MSR_{TS} as they would if transactional memory were available. This behavior is specified so that illegal transaction state transitions resulting from changes to $BESCR_{TS}$ made by privileged code will cause TM Bad Thing interrupts when ***rfebb*** is executed, thereby facilitating program debug.

A PCR bit may also determine how an instruction field value is interpreted or may define other behavior as specified in the bit definitions below.

The PCR has no effect on the setting of the MSR and [H]SRR1 by interrupts, and by the ***[h]rfid*** and ***mtmsr[d]*** instructions, except as specified elsewhere in this section.

Programming Note

Because the PCR does not prevent ***mtspr***, ***rfscv***, ***[h]rfid***, and ***mtmsr[d]*** instructions from setting bits in system registers that the PCR will make unavailable after a transition to problem state, these instructions may cause interrupts in a variety of unexpected ways. For example, consider an operating system that sets SRR1 such that ***rfid*** returns to problem state with MSR[TS] nonzero. A TM Bad Thing interrupt will result, despite that TM is made unavailable by the PCR.

Similarly, the PCR does not prevent ***rfebb*** instructions from setting bits in system registers that the PCR has made unavailable in problem state, and thus changes to $BESCR_{TS}$ made by privileged code have the potential to subsequently cause illegal transaction state transitions when ***rfebb*** is executed in problem state, resulting in the occurrence of TM Bad Thing interrupts.

When facilities that have enable bits in the MSR, FSCR, HFSCR, or MMCR0 are made unavailable by the value in the PCR, they become unavailable in problem state as specified above regardless of whether they are enabled by the corresponding MSR, FSCR, HFSCR, or MMCR0 bit; facility availability interrupts (e.g. [Hypervisor] Facility Available, Vector Unavailable, etc.) do not occur as a result of problem-state accesses even if the corresponding field in the MSR, [H]FSCR, or MMCR0 makes them unavailable in problem state.

Programming Note

Facilities that can be disabled in problem state by the PCR that also have enable bits in either the MSR or [H]FSCR include Transactional Memory, the BHRB instructions, event-based branch instructions, TAR, DSCR at SPR 3, SIER, MMCR2, the event-based branch instructions, and certain Floating-Point, Vector, and VSX instructions. When any of these facilities are made unavailable in problem state by the PCR, the corresponding [Hypervisor] Facility Unavailable, Floating-Point Unavailable, Vector, or VSX unavailable interrupts do not occur when the facility is accessed in problem state. Note, however, that the PCR does not affect privileged accesses, and thus any Hypervisor Facility Unavailable, Floating-Point Unavailable, Vector unavailable, or VSX unavailable interrupts that are specified to occur as a result of privileged accesses occur regardless of the PCR value.

The bit definitions for the PCR are shown below.

Bit	Description
0:1	**Reserved**
2	**Transactional Memory** (TM) [Category: Transactional Memory] This bit controls the availability, in problem state, of the instructions and facilities in the Transactional Memory category as it was defined in the latest version of the architecture for which new problem-state resources are made available; if the Transactional Memory category was not defined in that version of the architecture, then Transactional Memory instructions and facilities are unavailable. 0 The instructions and facilities in the Transactional Memory category are available in problem state. 1 The instructions and facilities in the Transactional Memory category are unavailable in problem state.

Programming Note

Since facilities in the TM category were not defined in Version 2.06, these facilities are not available in problem state when the v2.06 bit is set to 1 regardless of the value of the TM bit.

3:60	Reserved
61	**Version 2.06** (v2.06) This bit controls the availability, in problem state, of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.06. – ***icbt*** – ***lq, stq lbarx, lharx, stbcx, sthcx*** – ***lqarx, stqarx*** – ***mfbhrbe, clrbhrb*** – ***rfebb, bctar[l]*** – ***mtsle*** – All facilities in category TM – The instructions in Table 1 – The reserved no-op instructions (see Section 1.8.3 of Book I) – The reserved SPRs (see Section 1.3.3 of Book I) – PPR32 – DSCR at SPR number 3 – SIER and MMCR2 – $MMCR0_{42:47,\ 51:55}$ and $MMCRA_{0:64}$.

Programming Note

The specified bits of MMCR0 and MMCRA above cannot be changed by ***mtspr*** instructions and ***mfspr*** instructions return 0s for these bits.

- BESCR, EBBHR, and TAR
- The ability of the ***or 31,31,31*** and ***or 5,5,5*** instructions to change the value of $PPR_{PRI.}$
- The ability of ***mtspr*** instructions that attempt to set PPR_{PRI} to 001 or 101 to change the value of PPR_{PRI}.

0 The listed instructions, facilities, and behaviors are available in problem state.

1 The listed instructions, facilities, and behaviors are unavailable in problem state.

Mnemonic	Instruction Name	Category
bcdadd.	Decimal Add Modulo	VSX
bcdsub.	Decimal Subtract Modulo	VSX
fmrgew	Floating Merge Even Word	VSX
fmrgow	Floating Merge Odd Word	VSX
lxsiwax	Load VSX Scalar as Integer Word Algebraic Indexed	VSX
lxsiwzx	Load VSX Scalar as Integer Word and Zero Indexed	VSX
lxsspx	Load VSX Scalar Single-Precision Indexed	VSX
mfvsrd	Move From VSR Doubleword	VSX
mfvsrwz	Move From VSR Word and Zero	VSX
mtvsrd	Move To VSR Doubleword	VSX
mtvsrwa	Move To VSR Word Algebraic	VSX
mtvsrwz	Move To VSR Word and Zero	VSX
stxsiwx	Store VSX Scalar as Integer Word Indexed	VSX
stxsspx	Store VSX Scalar Single-Precision Indexed	VSX
vaddcuq	Vector Add & write Carry Unsigned Quadword	V
vaddecuq	Vector Add Extended & write Carry Unsigned Quadword	V
vaddeuqm	Vector Add Extended Unsigned Quadword Modulo	V
vaddudm	Vector Add Unsigned Doubleword Modulo	V
vadduqm	Vector Add Unsigned Quadword Modulo	V
vbpermq	Vector Bit Permute Quadword	V
vcipher	Vector AES Cipher	V.Crypto
vcipherlast	Vector AES Cipher Last	V.Crypto
vclzb	Vector Count Leading Zeros Byte	V
vclzd	Vector Count Leading Zeros Doubleword	V
vclzh	Vector Count Leading Zeros Halfword	V
vclzw	Vector Count Leading Zeros Word	V
vcmpequd[.]	Vector Compare Equal To Unsigned Doubleword	V
vcmpgtsd[.]	Vector Compare Greater Than Signed Doubleword	V
vcmpgtud[.]	Vector Compare Greater Than Unsigned Doubleword	V
veqv	Vector Logical Equivalence	V
vgbbd	Vector Gather Bits by Bytes by Doubleword	V
vmaxsd	Vector Maximum Signed Doubleword	V
vmaxud	Vector Maximum Unsigned Doubleword	V
vminsd	Vector Minimum Signed Doubleword	V
vminud	Vector Minimum Unsigned Doubleword	V
vmrgew	Vector Merge Even Word	VSX
vmrgow	Vector Merge Odd Word	VSX
vmulesw	Vector Multiply Even Signed Word	V
vmuleuw	Vector Multiply Even Unsigned Word	V
vmulosw	Vector Multiply Odd Signed Word	V
vmulouw	Vector Multiply Odd Unsigned Word	V
vmuluwm	Vector Multiply Unsigned Word Modulo	V
vnand	Vector Logical NAND	V

Table 1: Category: VSX and Vector Instructions Controlled by the v2.06 Bit

Mnemonic	Instruction Name	Category
vncipher	Vector AES Inverse Cipher	V.Crypto
vncipherlast	Vector AES Inverse Cipher Last	V.Crypto
vorc	Vector Logical OR with Complement	V
vpermxor	Vector Permute and Exclusive-OR	V.RAID
vpksdss	Vector Pack Signed Doubleword Signed Saturate	V
vpksdus	Vector Pack Signed Doubleword Unsigned Saturate	V
vpkudum	Vector Pack Unsigned Doubleword Unsigned Modulo	V
vpkudus	Vector Pack Unsigned Doubleword Unsigned Saturate	V
vpmsumb	Vector Polynomial Multiply-Sum Byte	V
vpmsumd	Vector Polynomial Multiply-Sum Doubleword	V
vpmsumh	Vector Polynomial Multiply-Sum Halfword	V
vpmsumw	Vector Polynomial Multiply-Sum Word	V
vpopcntb	Vector Population Count Byte	V
vpopcntd	Vector Population Count Doubleword	V
vpopcnth	Vector Population Count Halfword	V
vpopcntw	Vector Population Count Word	V
vrld	Vector Rotate Left Doubleword	V
vsbox	Vector AES S-Box	V.Crypto
vshasigmad	Vector SHA-512 Sigma Doubleword	V.Crypto
vshasigmaw	Vector SHA-256 Sigma Word	V.Crypto
vsld	Vector Shift Left Doubleword	V
vsrad	Vector Shift Right Algebraic Doubleword	V
vsrd	Vector Shift Right Doubleword	V
vsubcuq	Vector Subtract & write Carry Unsigned Quadword	V
vsubecuq	Vector Subtract Extended & write Carry Unsigned Quadword	V
vsubeuqm	Vector Subtract Extended Unsigned Quadword Modulo	V
vsubudm	Vector Subtract Unsigned Doubleword Modulo	V
vsubuqm	Vector Subtract Unsigned Quadword Modulo	V
vupkhsw	Vector Unpack High Signed Word	V
vupklsw	Vector Unpack Low Signed Word	V
xsaddsp	VSX Scalar Add Single-Precision	VSX
xscvdpspn	Scalar Convert Double-Precision to Single-Precision format Non-signalling	VSX
xscvdpspn	Scalar Convert Single-Precision to Double-Precision format Non-signalling	VSX
xscvsxdsp	VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision	VSX
xscvsxdsp	VSX Scalar round and Convert Signed Fixed-Point Doubleword to Single-Precision format	VSX
xscvuxdsp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision	VSX
xscvuxdsp	VSX Scalar round and Convert Unsigned Fixed-Point Doubleword to Single-Precision format	VSX
xsdivsp	VSX Scalar Divide Single-Precision	VSX
xsmaddasp	VSX Scalar Multiply-Add Type-A Single-Precision	VSX
xsmaddmsp	VSX Scalar Multiply-Add Type-M Single-Precision	VSX
xsmsubasp	VSX Scalar Multiply-Subtract Type-A Single-Precision	VSX
xsmsubmsp	VSX Scalar Multiply-Subtract Type-M Single-Precision	VSX
xsmulsp	VSX Scalar Multiply Single-Precision	VSX

Table 1: Category: VSX and Vector Instructions Controlled by the v2.06 Bit

Mnemonic	Instruction Name	Category
xsnmaddasp	VSX Scalar Negative Multiply-Add Type-A Single-Precision	VSX
xsnmaddmsp	VSX Scalar Negative Multiply-Add Type-M Single-Precision	VSX
xsnmsubasp	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision	VSX
xsnmsubmsp	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision	VSX
xsresp	VSX Scalar Reciprocal Estimate Single-Precision	VSX
xsrsp	VSX Scalar Round to Single-Precision	VSX
xsrsqrtesp	VSX Scalar Reciprocal Square Root Estimate Single-Precision	VSX
xssqrtsp	VSX Scalar Square Root Single-Precision	VSX
xssubsp	VSX Scalar Subtract Single-Precision	VSX
xxleqv	VSX Logical Equivalence	VSX
xxlnand	VSX Logical NAND	VSX
xxlorc	VSX Logical OR with Complement	VSX

Table 1: Category: VSX and Vector Instructions Controlled by the v2.06 Bit

0 The listed instructions, facilities, and behaviors are available in problem state.
1 The listed instructions, facilities, and behaviors are unavailable in problem state.

62 **Version 2.05** (v2.05)

This bit controls the availability, in problem state, of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.05.

- AMR access using SPR 13
- ***addg6s***
- ***bperm***
- ***cdtbcd, cbcdtd***
- ***dcffix[.]***
- ***divde[o][.], divdeu[o][.], divwe[o][.], divweu[o][.]***
- ***isel***
- ***lfiwzx*** [Category: Floating-Point: Phased-In]
- ***fctidu[.], fctiduz[.], fctiwu[.], fctiwuz[.], fcfids[.], fcfidu[.], fcfidus[.], ftdiv, ftsqrt*** [Category: Floating-Point: Phased-In]
- ***ldbrx, stdbrx*** [Category: 64-bit]
- ***popcntw, popcntd***
- All facilities in Category: VSX

0 The listed instructions, facilities, and behaviors are available in problem state.
1 The listed instructions, facilities, and behaviors are unavailable in problem state.

If this bit is set to 1, then the v2.06 bit must also be set to 1.

63 Reserved

The initial state of the PCR is all 0s.

Programming Note

Because the PCR has no effect on privileged instructions except as specified above, privileged instructions that are available on newer implementations but not available on older implementations will behave differently when the thread is in problem state. On older implementations, either an Illegal Instruction type Program interrupt or a Hypervisor Emulation Assistance interrupt will occur because the instruction is undefined; on newer implementations, a Privileged Instruction type Program interrupt will occur because the instruction is implemented. (On older implementations the interrupt will be an Illegal Instruction type Program interrupt if the implementation complies with a version of the architecture that precedes V. 2.05, or complies with V. 2.05 and does not support the Hypervisor Emulation Assistance category, and will be a Hypervisor Emulation Assistance interrupt otherwise.)

In future versions of the architecture, in general the lowest-order reserved bit of the PCR will be used to control the availability of the instructions and related resources that are new in that version of the architecture; the name of the bit will correspond to the previous version of the architecture (i.e. the newest version in which the instructions and related resources were not available).

In these future versions of the architecture, there will be a requirement that if any bit of the low-order defined bits is set to 1 then all higher-order bits of the defined low-order bits must also be set to 1, and the architecture version with which the implementation appears to comply, in problem state, will be the version corresponding to the name of the lowest-order 1 bit in the set of defined low-order PCR bits, or the current architecture version if none of these bits are 1. Also, in general the highest-order reserved bits will be used to control the availability of sets of instructions and related resources having the requirement that their availability be independent of versions of the architecture.

2.7 Other Hypervisor Resources

In addition to the resources described above, all hypervisor privileged instructions as well as the following resources are hypervisor resources, accessible to software only when the thread is in hypervisor state except as noted below.

- All implementation-specific resources except for privileged non-hypervisor implementation-specific SPRs. (See Section 4.4.4 for the list of the implementation-specific SPRs that are allowed to be privileged non-hypervisor SPRs.) Implementa-

tion-specific registers include registers (e.g., "HID" registers) that control hardware functions or affect the results of instruction execution. Examples include resources that disable caches, disable hardware error detection, set breakpoints, control power management, or significantly affect performance.

- ME bit of the MSR
- SPRs defined as hypervisor-privileged in Section 4.4.4. (Note: Although the Time Base, the PURR, and the SPURR can be altered only by a hypervisor program, the Time Base can be read by all programs and the PURR and SPURR can be read when the thread is in privileged state.)

The contents of a hypervisor resource can be modified by the execution of an instruction (e.g., ***mtspr***) only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). An attempt to modify the contents of a given hypervisor resource, other than MSR_{ME}, in privileged but non-hypervisor state ($MSR_{HV\ PR}$ = 0b00) causes a Privileged Instruction type Program interrupt. An attempt to modify MSR_{ME} in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

Programming Note

Because the SPRs listed above are privileged for writing, an attempt to modify the contents of any of these SPRs in problem state (MSR_{PR}=1) using ***mtspr*** causes a Privileged Instruction type Program exception, and similarly for MSR_{ME}.

2.8 Sharing Hypervisor Resources

Shared SPRs are SPRs that are accessible to multiple threads. Changes to shared SPRs made by one thread are immediately readable (using ***mfspr***) by all other threads sharing the SPR.

The LPIDR and DPDES must appear to software to be shared among threads of a sub-processor (see Section 2.9). If the implementation does not support sub-processors, the LPIDR and DPDES must be shared among all threads of the multi-threaded processor. <S.PC>The DHDES must be shared among threads of the multi-threaded processor.

Certain other hypervisor resources may be shared among threads. Programs that modify these resources must be aware of this sharing, and must allow for the fact that changes to these resources may affect more than one thread.

The following resources may be shared among threads.

- RMOR (see Section 2.3)
- HRMOR (see Section 2.4)
- LPIDR (see Section 2.5)
- PCR [Category: Processor Control] (see Section 2.6)
- PVR (see Section 4.3.1)
- RPR (see Section 4.3.7)
- SDR1 (see Section 5.7.7.2)
- AMOR (see Section 5.7.9.1)
- HMEER (see Section 6.2.9)
- Time Base (see Section 7.2)
- Virtual Time Base (see Section 7.3)
- Hypervisor Decrementer (see Section 7.5)
- certain implementation-specific registers or implementation-specific fields in architected registers

The set of resources that are shared is implementation-dependent.

Threads that share any of the resources listed above, with the exception of the PVR and the HRMOR, must be in the same partition.

For each field of the LPCR, except the AIL, ONL, HDICE, and MER fields, software must ensure that the contents of the field are identical among all threads that are in the same partition and are in a state such that the contents of the field could have side effects. (E.g., software must ensure that the contents of $LPCR_{LPES}$ are identical among all threads that are in the same partition and are not in hypervisor state.) For the HDICE field, software must ensure that the contents of the field are identical among all threads that share the Hypervisor Decrementer and are in a state such that the contents of the field could have side effects. There are no identity requirements for the other fields listed in the first sentence of this paragraph.

2.9 Sub-Processors

Hardware is allowed to sub-divide a multi-threaded processor into "sub-processors" that appear to privileged programs as multi-threaded processors with fewer threads. Such a multi-threaded processor appears to the hypervisor as a processor with a number of threads equal to the sum of all sub-processor threads, and in which the LPIDR for each sub-processor must appear to be shared among all threads of that sub-processor.

2.10 Thread Identification Register (TIR) [Category: Server.Processor Control]

The TIR is a 64-bit read-only register that contains the thread number, which is a binary number corresponding to the thread.

For implementations that do not support sub-processors, the thread number of a thread is unique among all thread numbers of threads on the multi-threaded processor.

For implementations that support sub-processors, the value of this register depends on whether it is read in hypervisor or privileged, non-hypervisor state as follows.

- When this register is read in privileged, non-hypervisor state, the thread number is unique among all thread numbers of threads on the sub-processor.
- When this register is read in hypervisor state, the thread number is unique among all thread numbers of threads on the multi-threaded processor.

Threads are numbered sequentially, with valid values ranging from 0 to t-1, where t is the number of threads implemented. A thread for which TIR = n is referred to as “thread n.”

The layout of the TIR is shown below.

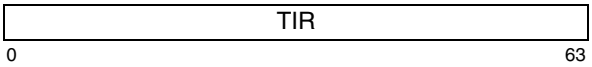


Figure 6. Thread Identification Register

Access to the TIR is privileged.

Since the thread number contained in this register is different if it is read in hypervisor from when it is read in privileged, non-hypervisor state in implementations that support sub-processors, the following conventions are used.

- The value returned in privileged, non-hypervisor state is referred to as the “privileged thread number.”
- The value returned in hypervisor state is referred to as the “hypervisor thread number.”

2.11 Hypervisor Interrupt Little-Endian (HILE) Bit

The Hypervisor Interrupt Little-Endian (HILE) bit is a bit in an implementation-dependent register or similar mechanism. The contents of the HILE bit are copied into MSR_{LE} by interrupts that set MSR_{HV} to 1 (see Section 6.5), to establish the Endian mode for the interrupt handler. The HILE bit is set, by an implementation-dependent method, during system initialization, and cannot be modified after system initialization.

The contents of the HILE bit must be the same for all threads under the control of a given instance of the hypervisor; otherwise all results are undefined.

Chapter 3. Branch Facility

3.1 Branch Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Facility that are not covered in Book I.

3.2 Branch Facility Registers

3.2.1 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the thread. On interrupt, the MSR bits are altered in accordance with Figure 51 on page 951. The MSR can also be modified by the ***mtmsr***[***d***], ***rfid***, and ***hrfid*** instructions. It can be read by the ***mfmsr*** instruction.

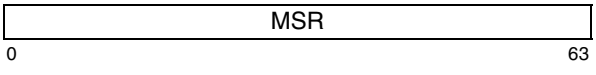


Figure 7. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Bit **Description**

0 ***Sixty-Four-Bit Mode*** (SF)

- 0 The thread is in 32-bit mode.
- 1 The thread is in 64-bit mode.

1:2 Reserved

3 ***Hypervisor State*** (HV)

- **0** The thread is not in hypervisor state.
- **1** If MSR_{PR}=0 the thread is in hypervisor state; otherwise the thread is not in hypervisor state.

Programming Note

The privilege state of the thread is determined by MSR_{HV} and MSR_{PR}, as follows.

HV	PR	
0	0	privileged
0	1	problem
1	0	hypervisor
1	1	problem

Hypervisor state is also a privileged state (MSR_{PR} = 0). All references to "privileged state" in the Books include hypervisor state unless otherwise stated or if it is obvious from the context.

MSR_{HV} can be set to 1 only by the *System Call* instruction and some interrupts. It can be set to 0 only by ***rfid*** and ***hrfid***.

It is possible to run an operating system in an environment that lacks a hypervisor, by always having MSR_{HV} = 1 and using MSR_{HV} || MSR_{PR} = 10 for the operating system (effectively, the OS runs in hypervisor state) and MSR_{HV} || MSR_{PR} = 11 for applications.

4 Reserved

5 ***Split Little Endian*** (SLE)

- 0 Instruction and data storage accesses for the thread are the same and use the value specified by MSR_{LE}.
- 1 Instruction and data storage accesses for the thread are opposite. Instruction storage accesses use the value specified by MSR_{LE}. Data stroage accesses use the value specified by ¬MSR_{LE}.

Programming Note

The only instructions that can alter MSR_{SLE} are ***mtsle***, ***hrfid***, ***rfid***, and ***mtmsrd***.

6:28 Reserved

29:30 ***Transaction State*** (TS) [Category: Transactional Memory]

00 Non-transactional
01 Suspended
10 Transactional
11 Reserved

Changes to MSR[TS] that are caused by Transactional Memory instructions, and by invocation of the transaction's failure handler, take effect immediately (even though these instructions and events are not context synchronizing).

31 ***Transactional Memory Available*** (TM) [Category: Transactional Memory]

0 The thread cannot execute any Transactional Memory instructions or access any Transactional Memory registers.
1 The thread can execute Transactional Memory instructions and access Transactional Memory registers unless the Transactional Memory facility has been made unavailable by some other register.

32:37 Reserved

38 ***Vector Available*** (VEC) [Category: Vector]

0 The thread cannot execute any vector instructions, including vector loads, stores, and moves.
1 The thread can execute vector instructions unless they have been made unavailable by some other register.

39 Reserved

40 ***VSX Available*** (VSX)

0 The thread cannot execute any VSX instructions, including VSX loads, stores, and moves.
1 The thread can execute VSX instructions unless they have been made unavailable by some other register.

Programming Note

An application binary interface defined to support Category: Vector-Scalar operations should also specify a requirement that `MSR.FP` and `MSR.VEC` be set to 1 whenever `MSR.VSX` is set to 1.

41:47 Reserved

48 ***External Interrupt Enable*** (EE)

0 External, Decrementer, Performance Monitor<S>, and Privileged Doorbell interrupts <S.PC.> are disabled.
1 External, Decrementer, Performance Monitor<S>, and Privileged Doorbell interrupts <S.PC.> are enabled.

This bit also affects whether Hypervisor Decrementer, Hypervisor Maintenance, and Directed Hypervisor Doorbell <S.PC.> interrupts are enabled; see Section 6.5.12 on page 961, Section 6.5.19 on page 965, and Section 6.5.20 on page 966.

49 ***Problem State*** (PR)

0 The thread is in privileged state.
1 The thread is in problem state.

Programming Note

Any instruction that sets MSR_{PR} to 1 also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

50 ***Floating-Point Available*** (FP)
[Category: Floating-Point]

0 The thread cannot execute any floating-point instructions, including floating-point loads, stores, and moves.
1 The thread can execute floating-point instructions unless they have been made unavailable by some other register.

51 ***Machine Check Interrupt Enable*** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

This bit is a hypervisor resource; see Chapter 2., “Logical Partitioning (LPAR) and Thread Control”, on page 843.

Programming Note

The only instructions that can alter MSR_{ME} are ***rfid*** and ***hrfid***.

52 ***Floating-Point Exception Mode 0*** (FE0)
[Category: Floating-Point]

See below.

53 ***Single-Step Trace Enable*** (SE)
[Category: Trace]

0 The thread executes instructions normally.
1 The thread generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is a *Power-Saving Mode* instruction, ***hrfid, rfid*** or a *Power-Saving Mode* instruction, all of which are never traced. Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state <TM>, is not one of the instructions that is forbidden in Transactional state (e.g., ***dcbf***; see Section 5.3.1 of Book II).

54 ***Branch Trace Enable*** (BE)
[Category: Trace]

0 The thread executes branch instructions normally.
1 The thread generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken.

Branch tracing need not be supported on all implementations that support the Trace category. If the function is not implemented, this bit is treated as reserved.

55 ***Floating-Point Exception Mode 1*** (FE1)
[Category: Floating-Point]

See below.

56:57 Reserved

58 ***Instruction Relocate*** (IR)

0 Instruction address translation is disabled.
1 Instruction address translation is enabled.

Programming Note

See the Programming Note in the definition of MSR_{PR}.

59 ***Data Relocate*** (DR)

0 Data address translation is disabled. Effective Address Overflow (EAO) (see Book I) does not occur.
1 Data address translation is enabled. EAO causes a Data Storage interrupt.

Programming Note

See the Programming Note in the definition of MSR_{PR}.

60 Reserved

61 ***Performance Monitor Mark*** (PMM)
[Category: Server]

Programming Note

Software can use this bit as a process-specific marker which, in conjunction with $MMCR0_{FCM0\ FCM1}$ (see Section 9.4.4) and MMCR2 (see Section 9.4.6), permits events to be counted on a process-specific basis. (The bit is saved by interrupts and restored by ***rfid***.)

Common uses of the PMM bit include the following.

- All counters count events for a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=1
  - $MMCR0_{FCM1}$=0
  - MMCR2 = 0x0000
- All counters count events for all but a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=0
  - $MMCR0_{FCM1}$=1
  - MMCR2 = 0x0000

Notice that for both of these uses a mark value of 1 identifies the “few” processes and a mark value of 0 identifies the remaining “many” processes. Because the PMM bit is set to 0 when an interrupt occurs (see Figure 51 on page 951), interrupt handlers are treated as one of the “many”. If it is desired to treat interrupt handlers as one of the “few”, the mark value convention just described would be reversed.

If only a specific counter n is to be frozen, $MMCR0_{FCM0\ FCM1}$ is set to 0b00, and $MMCR2_{FCnM0}$ and $MMCR2_{FCnM1}$ instead of $MMCR0_{FCM0}$ and $MMCR0_{FCM1}$ are set to the values described above.

62 ***Recoverable Interrupt*** (RI)

0 Interrupt is not recoverable.
1 Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 6.4.3, “Interrupt Processing” on page 947, 6.5.1, “System Reset Interrupt” on page 952, and 6.5.2, “Machine Check Interrupt” on page 953.

63 ***Little-Endian Mode*** (LE)

0 Instruction and data storage accesses for the the thread are in Big-Endian mode when MSR_{SLE}=0. When MSR_{SLE}=1, instruction storage accesses are in Big-Endian mode and data storage accesses are in Little-Endian mode.

1 Instruction and data storage accesses for the thread are in Little-Endian mode when MSR_{SLE}=0. When MSR_{SLE}=1, instruction storage accesses are in Little-Endian mode and data storage accesses are in Big-Endian mode.

Programming Note

The table below illustrates the Endian modes for all combinations of SLE and LE.

SLE	LE	Data	Instruction
0	0	Big	Big
0	1	Little	Little
1	0	Little	Big
1	1	Big	Little

Programming Note

The only instructions that can alter MSR_{LE} are ***rfid*** and ***hrfid***.

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

FE0	FE1	Mode
0	0	Ignore Exceptions
0	1	Imprecise Nonrecoverable
1	0	Imprecise Recoverable
1	1	Precise

3.2.2 State Transitions Associated with the Transactional Memory Facility [Category: Transactional Memory]

Updates to MSR_{TS} and MSR_{TM} caused by ***rfebb***, ***rfid***, ***hrfid***, or ***mtmsrd*** occur as described in Table 2. The value written, and whether or not the instruction causes an interrupt, are dependent on the current values of MSR_{TS} and MSR_{TM}, and the values being written to these fields. When the setting of MSR_{TS} causes an illegal state transition, a TM Bad Thing type Program interrupt is generated.

Programming Note

The transition rules are the same for ***mtmsrd*** as for the ***rfid***-type instructions because if a transition were illegal for ***mtmsrd*** but allowed for ***rfid***, or vice versa, software could use the instruction for which the transition is allowed to achieve the effect of the other instruction.

Table 2 shows all the Transaction State transitions that can be requested by ***rfebb***, ***rfid***, ***hrfid***, and ***mtmsrd***. The table covers behavior when TM is enabled by the PCR. For causes of the TM Bad Thing exception when TM is disabled by the PCR, see Section 6.5.9. In the table, the contents of MSR_{TS} and MSR_{TM} are abbreviated in the form AB, where A represents MSR_{TS} (N, T or S) and B represents MSR_{TM} (0 or 1). “x” in the “B” position means that the entry covers both MSR_{TM} values, with the same value applying in all columns of a given row for a given instance of the transition. (E.g., the first row means that the transition from N0 to N0 is allowed and results in N0, and that the transition from N0 to N1 is allowed and results in N1.) “Input MSR_{TS}M-SR_{TM}” in the second column refers to the MSR_{TS} and MSR_{TM} values supplied by BESCR for ***rfebb*** (just the TS value), SRR1 for ***rfid***, HSRR1 for ***hrfid***, or register RS for ***mtmsrd***.

Current $MSR_{TS}MSR_{TM}$	Input $MSR_{TS}MSR_{TM}$	Resulting MSR_{TS} MSR_{TM}	Comments
N0	Nx	Nx	May occur in the context of a Transactional Memory type of Facility Unavailable interrupt handler, enabling/disabling transactions for user-level applications.
	All others - Illegal[1]	N0	
T0	N/A		Unreachable state
S0	N0[2]	S0	Operating system code that is not TM aware may attempt to set TS and TM to zero, thinking they're reserved bits. Change is suppressed.
	T1	T1	May occur at an ***rfid*** returning to an application whose transaction was suspended on interrupt.
	Sx	Sx	This case may occur for an ***rfid*** returning to an application whose suspended transaction was interrupted.
	All others - Illegal[1]	S0	
N1	Nx	Nx	After a ***treclaim***, the OS dispatches Nx program.
	All others -IIIlegal[1]	N0	
T1	all	N1	Disallowed instructions in Transactional state
S1	T1	T1	May occur after ***trechkpt.*** when returning to an application.
	Sx	Sx	
	All others - Illegal[1]	S0	

Notes:
1. Generate TM Bad Thing type Program interrupt. "All others" includes all attempts to set MSR_{TS} to 0b11 (reserved value).
2. Instruction completes, change to MSR_{TM} suppressed, except when attempted by ***rfebb***, in which case the result is a TM Bad Thing type Program interrupt.

Table 2: Transaction state transitions that can be requested by ***rfebb, rfid***, ***hrfid***, and ***mtmsrd***.

Programming Note

For ***[h]rfid***, and ***mtmsrd***, the attempted transition from S0 to N0 is suppressed in order that interrupt handlers that are "unaware" of transactional memory, and load an MSR value that has not been updated to take account of transactional memory, will continue to work correctly. (If the interrupt occurs when a transaction is running or suspended, the interrupt will set MSR[TS || TM] to S0. If the interrupt handler attempts to load an MSR value that has not been updated to take account of transactional memory, that MSR value will have TS || TM = N0. It is desirable that the interrupt handler remain in state S0, so that it can return normally to the interrupted transaction.)

The problem solved by suppressing this transition does not apply to ***rfebb***, so for ***rfebb*** an attempt to transition from S0 to N0 is not suppressed, and instead causes a TM Bad Thing type Program interrupt.

3.3 Branch Facility Instructions

3.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The *System Call* instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

System Call *SC-form*

sc LEV

17	///	///	//	LEV	//	1	/
0	6	11	16	20	27	30	31

SRR0 $\leftarrow_{iea}$ CIA + 4
$SRR1_{33:36\ 42:47} \leftarrow 0$
$SRR1_{0:32\ 37:41\ 48:63} \leftarrow MSR_{0:32\ 37:41\ 48:63}$
MSR ← new_value (see below)
NIA ← 0x0000_0000_0000_0C00

The effective address of the instruction following the *System Call* instruction is placed into SRR0. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of SRR1, and bits 33:36 and 42:47 of SRR1 are set to zero.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 6.5, “Interrupt Definitions” on page 950. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 1 are reserved. Bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

This instruction is context synchronizing.

Special Registers Altered:

SRR0 SRR1 MSR

Programming Note

If LEV=1 the hypervisor is invoked. This is the only way that executing an instruction can cause hypervisor state to be entered.

Because this instruction is not privileged, it is possible for application software to invoke the hypervisor. However, such invocation should be considered a programming error.

Programming Note

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an ***sc*** mnemonic with one operand as the basic form, and an ***sc*** mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

Return From Interrupt Doubleword XL-form

rfid

19	///	///	///	18	/
0	6	11	16	21	31

```
MSR51 ← (MSR3 & SRR151) | ((¬MSR3) & MSR51)
MSR3 ← MSR3 & SRR13
if (MSR29:31 ¬= 0b010 | SRR129:31 ¬= 0b000) then
   MSR29:31 ← SRR129:31
MSR48 ← SRR148 | SRR149
MSR58 ← SRR158 | SRR149
MSR59 ← SRR159 | SRR149
MSR0:2 4:28 32 37:41 49:50 52:57 60:63←SRR10:2 4:28 32 37:41 49:50 52:57
60:63
NIA ←iea SRR00:61 || 0b00
```

If MSR_3=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of SRR1 are not equal to 0b000, then the value of bits 29 through 31 of SRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of SRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of SRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of SRR1 is placed into MSR_{59}. Bits 0:2, 4:28, 32, 37:41, 49:50, 52:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, “Transaction state transitions that can be requested by rfebb, rfid, hrfid, and mtmsrd.,” on page 861), or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the ***rfid*** instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $SRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $SRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

Hypervisor Return From Interrupt Doubleword *XL-form*

hrfid

19	///	///	///	274	/
0	6	11	16	21	31

```
if (MSR29:31 ¬= 0b010 | HSRR129:31 ¬= 0b000) then
   MSR29:31 ← HSRR129:31

MSR48 ← HSRR148 | HSRR149
MSR58 ← HSRR158 | HSRR149
MSR59 ← HSRR159 | HSRR149
MSR0:28 32 37:41 49:57 60:63 ← HSRR10:28 32 37:41 49:57 60:63
NIA ←iea HSRR00:61 || 0b00
```

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of HSRR1 are not equal to 0b000, then the value of bits 29 through 31 of HSRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of HSRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of HSRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of HSRR1 is placed into MSR_{59}. Bits 0:28, 32, 37:41, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, “Transaction state transitions that can be requested by rfebb, rfid, hrfid, and mtmsrd.,” on page 861), or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the *hrfid* instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $HSRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $HSRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:
MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

3.3.2 Power-Saving Mode Instructions

The *Power-Saving Mode* instructions provide a means by which the hypervisor can put the thread into power-saving mode. When the thread is in power-saving mode it does not execute instructions, and it may consume less power than it would consume when it is not in power-saving mode.

There are four levels of power-savings, called doze, nap, sleep, and rvwinkle. For each level in this list, the power consumed is less than or equal to the power consumed in the preceding level, and the time required for the thread to exit from the level and for software then to resume normal operation is greater than or equal to the corresponding time for the preceding level. Doze power-saving level requires a minimum amount of such time, while the other levels may require more time. Resources other than those listed in the instruction descriptions that are maintained in each level other than doze, and the actions required by the hypervisor in order for software to resume normal operation after the thread exits from power-saving mode, are implementation-specific.

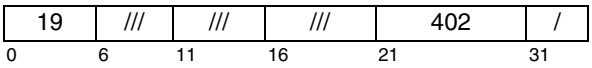
Read-only resources (including the HILE bit) are maintained in all power-saving levels. Descriptions of resource state loss in the *Power-Saving Mode* instruction descriptions do not apply to read-only resources.

Programming Note

The hypervisor determines which power-saving level to enter based on how responsive the system needs to be. If the hypervisor decides that some loss of state is acceptable, it can use the ***nap*** instruction rather than the ***doze*** instruction, and when the thread exits from power-saving mode the hypervisor can quickly determine whether any resources need to be restored.

Doze *XL-form*

doze



The thread is placed into doze power-saving level.

When the thread is in doze power-saving level, the state of all thread resources is maintained as if the thread was not in power-saving mode.

When the interrupt that causes exit from doze power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

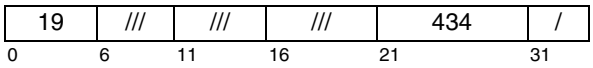
This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

None

Nap *XL-form*

nap



The thread is placed into nap power-saving level.

When the thread is in nap power-saving level, the state of the Decrementer and all hypervisor resources is maintained as if the thread was not in power-saving mode, and sufficient information is maintained to allow the hypervisor to resume execution.

When the interrupt that causes exit from nap power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

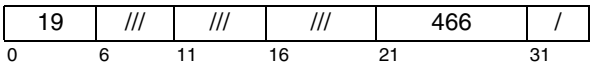
None

Programming Note

If the state of the Decrementer were not maintained and updated as if the thread was not in power-saving mode, Decrementer exceptions would not reliably cause exit from nap power-saving level even if Decrementer exceptions were enabled to cause exit.

Sleep *XL-form*

sleep



The thread is placed into sleep power-saving level.

When the thread is in sleep power-saving level, the state of all resources may be lost except for the HRMOR.

When the interrupt that causes exit from sleep power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

None

Programming Note

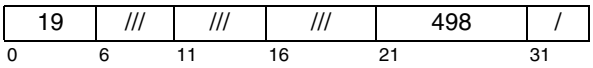
If the state of the Decrementer is not maintained and updated, in sleep or rvwinkle power-saving level, as if the thread was not in power-saving mode, Decrementer exceptions will not reliably cause exit from power-saving mode even if Decrementer exceptions are enabled to cause exit.

Note

See the Notes that appear in the ***rvwinkle*** instruction description.

Rip Van Winkle *XL-form*

rvwinkle



The thread is placed into rvwinkle power-saving level.

When the thread is in rvwinkle power-saving level, the state of all resources may be lost except for the HRMOR.

When the interrupt that causes exit from rvwinkle power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

None

Programming Note

In the short story by Washington Irving, Rip Van Winkle is a man who fell asleep on a green knoll and awoke twenty years later.

Note

See the Notes that appear in the ***sleep*** instruction description.

3.3.2.1 Entering and Exiting Power-Saving Mode

In order to enter power-saving mode, the hypervisor must use the instruction sequence shown below. Before executing this sequence, the hypervisor must ensure that $LPCR_{MER}$ contains the value 0, the $LPCR_{PECE}$ contains the desired value if doze or nap power-saving level is to be entered, MSR_{SF}, MSR_{HV}, and MSR_{ME} contain the value 1, and all other bits of the MSR contain the value 0 except for MSR_{RI}, which may contain either 0 or 1. Depending on the implementation and on the power-saving mode being entered, it may also be necessary for the hypervisor to save the state of certain resources before entering the sequence. The sequence must be exactly as shown, with no intervening instructions, except that any GPR may be used as Rx and as Ry, and any value may be used for "save_area" provided the resulting effective address is double-word aligned and corresponds to a valid real address.

```
   std  Rx,save_area(Ry)    /* last store neces-*/
                            /* sary to save state*/
   ptesync                  /* order load after*/
                            /* last store      */
   ld   Rx,save_area(Ry)    /* reload from last */
                            /* store location, */
                            /* for synchro-    */
                            /* nization        */
loop:
   cmp   Rx,Rx              /* create dependency */
   bne   loop
   nap/doze/sleep/rvwinkle /* enter power- */
                            /* saving mode    */
b  $                        /* branch to self */
```

After the thread has entered power-saving mode as specified above, various exceptions may cause exit from power-saving mode. The exceptions include, System Reset, Machine Check, Decrementer, External, Hypevisor Maintenance, and implementation-specific exceptions. Upon exit from power-saving mode, if the exception was a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the type of exception that caused exit from power-saving mode. See Section 6.5.1 for additional information.

Programming Note

The ***ptesync*** instruction (see Book III-S, Section 5.9.2) in the preceding sequence, in conjunction with the ***ld*** instruction and the loop, ensure that all storage accesses associated with instructions preceding the ***ptesync*** instruction, and all Reference, and Change bit updates associated with additional address translations that were performed, by the thread executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, have been performed with respect to all threads and mechanisms, to the extent required by the associated Memory Coherence Required attributes, before the thread enters power-saving mode. The ***b*** instruction (branch to self) is not executed since the preceding *Power-Saving Mode* instruction puts the thread in a power-saving mode in which instructions are not executed. Even though it is not executed, requiring it to be present simplifies implementation and testing because it reduces the synchronization needed between execution of the instruction stream and entry into power-saving mode.

If the performance monitor is in use when the thread enters power-saving mode, the Performance Monitor data obtainable when the thread exits from power-saving mode may be incomplete or otherwise misleading.

Programming Note

Software is not required to set the RI bit to any particular value prior to entering power-saving mode because the setting of $SRR1_{62}$ upon exit from power-saving mode is independent of the value of the RI bit upon entry into power-saving mode.

3.4 Event-Based Branch Facility and Instruction

The Event-Based Branch facility is described in Chapter 7 of Book II, but only at the level required by the application program.

Event-based branches can only occur in problem state and when event-based branches has been enabled in the FSCR and HFSCR. If MSR_{PR}=0, $BESCR_{GE}$=1, and an event-based exception exists, the corresponding event-based branch does not occur until MSR_{PR}=1, $FSCR_{EBB}$=1, and $HFSCR_{EBB}$=1.

If the ***rfebb*** instruction attempts to cause a transition to Transactional or Suspended state when PCR_{TM}=1 or an illegal transaction state transition (see Section 3.2.2), a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism is the address of the ***rfebb*** instruction.)

Chapter 4. Fixed-Point Facility

4.1 Fixed-Point Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Facility that are not covered in Book I.

4.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 885) and ***mtspr*** (page 884) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

4.3 Fixed-Point Facility Registers

4.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the implementation. The contents of the PVR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PVR is privileged; write access is not provided.

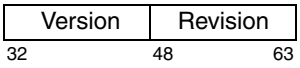


Figure 8. Processor Version Register

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

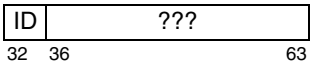
Version A 16-bit number that identifies the version of the implementation. Different version numbers indicate major differences between implementations, such as which categories are supported.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between implementations having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA process. Revision numbers are assigned by an implementation-defined process.

4.3.2 Chip Information Register

The Chip Information Register (CIR) is a 32-bit read-only register that contains a value identifying the manufacturer and other characteristics of the chip on which the processor is implemented. The contents of the CIR can be copied to a GPR by the ***mfspr*** instruction. Read access to the CIR is privileged; write access is not provided.



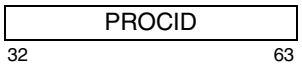
Bit	Description
32:35	**Manufacturer ID** (ID) A four-bit field that identifies the manufacturer of the chip.
36:63	Implementation-dependent.

Figure 9. Chip Information Register

4.3.3 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the thread from other threads in the system. The contents of the PIR can be copied to a GPR by the

mfspr instruction. Read access to the PIR is privileged; write access is not provided.



Bits	Name	Description
0:31	PROCID	Thread ID

Figure 10. Processor Identification Register

The means by which the PIR is initialized are implementation-dependent.

The PIR is a hypervisor resource; see Chapter 2.

4.3.4 Control Register

The Control Register (CTRL) is a 32-bit register as shown below.

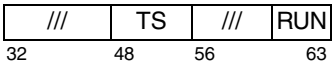


Figure 11. Control Register

The field definitions for the CTRL are shown below.

Bit(s)	Description
32:47	Reserved
48:55	**Thread State** (TS) Problem State Access Reserved Privileged accesses Bits 0:7 of this field are read-only bits that indicate the state of $CTRL_{RUN}$ for threads with privileged thread numbers 0 through 7, respectively; bits corresponding to privileged thread numbers higher than the maximum privileged thread number supported are set to 0s. Hypervisor accesses Bits 0:7 of this field are read-only bits that indicate the state of $CTRL_{RUN}$ for threads with hypervisor thread numbers 0 through 7, respectively; bits corresponding to hypervisor thread numbers higher than the maximum hypervisor thread number supported are set to 0s.
56:62	Reserved
63	RUN This bit controls an external I/O pin. This signal may be used for the following: ■ driving the RUN Light on a system operator panel ■ Direct External exception routing ■ Performance Monitor Counter incrementing *(* See Chapter 9.) The RUN bit can be used by the operating system to indicate when the thread is doing useful work.

Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

4.3.5 Program Priority Register

Privileged programs may set a wider range of program priorities in the PRI field of PPR and PPR32 than may be set by problem-state programs (see Chapter 3 of Book II). Problem-state programs may only set values in the range of 0b001 to 0b100 unless the Problem

State Priority Boost register (see Section 4.3.6) allows the value 0b101. Privileged programs may set values in the range of 0b001 to 0b110. Hypervisor software may also set 0b111. For all priorities except 0b101, if a program attempts to set a value that is not allowed for its privilege level, the PRI field remains unchanged. If a problem-state program attempts to set its priority value to 0b101 when this priority value is not allowed for problem state programs, the priority is set to 0b100. The values and their corresponding meanings are as follows.

Bit(s)	Description
11:13	***Program Priority*** (PRI) 001 very low 010 low 011 medium low 100 medium 101 medium high 110 high 111 very high

4.3.6 Problem State Priority Boost Register

The Problem State Priority Boost (PSPB) register is a 32-bit register that controls whether problem-state programs have access to program priority medium high. (See Section 3.1 of Book II.)

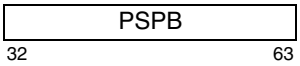


Figure 12. Problem-State Priority Boost Register

A problem state program is able to set the program priority to medium high only when the PSPB of the thread contains a non-zero value.

The maximum value to which the PSPB can be set must be a power of 2 minus 1. Bits that are not required to represent this maximum value must return 0s when read regardless of what was written to them.

When the PSPB is set to a value less than its maximum value but greater than 0, its contents decrease monotonically at the same rate as the SPURR until its contents minus the amount it is to be decreased are 0 or less when a problem-state program is executing on the thread at a priority of medium high.When the contents of the PSPB minus the amount it is to be decreased are 0 or less, its contents are replaced by 0.

When the PSPB is set to its maximum value or 0, its contents do not change until it is set to a different value.

Whenever the priority of a thread is medium high and either of the following conditions exist, hardware changes the priority to medium:

- the PSPB counts down to 0, or
- PSPB=0 and the privilege state of the thread is changed to problem state (MSR_{PR}=1).

4.3.7 Relative Priority Register

The Relative Priority Register (RPR) is a 64-bit register that allows the hypervisor to control the relative priorities corresponding to each valid value of PPR_{PRI}.

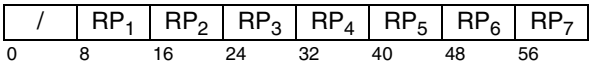


Figure 13. Relative Priority Register

Each RP_n field is defined as follows.

Bits	Meaning
0:1	Reserved
2:7	**Relative priority of priority level n:** Specifies the relative priority that corresponds to the priority corresponding to PPR_{PRI}=n, where a value of 0 indicates the lowest relative priority and a value of 0b111111 indicates the highest relative priority.

Programming Note

The hypervisor must ensure that the values of the RP_n fields increase monotonically for each n and are of different enough magnitudes to ensure that each priority level provides a meaningful difference in priority.

4.3.8 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

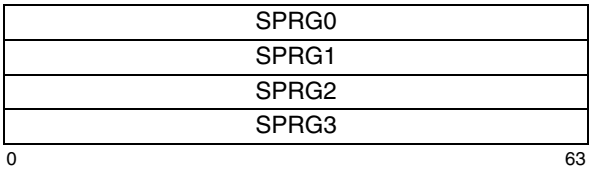


Figure 14. Software-use SPRs

SPRG0, SPRG1, and SPRG2 are privileged registers. SPRG3 is a privileged register except that the contents may be copied to a GPR in Problem state when accessed using the ***mfspr*** instruction.

Programming Note

Neither the contents of the SPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by non-hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per thread save areas).

Operating systems must ensure that no sensitive data are left in SPRG3 when a problem state program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between problem state programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in problem state.

HSPRG0 and HSPRG1 are 64-bit registers provided for use by hypervisor programs.

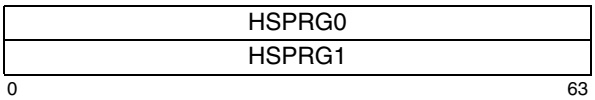


Figure 15. SPRs for use by hypervisor programs

Programming Note

Neither the contents of the HSPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per thread save areas).

4.4 Fixed-Point Facility Instructions

4.4.1 Fixed-Point Load and Store Caching Inhibited Instructions

The storage accesses caused by the instructions described in this section are performed as though the specified storage location is Caching Inhibited and Guarded. The instructions can be executed only in hypervisor state. Software must ensure that the specified storage location is not in the caches. If the specified storage location is in a cache, the results are undefined.

The *Fixed-Point Load and Store Caching Inhibited* instructions must be executed only when MSR_{DR}=0. The storage location specified by the instructions must not be in storage specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. If either of these conditions is violated, the result is a data storage interrupt.

Programming Note

The instructions described in this section can be used to permit a control register on an I/O device to be accessed without permitting the corresponding storage location to be copied into the caches.

See also, in Book I, the introductions to Section 3.3.1, “Fixed-Point Storage Access Instructions”, Section 3.3.2, “Fixed-Point Load Instructions”, and Section 3.3.3, “Fixed-Point Store Instructions”.

Load Byte and Zero Caching Inhibited Indexed X-form

lbzcix RT,RA,RB

31	RT	RA	RB	853	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 56 0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Halfword and Zero Caching Inhibited Indexed X-form

lhzcix RT,RA,RB

31	RT	RA	RB	821	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48 0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Word and Zero Caching Inhibited Indexed X-form

lwzcix RT,RA,RB

31	RT	RA	RB	789	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Doubleword Caching Inhibited Indexed X-form

ldcix RT,RA,RB

31	RT	RA	RB	885	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Byte Caching Inhibited Indexed X-form

stbcix RS,RA,RB

31	RS	RA	RB	981	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Halfword Caching Inhibited Indexed X-form

sthcix RS,RA,RB

31	RS	RA	RB	949	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)48:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Word Caching Inhibited Indexed X-form

stwcix RS,RA,RB

31	RS	RA	RB	917	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)32:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Doubleword Caching Inhibited Indexed X-form

stdcix RS,RA,RB

31	RS	RA	RB	1013	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

4.4.2 OR Instruction

or Rx,Rx,Rx can be used to set PPR_{PRI} (see Section 4.3.5) as shown in Figure 16. For all priorities except medium high, PPR_{PRI} remains unchanged if the privilege state of the thread executing the instruction is lower than the privilege indicated in the figure. For priority medium high, PPR_{PRI} is set to medium if the thread executing the instruction is in problem state and medium high priority is not allowed for problem state programs. (The encodings available to problem-state programs, as well as encodings for additional shared resource hints not shown here, are described in Chapter 3 of Book II.)

Rx	PPR_{PRI}	Priority	Privileged
31	001	very low	no
1	010	low	no
6	011	medium low	no
2	100	medium	no
5	101	medium high	no/yes[1]
3	110	high	yes
7	111	very high	hypv

[1]This value is privileged unless the Problem State Priority Boost register allows the priority value 0b101 (See Section 4.3.6.)

Figure 16. Priority levels for *or Rx,Rx,Rx*

4.4.3 Transactional Memory Instructions [Category: Transactional Memory]

Privileged software that makes the Transactional Memory Facility available to applications takes on the responsibility of managing the facility's resources and the application's transactional state during interrupt handling, service calls, task switches, and its own use of TM. In addition to the existing instructions like ***rfid*** and problem state TM instructions that play a role in this management, ***treclaim*** and ***trechkpt.*** may be used, as described below. See Section 3.2.2 for additional information about managing the TM facility and associated state transitions.

Transaction Reclaim *X-form*

treclaim. RA

31	///	RA	///	942	1
0	6	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0

if MSR_TS = 0b10 | MSR_TS = 0b01 then
      #Transactional or Suspended
      if RA = 0 then cause <- 0x00000001
     else        cause <- GPR(RA)_56:63 || 0x000001
      if TEXASR_FS = 0 then
        Discard transactional footprint
      TMRecordFailure(cause)
      Revert checkpointed registers to pre-transac-
tional values
      Discard all resources related to current
transaction

MSR_TS <- 0b00                  #Non-transactional
```

The *treclaim.* instruction frees the transactional facility for use by a new transaction. It sets condition register field 0 to 0 || MSR_{TS} || 0. If the transactional facility is in the Transactional state or Suspended state, failure recording is performed as defined in Section 5.3.2 of Book II. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to $GPR(RA)_{56:63}$ || 0x000001. The checkpointed registers are reverted to their pre-transactional values, and all resources related to the current transaction are discarded, including the transactional footprint (if it wasn't already discarded for a pending failure).

The transaction state is set to Non-transactional.

If an attempt is made to execute ***treclaim.*** in Non-transactional state, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

Special Registers Altered

CR0TEXASR TFIAR TS

Programming Note

The *treclaim.* instruction can be used by an interrupt handler to deallocate the current thread's transactional resources in preparation for subsequent use of the facility by a new transaction. (An abort is not appropriate for this use, because (a) the interrupt handler is in Suspended state and an abort in Suspended state leaves the thread in Suspended state, and (b) an abort in Suspended state does not restore the checkpointed registers to their pre-transaction values.) After ***treclaim.*** is executed, when the interrupted program is next dispatched it should be resumed by first using ***trechkpt.*** to restore the pre-transactional register values into the checkpoint area. Failure handling for that program will occur when the program next attempts to execute an instruction in the Transactional state, which will cause the failure handler to be invoked because TDOOMED will be 1. (This will be immediate if the program was in the Transactional state when the interrupt occurred, or will be after ***tresume.*** is executed if the program was in the Suspended state when the interrupt occurred.)

Transaction Recheckpoint ***X-form***

trechkpt.

31	///	///	///	1006	1
0	6	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0

MSR_TS ← 0b01
TDOOMED ← 1
checkpoint area ←(checkpointed registers)
```

The ***trechkpt.*** instruction copies the current (pre-transactional, saved and restored by the operating system) register state to the checkpoint area. It sets condition register field 0 to 0 || MSR_{TS} || 0. The current values of the checkpointed registers are loaded into the checkpoint area. TDOOMED is set to 0b1.

The transaction state is set to Suspended.

If an attempt is made to execute this instruction in Transactional or Suspended state or when $TEXAS\text{-}R_{FS}$=0, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

Special Registers Altered

CR0 TS

4.4.4 Move To/From System Register Instructions

The *Move To Special Purpose Register* and *Move From Special Purpose Register* instructions are described in Book I, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in privileged state. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are implementation-dependent. In the descriptions of these instructions given below, the “defined” SPR numbers are the SPR numbers shown in the figure for the instruction and the implementation-specific SPR numbers that are implemented, and similarly for “defined” registers.

SPR numbers that are not shown in Figure 17 and are in the ranges shown below are reserved for implementation-specific uses.

848 - 863
880 - 895
976 - 991
1008 - 1023

Implementation-specific registers must be privileged. SPR numbers for implementation-specific SPRs should be registered in advance with the Power ISA architects.

Extended mnemonics

Extended mnemonics are provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. See Appendix A, “Assembler Extended Mnemonics” on page 1015.

Figure 17. SPR encodings (Sheet 1 of 3)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
1	00000 00001	XER	no	no	64	B
3	00000 00011	DSCR	no	no	64	STM
8	00000 01000	LR	no	no	64	B
9	00000 01001	CTR	no	no	64	B
13	00000 01101	AMR	no[5]	no	64	S
17	00000 10001	DSCR	yes	yes	64	STM
18	00000 10010	DSISR	yes	yes	32	S
19	00000 10011	DAR	yes	yes	64	S
22	00000 10110	DEC	yes	yes	32	B
25	00000 11001	SDR1	hypv[3]	hypv[3]	64	S
26	00000 11010	SRR0	yes	yes	64	B
27	00000 11011	SRR1	yes	yes	64	B
28	00000 11100	CFAR	yes	yes	64	S
29	00000 11101	AMR	yes[5]	yes	64	S
61	00001 11101	IAMR	yes[8]	yes	64	S
128	00100 00000	TFHAR	no	no	64	TM
129	00100 00001	TFIAR	no	no	64	TM
130	00100 00010	TEXASR	no	no	64	TM
131	00100 00011	TEXASRU	no	no	32	TM
136	00100 01000	CTRL	-	no	32	S
152	00100 11000	CTRL	yes	-	32	S
153	00100 11001	FSCR	yes	yes	64	S
157	00100 11101	UAMOR	yes[6]	yes	64	S
159	00100 11111	PSPB	yes	yes	32	S
176	00101 10000	DPDES	hypv[3]	yes	64	S.PC
177	00101 10001	DHDES	hypv[3]	hypv[3]	64	S.PC
180	00101 10100	DAWR0	hypv[3]	hypv[3]	64	S
186	00101 11010	RPR	hypv[3]	hypv[3]	64	S
187	00101 11011	CIABR	hypv[3]	hypv[3]	64	S
188	00101 11100	DAWRX0	hypv[3]	hypv[3]	32	S
190	00101 11110	HFSCR	hypv[3]	hypv[3]	64	S
256	01000 00000	VRSAVE	no	no	32	B
259	01000 00011	SPRG3	-	no	64	B
268	01000 01100	TB	-	no	64	B
269	01000 01101	TBU	-	no	32	B
272-275	01000 100xx	SPRG[0-3]	yes	yes	64	B
282	01000 11010	EAR	hypv[3]	hypv[3]	32	EC
283	01000 11011	CIR	-	yes	32	S
284	01000 11100	TBL	hypv[3]	-	32	B
285	01000 11101	TBU	hypv[3]	-	32	B
286	01000 11110	TBU40	hypv	-	64	S
287	01000 11111	PVR	-	yes	32	B
304	01001 10000	HSPRG0	hypv[3]	hypv[3]	64	S
305	01001 10001	HSPRG1	hypv[3]	hypv[3]	64	S
306	01001 10010	HDSISR	hypv[3]	hypv[3]	32	S
307	01001 10011	HDAR	hypv[3]	hypv[3]	64	S
308	01001 10100	SPURR	hypv[3]	yes	64	S
309	01001 10101	PURR	hypv[3]	yes	64	S
310	01001 10110	HDEC	hypv[3]	hypv[3]	32	S
312	01001 11000	RMOR	hypv[3]	hypv[3]	64	S
313	01001 11001	HRMOR	hypv[3]	hypv[3]	64	S
314	01001 11010	HSRR0	hypv[3]	hypv[3]	64	S
315	01001 11011	HSRR1	hypv[3]	hypv[3]	64	S

Figure 17. SPR encodings (Sheet 2 of 3)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
318	01001 11110	LPCR	hypv[3]	hypv[3]	64	S
319	01001 11111	LPIDR	hypv[3]	hypv[3]	32	S
336	01010 10000	HMER	hypv[3,4]	hypv[3]	64	S
337	01010 10001	HMEER	hypv[3]	hypv[3]	64	S
338	01010 10010	PCR	hypv[3]	hypv[3]	64	S
339	01010 10011	HEIR	hypv[3]	hypv[3]	32	S
349	01010 11101	AMOR	hypv[3]	hypv[3]	64	S
446	01101 11110	TIR	-	yes	64	S.PC
512	10000 00000	SPEFSCR	no	no	32	SP
526	10000 01110	ATB/ATBL	-	no	64	ATB
527	10000 01111	ATBU	-	no	32	ATB
768	11000 00000	SIER	-	yes	64	S
769	11000 00001	MMCR2	no[7]	no[7]	64	S
770	11000 00010	MMCRA	no[7]	no[7]	64	S
771	11000 00011	PMC1	no[7]	no[7]	32	S
772	11000 00100	PMC2	no[7]	no[7]	32	S
773	11000 00101	PMC3	no[7]	no[7]	32	S
774	11000 00110	PMC4	no[7]	no[7]	32	S
775	11000 00111	PMC5	no[7]	no[7]	32	S
776	11000 01000	PMC6	no[7]	no[7]	32	S
779	11000 01011	MMCR0	no[7]	no[7]	64	S
780	11000 01100	SIAR	-	no[7]	64	S
781	11000 01101	SDAR	-	no[7]	64	S
782	11000 01110	MMCR1	-	no[7]	64	S
784	11000 10000	SIER	yes	yes	64	S
785	11000 10001	MMCR2	yes	yes	64	S
786	11000 10010	MMCRA	yes	yes	64	S
787	11000 10011	PMC1	yes	yes	32	S
788	11000 10100	PMC2	yes	yes	32	S
789	11000 10101	PMC3	yes	yes	32	S
790	11000 10110	PMC4	yes	yes	32	S
791	11000 10111	PMC5	yes	yes	32	S
792	11000 11000	PMC6	yes	yes	32	S
795	11000 11011	MMCR0	yes	yes	64	S
796	11000 11100	SIAR	yes	yes	64	S
797	11000 11101	SDAR	yes	yes	64	S
798	11000 11110	MMCR1	yes	yes	64	S
800	11001 00000	BESCRS	no	no	64	S
801	11001 00001	BESCRSU	no	no	32	S
802	11001 00010	BESCRR	no	no	64	S
803	11001 00011	BESCRRU	no	no	32	S
804	11001 00100	EBBHR	no	no	64	S
805	11001 00101	EBBRR	no	no	64	S
806	11001 00110	BESCR	no	no	64	S
808	11001 01000	reserved[9]	no	no	na	B
809	11001 01001	reserved[9]	no	no	na	B
810	11001 01010	reserved[9]	no	no	na	B
811	11001 01011	reserved[9]	no	no	na	B
815	11001 01111	TAR	no	no	64	S

Figure 17. SPR encodings (Sheet 3 of 3)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
848	11010 10000	IC	hypv[3]	yes	64	S
849	11010 10001	VTB	hypv[3]	yes	64	S
896	11100 00000	PPR	no	no	64	S
898	11100 00010	PPR32	no	no	32	B
1023	11111 11111	PIR	-	yes	32	S

- This register is not defined for this instruction.

1 Note that the order of the two 5-bit halves of the SPR number is reversed.

2 See Section 1.3.5 of Book I.

3 This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2).

4 This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using *mtspr SPR,RS*.

5 The value specified in register RS may be masked by the contents of the [U]AMOR before being placed into the AMR; see the ***mtspr*** instruction description.

6 The value specified in register RS may be ANDed with the contents of the AMOR before being placed into the UAMOR; see the ***mtspr*** instruction description.

7 $MMCR0_{PMCC}$ controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 9.4.4 for details.

8 The value specified in Register RS may be masked by the contents of the AMOR before being placed into the IAMR; see the ***mtspr*** instruction description.

9 Accesses to these SPRs are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I.

SPR numbers 777-778, 783, 793-794, and 799 are reserved for the Performance Monitor. All other SPR numbers that are not shown above and are not implementation-specific are reserved.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(13): if MSRHV PR = 0b10 then
              SPR(13) ← (RS)
            else
              if MSRHV PR = 0b00 then
                SPR(13) ← ((RS) & AMOR) |
                          ((SPR(13)) & ¬AMOR)
              else
                SPR(13) ← ((RS) & UAM OR) |
                          ((SPR(13)) & ¬UAMOR)
  case(29,61):if MSRHV PR = 0b10 then
              SPR(n) ← (RS)
            else
               SPR(n) ← ((RS) & AMOR) |
                         ((SPR(n)) & ¬AMOR)
  case (157): if MSRHV PR = 0b10 then
                SPR(157) ← (RS)
              else
                SPR(157) ← (RS) & AMOR
  case (336):SPR(336) ← (SPR(336)) & (RS)
  case (808, 809, 810, 811):
  default: if length(SPR(n)) = 64 then
             SPR(n) ← (RS)
           else
             SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs" in Book I. Otherwise, the contents of register RS are placed into the designated Special Purpose Register, except as described in the next four paragraphs. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

When the designated SPR is the Authority Mask Register (AMR), (using SPR 13 or SPR 29), or the designated SPR is the Instruction Authority Mask Register (IAMR), and $MSR_{HV\ PR}$=0b00, the contents of bit positions of register RS corresponding to 1 bits in the Authority Mask Override Register (AMOR) are placed into the corresponding bits of the AMR or IAMR, respectively; the other AMR or IAMR bits are not modified.

When the designated SPR is the AMR, using SPR 13, and MSR_{PR}=1, the contents of bit positions of register RS corresponding to 1 bits in the User Authority Mask Override Register (UAMOR) are placed into the corresponding bits of the AMR; the other AMR bits are not modified.

When the designated SPR is the UAMOR and $MSR_{HV\ PR}$=0b00, the contents of register RS are ANDed with the contents of the AMOR and the result is placed into the UAMOR.

When the designated SPR is the Hypervisor Maintenance Exception Register (HMER), the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

spr_0=1 if and only if writing the register is privileged. Execution of this instruction specifying an SPR number with spr_0=1 causes a Privileged Instruction type Program interrupt when MSR_{PR}=1 and, if the SPR is a hypervisor resource (see Figure 17), when $MSR_{HV\ PR}$=0b00.

Execution of this instruction specifying an SPR number that is not defined for the implementation, including SPR numbers that are shown in Figure 17 but are in a category that is not supported by the implementation, causes one of the following.

- if spr_0=0:
  - if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
  - if MSR_{PR}=0: Hypervisor Emulation Assistance interrupt for SPR 0 and no operation (i.e. the instruction is treated as a no-op) for all other SPRs
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: no operation (i.e. the instruction is treated as a no-op)

If an attempt is made to execute ***mtspr*** specifying a TM SPR in other than Non-transactional state, a TM Bad Thing type Program interrupt is generated.

Special Registers Altered:

See Figure 17

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 12. "Synchronization Requirements for Context Alterations" on page 1009.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(129):
    if (MSRHV PR = 0b10)|(TFIARHV PR=MSRHV PR) |
     ((MSRHV PR = 0b00) & (TFIARHV PR= 0b01))then
       RT ← SPR(n)
    else
       RT ← 0
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      RT ← SPR(n)
    else
      RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the designated Special Purpose Register is the TFIAR and TFIAR indicates the failure was recorded in a state more privileged than the current state, register RT is set to zero.<tm> If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

Programming Note

Note that when a problem state transaction’s failure is recorded in hypervisor state and there is a subsequent need for a context switch in privileged, non-hypervisor state, an attempt to save TFIAR will result in zeros being saved. This is harmless because if the original application ever tries to read the TFIAR, it would read zeros anyway, since the failure took place in hypervisor state.

spr_0=1 if and only if reading the register is privileged. Execution of this instruction specifying an SPR number with spr_0=1 causes a Privileged Instruction type Program interrupt when MSR_{PR}=1 and, if the SPR is a hypervisor resource (see Figure 17), when $MSR_{HV\ PR}$=0b00.

Execution of this instruction specifying an SPR number that is not defined for the implementation, including SPR numbers that are shown in Figure 17 but are in a category that is not supported by the implementation, causes one of the following.

- if spr_0=0:
  - if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
  - if MSR_{PR}=0: Hypervisor Emulation Assistance interrupt for SPRs 0, 4, 5, and 6 and no operation (i.e. the instruction is treated as a no-op) for all other SPRs
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: no operation (i.e. the instruction is treated as a no-op)

Special Registers Altered:

None

Note

See the Notes that appear with ***mtspr***.

Move To Machine State Register X-form

mtmsr RS,L

31	RS	///	L	///	146	/
0	6	11	15	16	21	31

```
if L = 0 then
   MSR48 ← (RS)48 | (RS)49
   MSR58 ← (RS)58 | (RS)49
   MSR59 ← (RS)59 | (RS)49
   MSR32:47 49:50 52:57 60:62 ←(RS)32:47 49:50 52:57 60:62
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the ***mtmsr*** instruction description in this section, references to "***mtmsr***" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsr*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

This instruction does not alter MSR_{ME} or MSR_{LE}. (This instruction does not alter MSR_{HV} because it does not alter any of the high-order 32 bits of the MSR.)

If the only MSR bits to be altered are $MSR_{EE\ RI}$, to obtain the best performance L=1 should be used.

Programming Note

If MSR_{EE}=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an ***mtmsrd*** instruction that sets MSR_{EE} to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.8, "Interrupt Priorities" on page 970). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 12.

Programming Note

mtmsr serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mtmsr*** mnemonic with two operands as the basic form, and an ***mtmsr*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

There is no need for an analogous version of the ***mfmsr*** instruction, because the existing instruction copies the entire contents of the MSR to the selected GPR.

Move To Machine State Register Doubleword *X-form*

mtmsrd RS,L

31	RS	///	L	///	178	/
0	6	11	15	16	21	31

```
if L = 0 then

   if (MSR29:31 ¬= 0b010 | RS29:31 ¬= 0b000) then
      MSR29:31 ← RS29:31
   MSR48 ← (RS)48 | (RS)49
   MSR58 ← (RS)58 | (RS)49
   MSR59 ← (RS)59 | (RS)49
   MSR0:2 4:28 32:47 49:50 52:57 60:62
            ← (RS)0:2 4 6:28 32:47 49:50 52:57 60:62
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of RS are not equal to 0b000, then the value of bits 29 through 31 of RS is placed into bits 29 through 31 of the MSR.The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 0:2, 4:28, 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, "Transaction state transitions that can be requested by rfebb, rfid, hrfid, and mtmsrd.," on page 861), or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the ***mtmsrd*** instruction.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the ***mtmsrd*** instruction description in this section, references to "***mtmsrd***" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsrd*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

This instruction does not alter MSR_{LE}, MSR_{ME} or MSR_{HV}.

If the only MSR bits to be altered are $MSR_{EE\ RI}$, to obtain the best performance L=1 should be used.

Programming Note

If MSR_{EE}=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an ***mtmsrd*** instruction that sets MSR_{EE} to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.8, "Interrupt Priorities" on page 970). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 12.

Programming Note

mtmsrd serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mtmsrd*** mnemonic with two operands as the basic form, and an ***mtmsrd*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Move From Machine State Register *X-form*

mfmsr RT

31	RT	///	///	83	/
0	6	11	16	21	31

RT ← MSR

The contents of the MSR are placed into register RT.

This instruction is privileged.

Special Registers Altered:

None

Chapter 5. Storage Control

5.1 Overview

A program references storage using the effective address computed by the hardware when it executes a *Load*, *Store*, *Branch*, or *Cache Management* instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 5.7.3, in Section 5.7.5 and in the following sections. The real address is what is presented to the storage subsystem.

For a complete discussion of storage addressing and effective address calculation, see Section 1.10 of Book I.

5.2 Storage Exceptions

A *storage exception* results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Watchpoint).

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a Load or Store instruction. See Section 2.1 of Book II, and Section 6.6 in this Book.

5.3 Instruction Fetch

Instructions are fetched under control of MSR_{IR}.

MSR_{IR}=0

The effective address of the instruction is interpreted as described in Section 5.7.3.

MSR_{IR}=1

The effective address of the instruction is translated by the Address Translation mechanism described beginning in Section 5.7.5.

5.3.1 Implicit Branch

Explicitly altering certain MSR bits (using ***mtmsr***[***d***]), or explicitly altering SLB entries, Page Table Entries, or certain System Registers (including the HRMOR, and possibly other implementation-dependent registers), may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an *implicit branch*. For example, an ***mtmsrd*** instruction that changes the value of MSR_{SF} may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 12. "Synchronization Requirements for Context Alterations" on page 1009. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

5.3.2 Address Wrapping Combined with Changing MSR Bit SF

If the current instruction is at effective address $2^{32} - 4$ and is an ***mtmsrd*** instruction that changes the contents of MSR_{SF}, the effective address of the next sequential instruction is undefined.

Programming Note

In the case described in the preceding paragraph, if an interrupt occurs before the next sequential instruction is executed, the contents of SRR0, or HSRR0, as appropriate to the interrupt, are undefined.

5.4 Data Access

Data accesses are controlled by MSR_{DR}.

MSR_{DR}=0

The effective address of the data is interpreted as described in Section 5.7.3.

MSR_{DR}=1

The effective address of the data is translated by the Address Translation mechanism described in Section 5.7.5.

5.5 Performing Operations Out-of-Order

An operation is said to be performed "in-order" if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed "out-of-order" if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as *Branch*, *Trap*, *System Call*, and *Return From Interrupt* instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, operations are performed out-of-order when resources are available that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, any results of the operation are abandoned (except as described below).

In the remainder of this section, including its subsections, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

A data access that is performed out-of-order may correspond to an arbitrary *Load* or *Store* instruction (e.g., a *Load* or *Store* instruction that is not in the instruction stream being executed). Similarly, an instruction fetch that is performed out-of-order may be for an arbitrary instruction (e.g., the aligned word at an arbitrary location in instruction storage).

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

- Stores
  Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order).
- Accessing Guarded Storage
  The restrictions for this case are given in Section 5.8.1.1.

The only permitted side effects of performing an operation out-of-order are the following.

- A Machine Check or Checkstop that could be caused by in-order execution may occur out-of-order, except as described in Section 5.7.3.3.1 for the Hypervisor Real Mode Storage Control facility.
- On implementations which support Reference and Change bits, these bits may be set as described in Section 5.7.8.
- Non-Guarded storage locations that could be fetched into a cache by in-order fetching or execution of an arbitrary instruction may be fetched out-of-order into that cache.

5.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 5.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist.

In the case that the accessed storage location does not exist, the Checkstop state may be entered. See Section 6.5.2 on page 953.

Programming Note

In configurations supporting multiple partitions, hypervisor software must ensure that a storage access by a program in one partition will not cause a Checkstop or other system-wide event that could affect the integrity of other partitions (see Chapter 2). For example, such an event could occur if a real address placed in a Page Table Entry or made accessible to a partition using the Offset Real Mode Address mechanism (see Section 5.7.3.2) does not exist.

5.7 Storage Addressing

Storage Control Overview

- Real address space size is 2^m bytes, m≤60; see Note 1.
- Real page size is 2^{12} bytes (4 KB).
- Effective address space size is 2^{64} bytes.
- An effective address is translated to a virtual address via the Segment Lookaside Buffer (SLB).
  - Virtual address space size is 2^n bytes, 65≤n≤78; see Note 2.
  - Segment size is 2^s bytes, s=28 or 40.
  - 2^{n-40} ≤ number of virtual segments ≤ 2^{n-28}; see Note 2.
  - Virtual page size is 2^p bytes, where 12≤p, and 2^p is no larger than either the size of the biggest segment or the real address space; a size of 4 KB, 64 KB, and an implementation-dependent number of other sizes are supported; see Note 3. The Page Table specifies the virtual page size. The SLB specifies the base virtual page size, which is the smallest virtual page size that the segment can contain. The base virtual page size is 2^b bytes.
  - Segments contain pages of a single size, a mixture of 4 KB and 64 KB pages, or a mixture of page sizes that include implementation-dependent page sizes.
- A virtual address is translated to a real address via the Page Table.

Notes:

1. The value of m is implementation-dependent (subject to the maximum given above). When used to address storage, the high-order 60-m bits of the "60-bit" real address must be zeros.
2. The value of n is implementation-dependent (subject to the range given above). In references to 78-bit virtual addresses elsewhere in this Book, the high-order 78-n bits of the "78-bit" virtual address are assumed to be zeros.
3. The supported values of p for the larger virtual page sizes are implementation-dependent (subject to the limitations given above).

5.7.1 32-Bit Mode

The computation of the 64-bit effective address is independent of whether the thread is in 32-bit mode or 64-bit mode. In 32-bit mode (MSR_{SF}=0), the high-order 32 bits of the 64-bit effective address are treated as zeros for the purpose of addressing storage. This applies to both data accesses and instruction fetches. It applies independent of whether address translation is enabled or disabled. This truncation of the effective address is the only respect in which storage accesses in 32-bit mode differ from those in 64-bit mode.

Programming Note

Treating the high-order 32 bits of the effective address as zeros effectively truncates the 64-bit effective address to a 32-bit effective address such as would have been generated on a 32-bit implementation of the Power ISA. Thus, for example, the ESID in 32-bit mode is the high-order four bits of this truncated effective address; the ESID thus lies in the range 0-15. When address translation is enabled, these four bits would select a Segment Register on a 32-bit implementation of the Power ISA. The SLB entries that translate these 16 ESIDs can be used to emulate these Segment Registers.

5.7.2 Virtualized Partition Memory (VPM) Mode

VPM mode enables the hypervisor to reassign all or part of a partition's memory transparently so that the reassignment is not visible to the partition. When this is done, the partition's memory is said to be "virtualized." The VPM field in the LPCR enables VPM mode separately when address translation is enabled and when translation is disabled.

If the thread is not in hypervisor state, and either address translation is enabled and VPM_1=1, or address translation is disabled and VPM_0=1, conditions that would have caused a Data Storage or an Instruction Storage interrupt if the affected memory were not virtualized instead cause a Hypervisor Data Storage or a Hypervisor Instruction Storage interrupt respectively. Because the Hypervisor Data Storage and Hypervisor Instruction Storage interrupts always put the thread in hypervisor state, they permit the hypervisor to handle the condition if appropriate (e.g., to restore the contents of a page that was reassigned), and to reflect it to the operating system's Data Storage or Instruction Storage interrupt handler otherwise.

When address translation is enabled, VPM mode has no effect on address translation. When address translation is disabled, addressing is controlled as specified in Section 5.7.3.

5.7.3 Real And Virtual Real Addressing Modes

When a storage access is an instruction fetch performed when instruction address translation is disabled, or if the access is a data access and data address translation is disabled, it is said to be per-

formed in "real addressing mode" if VPM_0=0 and the thread is not in hypervisor state. If the thread is in hypervisor state, the access is said to be performed in "hypervisor real addressing mode" regardless of the value of VPM_0. If the thread is not in hypervisor state and VPM_0=1, the access is said to be performed in "virtual real addressing mode." Storage accesses in real, hypervisor real, and virtual real addressing modes are performed in a manner that depends on the contents of MSR_{HV}, VPM, VRMASD, HRMOR, RMLS, RMOR (see Chapter 2), bit 0 of the effective address (EA0), and the state of the Real Mode Storage Control Facility as described below. Bits 1:3 of the effective address are ignored.

MSR_{HV}=1

- If EA_0=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 5.7.3.1, controls the access.
- If EA_0=1, bits 4:63 of the effective address are used as the real address for the access.

MSR_{HV}=0

- If VPM_0=0, the Offset Real Mode Address mechanism, described in Section 5.7.3.2, controls the access.
- If VPM_0=1, the Virtual Real Mode Addressing mechanism, described in Section 5.7.3.4, controls the access.

5.7.3.1 Hypervisor Offset Real Mode Address

If MSR_{HV} = 1 and EA_0 = 0, the access is controlled by the contents of the Hypervisor Real Mode Offset Register, as follows.

Hypervisor Real Mode Offset Register (HRMOR)

Bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the HRMOR, and the 60-bit result is used as the real address for the access. The supported offset values are all values of the form $i \times 2^r$, where $0 \le i < 2^j$, and j and r are implementation-dependent values having the properties that $12 \le r \le 26$ (i.e., the minimum offset granularity is 4 KB and the maximum offset granularity is 64 MB) and j+r = m, where the real address size supported by the implementation is m bits.

Programming Note

$EA_{4:63-r}$ should equal $^{60-r}0$. If this condition is satisfied, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset.

If m<60, $EA_{4:63-m}$ and $HRMOR_{4:63-m}$ must be zeros.

5.7.3.2 Offset Real Mode Address

If VPM_0=0 and MSR_{HV}=0, the access is controlled by the contents of the Real Mode Limit Selector and Real Mode Offset Register, as specified below, and the set of storage locations accessible by code is referred to as the Real Mode Area (RMA).

Real Mode Limit Selector (RMLS)

If bits 4:63 of the effective address for the access are greater than or equal to the value (limit) represented by the contents of the $LPCR_{RMLS}$, the access causes a storage exception (see Section 5.7.9.3). In this comparison, if m<60, bits 4:63-m of the effective address may be ignored (i.e., treated as if they were zeros), where the real address size supported by the implementation is m bits. The supported limit values are of the form 2^j, where $12 \le j \le 60$. Subject to the preceding sentence, the number and values of the limits supported are implementation-dependent.

Real Mode Offset Register (RMOR)

If the access is permitted by the $LPCR_{RMLS}$, bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the RMOR, and the low-order m bits of the 60-bit result are used as the real address for the access. The supported offset values are all values of the form $i \times 2^s$, where $0 \le i < 2^k$, and k and s are implementation-dependent values having the properties that 2^s is the minimum limit value supported by the implementation (i.e., the minimum value representable by the contents of the $LPCR_{RMLS}$) and k+s = m.

Programming Note

The offset specified by the RMOR should be a nonzero multiple of the limit specified by the RMLS. If these registers are set thus, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset. (The offset must not be zero, because real page 0 contains the fixed interrupt vectors and real pages 1 and 2 may be used for implementation-specific purposes; see Section 5.7.4, "Address Ranges Having Defined Uses" on page 895.)

5.7.3.3 Storage Control Attributes for Accesses in Real and Hypervisor Real Addressing Modes

Storage accesses in hypervisor real addressing mode are performed as though all of storage had the following storage control attributes, except as modified by the Hypervisor Real Mode Storage Control facility (see Section 5.7.3.3.1). (The storage control attributes are defined in Book II.)

- not Write Through Required
- not Caching Inhibited, for instruction fetches
- not Caching Inhibited, for data accesses except those caused by the *Load/Store Caching Inhibited* instructions; Caching Inhibited, for data accesses caused by the *Load/Store Caching Inhibited* instructions
- Memory Coherence Required, for data accesses
- Guarded
- not SAO

Storage accesses in real addressing mode are performed as though all of storage had the following storage control attributes. (Such accesses use the Offset Real Mode Address mechanism.)

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required, for data accesses
- not Guarded
- not SAO

Additionally, storage accesses in real or hypervisor real addressing modes are performed as though all storage was not No-execute.

Programming Note

Because storage accesses in real addressing mode and hypervisor real addressing mode do not use the SLB or the Page Table, accesses in these modes bypass all checking and recording of information contained therein (e.g., storage protection checks that use information contained therein are not performed, and reference and change information is not recorded).

5.7.3.3.1 Hypervisor Real Mode Storage Control

The Hypervisor Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as non-Guarded in hypervisor real addressing mode ($MSR_{HV\ PR}$=0b10, and MSR_{IR}=0 or MSR_{DR}=0, as appropriate for the type of access). The remaining portions are treated as Guarded in hypervisor real addressing mode. The means is a hypervisor resource (see Chapter 2), and may also be system-specific.

Implementations may use either, or both, of two techniques to specify portions of real storage that are treated as non-Guarded in hypervisor real addressing mode. For the first technique, the facility provides for the specification, at coarse granularity, of the boundary between non-Guarded and Guarded real storage. Any storage location below the specified boundary is treated as non-Guarded in hypervisor real addressing mode, and any storage location at or above the boundary is treated as Guarded in hypervisor real addressing mode. For the second technique, the facility divides real storage into history blocks, in implementation-specific sizes. The history for instruction fetches is tracked separately from that for data accesses. If there is no instruction fetch history for a block and it is the target of an instruction fetch, the access is performed as though the block is Guarded, but the block is treated as non-Guarded for subsequent instruction fetches on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using a *Load/Store Caching Inhibited* instruction, the access is performed as though the block is Guarded, and the block is treated as Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using any other *Load* or *Store* instruction, the access is performed as though the block is Guarded, but the block is treated as non-Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain.

The storage location specified by a *Load/Store Caching Inhibited* instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. If the second technique is used, the storage location specified by any other *Load* or *Store* instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as Guarded. (For the second technique, "specified by the Hypervisor Real Mode Storage Control facility" means "specified in a history block".) For the second technique, the history can be erased using an slbia instruction; see See Section 5.9.3.1.

Programming Note

There are two cautions about mixing different types of accesses (i.e.*Load/Store Caching Inhibited* instructions vs. any other *Load* or *Store* instruction vs. instruction fetches). The first, as indicated above, is to avoid confusing the history mechanism, and the granularity for concern is a history block. For this caution, instruction fetches are irrelevant because they have their own history mechanism and are always intended to be non-guarded.

The second caution is to avoid storage paradoxes that result from a caching-inhibited access to a location that is held in a cache. The nature of this caution and its solution are described in Section 5.8.2.2, “Altering the Storage Control Bits”. The minimum granularity for concern is the history block, but may be larger, depending on extant translations to the storage in question. Since the consistency of instruction storage is managed by software and hypervisor real mode instruction fetches are always not Caching Inhibited, instruction fetches are also irrelevant to this caution.

The facility does not apply to implicit accesses to the Page Table performed during address translation or in recording reference and change information. These accesses are performed as described in Section 5.7.3.5.

Programming Note

The preceding capability can be used to improve the performance of hypervisor software that runs in hypervisor real addressing mode, by causing accesses to instructions and data that occupy well-behaved storage to be treated as non-Guarded.

5.7.3.4 Virtual Real Mode Addressing Mechanism

If VPM_0=1, MSR_{HV}=0, and MSR_{DR}=0 or MSR_{IR}=0 as appropriate for the type of access, the access is said to be made in virtual real addressing mode and is controlled by the mechanism specified below. The set of storage locations accessible by code is referred to as the Virtualized Real Mode Area (VRMA).

In virtual real addressing mode, address translation, storage protection, and reference and change recording are handled as follows.

- Address translation and storage protection are handled as if address translation were enabled, except that translation of effective addresses to virtual addresses use the SLBE values in Figure 18 instead of the entry in the SLB corresponding to the ESID. In this translation, bits 0:23 of the effective address are ignored (i.e. treated as if they were 0s), bits 24:63-m may be ignored if m < 40, and the Virtual Page Class Key Protection mechanism does not apply.

  Programming Note

  The Virtual Page Class Key Protection mechanism does not apply because the authority mask that an OS has set for application programs executing with address translation enabled may not be the same as the authority mask required by the OS when address translation is disabled, such as when first entering an interrupt handler.

- Reference and change recording are handled as if address translation were enabled.

Field	Value
ESID	$^{36}0$
V	1
B	0b01 - 1 TB
VSID	0b00 || 0x0_01FF_FFFF
K_s	0
K_p	undefined
N	0
L	$VRMASD_L$
C	0
LP	$VRMASD_{LP}$

Figure 18. SLBE for VRMA

Programming Note

The C bit in Figure 18 is set to 0 because the implementation-dependent lookaside information associated with the VRMA is expected to be long-lived. See Section 5.9.3.1.

Programming Note

The 1 TB VSID 0x0_01FF_FFFF should not be used by the operating system for purposes other than mapping the VRMA when address translation is enabled.

Programming Note

Software should specify PTE_B = 0b01 for all Page Table Entries that map the VRMA in order to be consistent with the values in Figure 18.

Programming Note

All accesses to the RMA are considered not Guarded. The G bit of the associated Page Table Entry determines whether an access to the VRMA is Guarded. Therefore, if an instruction is fetched from the VRMA, a Hypervisor Instruction Storage interrupt will result if G=1 in the associated Page Table Entry.

Programming Note

The RMA is considered non-SAO storage. However, any page in the VRMA is treated as SAO storage if WIMG = 0b1110 in the associated Page Table Entry.

5.7.3.5 Storage Control Attributes for Implicit Storage Accesses

Implicit accesses to the Page Table during address translation and in recording reference and change information are performed as though the storage occupied by the Page Table had the following storage control attributes.

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required
- not Guarded
- not SAO

The definition of "performed" given in Book II applies also to these implicit accesses; accesses for performing address translation are considered to be loads in this respect, and accesses for recording reference and change information are considered to be stores. These implicit accesses are ordered by the ***ptesync*** instruction as described in Section 5.9.2.

5.7.4 Address Ranges Having Defined Uses

The address ranges described below have uses that are defined by the architecture.

- Fixed interrupt vectors

  Except for the first 256 bytes, which are reserved for software use, the real page beginning at real address 0x0000_0000_0000_0000 is either used for interrupt vectors or reserved for future interrupt vectors.

- Implementation-specific use

  The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes.

- Offset Real Mode interrupt vectors

  The real pages beginning at the real address specified by the HRMOR and RMOR are used similarly to the page for the fixed interrupt vectors.

- Relocated interrupt vectors

  Depending on the values of $MSR_{IR\ DR}$ and $LPCR_{AIL}$ and on whether the specific interrupt will cause MSR_{HV} to change, either the virtual page containing the byte addressed by effective address 0x0000_0000_0001_8000 or the virtual page containing the byte addressed by effective address 0xc000_0000_0000_4000 may be used similarly to the page for the fixed interrupt vectors. (See Section 2.2.)

- Page Table

  A contiguous sequence of real pages beginning at the real address specified by SDR1 contains the Page Table.

5.7.5 Address Translation Overview

The effective address (EA) is the address generated by the hardware for an instruction fetch or for a data access. If address translation is enabled, this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage.

The first step in address translation is to convert the effective address to a virtual address (VA), as described in Section 5.7.6. The second step, conversion of the virtual address to a real address (RA), is described in Section 5.7.7.

If the effective address cannot be translated, a storage exception (see Section 5.2) occurs.

Figure 19 gives an overview of the address translation process.

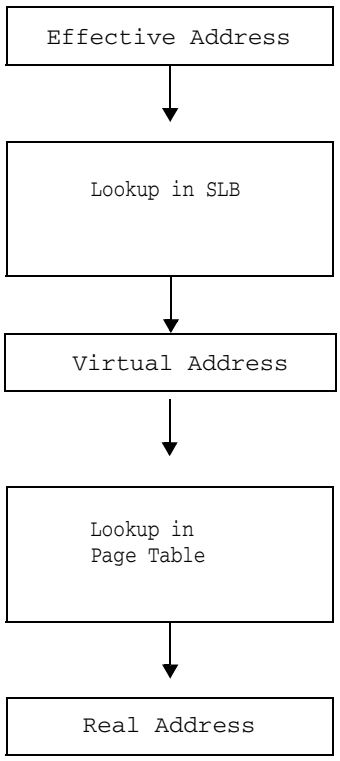


Figure 19. Address translation overview

5.7.6 Virtual Address Generation

Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 20.

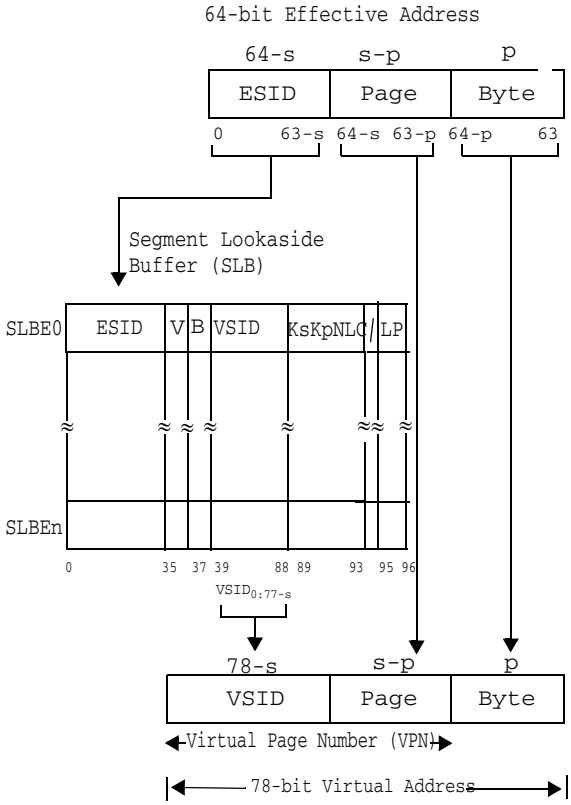


Figure 20. Translation of 64-bit effective address to 78 bit virtual address

5.7.6.1 Segment Lookaside Buffer (SLB)

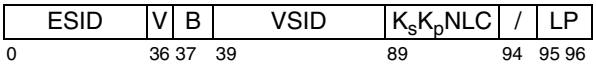
The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The contents of the SLB are managed by software, using the instructions described in Section 5.9.3.1. See Chapter 12. “Synchronization Requirements for Context Alterations” on page 1009 for the rules that software must follow when updating the SLB.

SLB Entry

Each SLB entry (SLBE, sometimes referred to as a “segment descriptor”) maps one ESID to one VSID. Figure 21 shows the layout of an SLB entry

.



Bit(s)	Name	Description
0:35	ESID	Effective Segment ID
36	V	Entry valid (V=1) or invalid (V=0)
37:38	B	Segment Size Selector 0b00 - 256 MB (s=28) 0b01 - 1 TB (s=40) 0b10 - reserved 0b11 - reserved
39:88	VSID	Virtual Segment ID
89	K_s	Supervisor (privileged) state storage key (see Section 5.7.9.2)
90	K_p	Problem state storage key (See Section 5.7.9.2.)
91	N	No-execute segment if N=1
92	L	Virtual page size selector bit 0
93	C	Class
95:96	LP	Virtual page size selector bits 1:2

All other fields are reserved. B_0 ($SLBE_{37}$) is treated as a reserved field.

Figure 21. SLB Entry

Instructions cannot be executed from a No-execute (N=1) segment.

Segments may contain a mixture of pages sizes. The L and LP bits specify the base virtual page size that the segment may contain. The $SLB_{L||LP}$ encodings are those shown in Figure 22. The base virtual page size (also referred to as the "base page size") is the smallest virtual page size for the segment. The base virtual page size is 2^b bytes. The actual virtual page size (also referred to as the "actual page size" or "virtual page size") is specified by $PTE_{L\ LP}$

encoding	base page size
0b000	4 KB
0b101	64 KB
additional values[1]	2^b bytes, where b > 12 and b may differ among encoding values

[1] The "additional values" are implementation-dependent, as are the corresponding base virtual page sizes. Any values that are not supported by a given implementation are reserved in that implementation.

Figure 22. Page Size Encodings

For each SLB entry, software must ensure the following requirements are satisfied.

- L||LP contains a value supported by the implementation.
- The base virtual page size selected by the L and LP fields does not exceed the segment size selected by the B field.
- If s=40, the following bits of the SLB entry contain 0s.
  - $ESID_{24:35}$
  - $VSID_{38:49}$

  The bits in the above two items are ignored by the hardware.

The Class field of the SLBE is used in conjunction with the ***slbie*** and ***slbia*** instructions (see Section 5.9.3.1). "Class" refers to a grouping of SLB entries and implementation-specific lookaside information so that only entries in a certain group need be invalidated and others might be preserved. The Class value assigned to an implementation-specific lookaside entry derived from an SLB entry must match the Class value of that SLB entry. The Class value assigned to an implementation-specific lookaside entry that is not derived from an SLB entry (such as real mode address "translations") is 0.

Software must ensure that the SLB contains at most one entry that translates a given effective address, and that if the SLB contains an entry that translates a given effective address, then any previously existing translation of that effective address has been invalidated. An attempt to create an SLB entry that violates this requirement may cause a Machine Check.

Programming Note

It is permissible for software to replace the contents of a valid SLB entry without invalidating the translation specified by that entry provided the specified restrictions are followed. See Chapter 12 Note 11.

5.7.6.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following conditions must be satisfied for indicated fields in the SLBE.

- V=1
- $ESID_{0:63-s}$=$EA_{0:63-s}$, where the value of s is specified by the B field in the SLBE being tested

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the SLB search succeeds, the virtual address (VA) is formed from the EA and the matching SLB entry fields as follows.

$VA=VSID_{0:77-s} \,||\, EA_{64-s:63}$

The Virtual Page Number (VPN) is bits 0:77-p of the virtual address. The value of p is the actual virtual page size specified by the PTE used to translate the virtual address (see Section 5.7.7.1). If $SLBE_N$ = 1, the N (No-execute) value used for the storage access is 1.

If the SLB search fails, a *segment fault* occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

5.7.7 Virtual to Real Translation

Conversion of a 78-bit virtual address to a real address is done by searching the Page Table as shown in Figure 23.

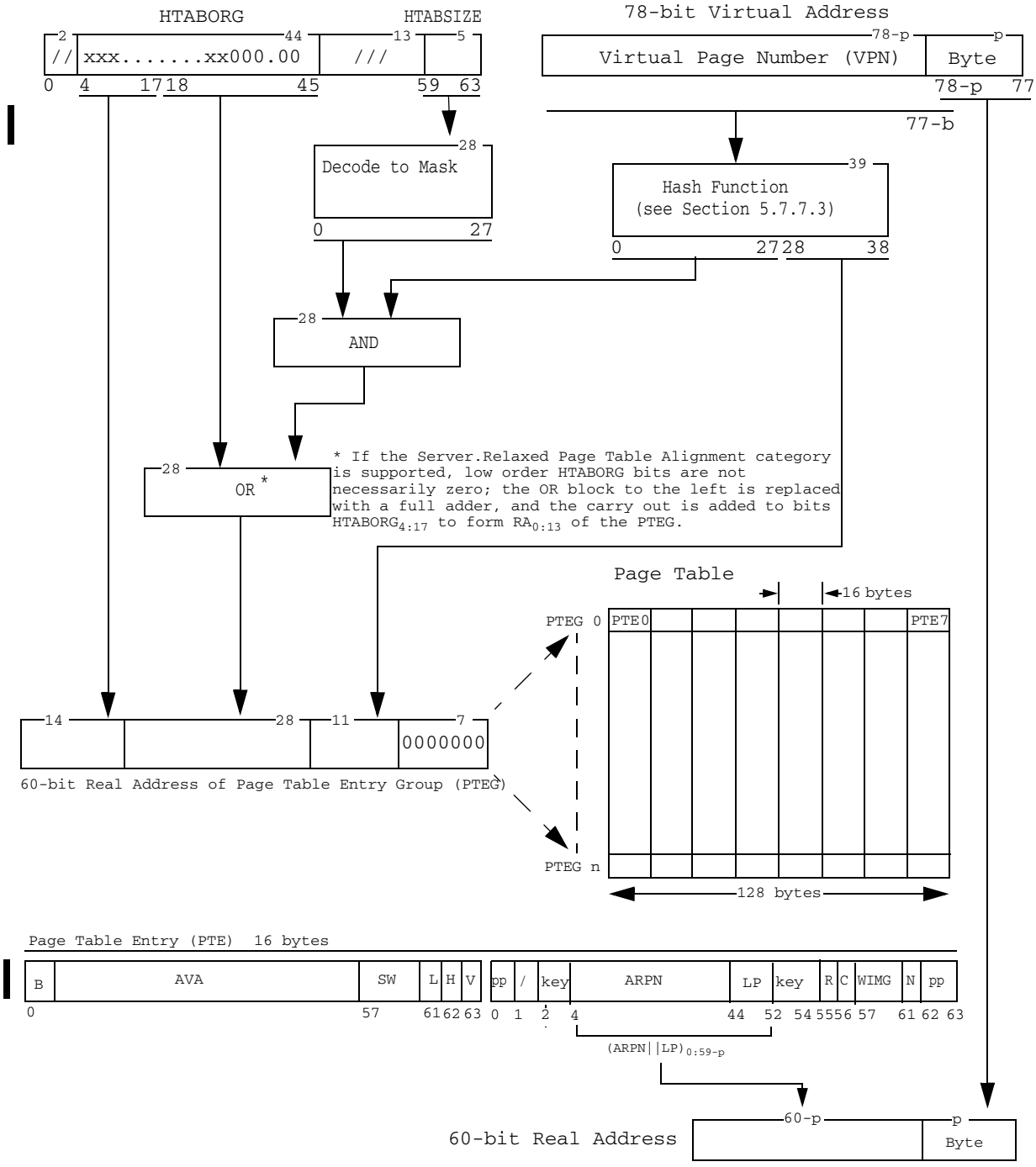


Figure 23. Translation of 78-bit virtual address to 60-bit real address

5.7.7.1 Page Table

The Hashed Page Table (HTAB) is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers, where the real page number of a real page is bits 0:47 of the address of the first byte in the real page. The HTAB's size can be any size 2^n bytes where $18 \le n \le 46$. The HTAB must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 5.7.3.5). The starting address must be a multiple of its size unless the implementation supports the Server.Relaxed Page Table Alignment category, in which case its starting address is a multiple of 2^{18} bytes (see Section 5.7.7.4).

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

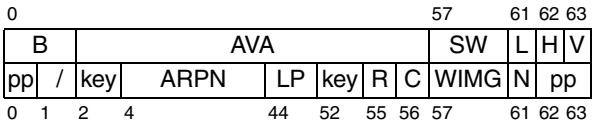
See Section 5.10 for the rules that software must follow when updating the Page Table.

Programming Note

The Page Table must be treated as a hypervisor resource (see Chapter 2), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data.

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 24 shows the layout of a PTE. This layout is independent of the Endian mode of the thread.



Dword	Bit(s)	Name	Description
0	0:1	B	Segment Size 0b00 - 256 MB 0b01 - 1 TB 0b10 - reserved 0b11 - reserved
	2:56	AVA	Abbreviated Virtual Address
	57:60	SW	Available for software use
	61	L	Virtual page size 0b0 - 4 KB 0b1 - greater than 4KB (large page)
	62	H	Hash function identifier
	63	V	Entry valid (V=1) or invalid (V=0)
1	0	pp	Page Protection bit 0
	2:3	key	KEY bits 0:1
	4:43	ARPN	Abbreviated Real Page Number
	44:51	LP	Large page size selector
	52:54	key	KEY bits 2:4
	55	R	Reference bit
	56	C	Change bit
	57:60	WIMG	Storage control bits
	61	N	No-execute page if N=1
	62:63	pp	Page Protection bits 1:2

All other fields are reserved.

Figure 24. Page Table Entry

Programming Note

The H bit in the Page Table entry should not be set to one unless the secondary Page Table search has been enabled.

If $b \le 23$, the Abbreviated Virtual Address (AVA) field contains bits 0:54 of the VA. Otherwise bits 0:77-b of the AVA field contain bits 0:77-b of the VA, and bits 78-b:54 of the AVA field must be zero.

Programming Note

The AVA field omits the low-order 23 bits of the VA. These bits are not needed in the PTE, because the low-order b of these bits are part of the byte offset into the virtual page and, if b<23, the high-order 23-b of these bits are always used in selecting the PTEGs to be searched (see Section 5.7.7.3).

On implementations that support a virtual address size of only n bits, n<78, bits 0:77-n of the AVA field must be zeros.

A virtual page is mapped to a sequence of 2^{p-12} contiguous real pages such that the low-order p-12 bits of the real page number of the first real page in the sequence are 0s.

$PTE_{L\ LP}$ specify both a base virtual page size (henceforth referred to as the "base page size") and an actual virtual page size (henceforth referred to as the "actual page size" or "virtual page size"). The actual page size is the size of the virtual page mapped by the PTE. The base page size is the smallest actual page size that a segment can contain. See Section 5.7.6.

If PTE_L=0, the base virtual page size and actual virtual page size are 4KB, and ARPN concatenated with LP (ARPN||LP) contains the page number of the real page that maps the virtual page described by the entry.

If PTE_L=1, the base page size and actual page size are specified by PTE_{LP}. In this case, the contents of PTE_{LP} have the format shown in Figure 25. Bits labelled "r" are

bits of the real page number. Bits labelled “z” specify the base page size and actual page size. The values of the “z” bits used to specify each size are implementation-dependent. The values of the “z” bits used to specify each size, along with all possible values of “r” bits in the LP field, must result in LP values distinct from other LP values for other sizes. Actual page sizes 4KB and 64KB are always supported; other actual page sizes are implementation-dependent. If PTE_L=1, the actual page size must be greater than 4 KB. Which combinations of different base page size and actual page size are supported is implementation-dependent, except that the combination of a base page size of 4 KB with an actual page size of 64 KB is always supported.

PTE_{LP}	actual page size
r r r r _r r r z	≥8 KB
r r r r _r r z z	≥16 KB
r r r r _r z z z	≥32 KB
r r r r _z z z z	≥64 KB
r r r z _z z z z	≥128 KB
r r z z _z z z z	≥256 KB
r z z z _z z z z	≥512 KB
z z z z _z z z z	≥1 MB

Figure 25. Format of PTE_{LP} when PTE_L=1

There are at least 2 formats of PTE_{LP} that specify a 64 KB page. One format is used with $SLBE_{L||LP}$ = 0b000 and one format is used with $SLBE_{L||LP}$ = 0b101.

The actual page size selected by the LP field must not exceed the segment size selected by the B field. Forms of PTE_{LP} not supported by a given implementation are treated as reserved values for that implementation.

The concatenation of the ARPN field and bits labeled “r” in the LP field contain the high-order bits of the real page number of the real page that maps the first 4KB of the virtual page described by the entry.

The low-order p-12 bits of the real page number contained in the ARPN and LP fields must be 0s and are ignored by the hardware.

Programming Note

The actual page size specified by a given PTE_{LP} format is at least $2^{12+(8-c)}$, where c is the number of r bits in the format.

Programming Note

Implementations often have implementation-dependent lookaside buffers (e.g. TLBs and ERATs) used to cache translations of recently used storage addresses. Mapping virtual storage to large pages may increase the effectiveness of such lookaside buffers, improving performance, because it is possible for such buffers to translate a larger range of addresses, reducing the frequency that the Page Table must be searched to translate an address.

Instructions cannot be executed from a No-execute (N=1) page.

Page Table Size

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs (when the secondary Page Table search is enabled) or more than 8 entries for the same primary PTEG (when the secondary Page Table search is disabled).

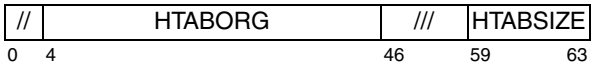
While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 5.7.7.2) will reduce the frequency of occurrence of such collisions.

Programming Note

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is 2^{31} bytes (2 GB), then we have $2^{31-12}=2^{19}$ real pages. The minimum recommended Page Table size would be 2^{18} PTEGs, or 2^{25} bytes (32 MB).

5.7.7.2 Storage Description Register 1

The Storage Description Register 1 (SDR1) register is shown in Figure 26.



Bits	Name	Description
4:45	HTABORG	Real address of Page Table
59:63	HTABSIZE	Encoded size of Page Table

All other fields are reserved.

Figure 26. SDR1

SDR1 is a hypervisor resource; see Chapter 2.

The HTABORG field in SDR1 contains the high-order 42 bits of the 60-bit real address of the Page Table. The Page Table is thus constrained to lie on a 2^{18} byte (256 KB) boundary at a minimum. At least 11 bits from the hash function (see Figure 23) are used to index into the Page Table. The minimum size Page Table is 256 KB (2^{11} PTEGs of 128 bytes each).

The Page Table can be any size 2^n bytes where $18 \le n \le 46$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0 unless the implementation supports the Server.Relaxed Page Table Alignment category; see Section 5.7.7.4.

The HTABSIZE field in SDR1 contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the Page Table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form 0b00...011...1, which is a string of 28 - HTABSIZE 0-bits followed by a string of HTABSIZE 1-bits. The 1-bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index (see Figure 23). The number of low-order 0 bits in HTABORG must be greater than or equal to the value in HTABSIZE.

On implementations that support a real address size of only m bits, m<60, bits 0:59-m of the HTABORG field are treated as reserved bits, and software must set them to zeros.

Programming Note

Let n equal the virtual address size (in bits) supported by the implementation. If n<67, software should set the HTABSIZE field to a value that does not exceed n-39. Because the high-order 78-n bits of the VSID are assumed to be zeros, the hash value used in the Page Table search will have the high-order 67-n bits either all 0s (primary hash; see Section 5.7.7.3) or all 1s (secondary hash). If HTABSIZE > n-39, some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.

Example:

Suppose that the Page Table is 16,384 (2^{14}) 128-byte PTEGs, for a total size of 2^{21} bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3 and the value in HTABORG must have its low-order 3 bits (bits 43:45 of SDR1) equal to 0. This means that the Page Table must begin on a $2^{3+11+7} = 2^{21}$ = 2 MB boundary.

5.7.7.3 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 5.7.3.5.

An outline of the HTAB search process is shown in Figure 23. If the implementation supports the Server.Relaxed Page Table Alignment category see Section 5.7.7.4. Up to two hash functions are used to locate a PTE that may translate the given virtual address.

1. A 39-bit hash value is computed from the VA. The value of s is the value specified in the SLBE that was used to generate the virtual address; the value of b is equal to $\log_2$(base page size specified in the SLBE that was used to translate the address).**Primary Hash:**

   If s=28, the hash value is computed by Exclusive ORing $VA_{11:49}$ with $(^{11+b}0 || VA_{50:77-b})$

   If s=40, the hash value is computed by Exclusive ORing the following three quantities: $(VA_{24:37} || ^{25}0)$, $(0 || VA_{0:37})$, and $(^{b-1}0 || VA_{38:77-b})$

   The 60-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 4:17 of SDR1 (the high-order 14 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies a particular PTEG, called the “primary PTEG”, whose eight PTEs will be tested.

2. **Secondary Hash:**

   If the secondary Page Table search is enabled ($LPCR_{TC}$=0), perform the secondary hash function as follows; otherwise do not perform step 2 and proceed to step 3 below.

   If s=28, the hash value is computed by taking the ones complement of the Exclusive OR of $VA_{11:49}$ with $(^{11+b}0 || VA_{50:77-b})$

   If s=40, the hash value is computed by taking the ones complement of the Exclusive OR of the following three quantities: $(VA_{24:37} || ^{25}0)$, $(0 || VA_{0:37})$, and $(^{b-1}0 || VA_{38:77-b})$

   The 60-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 4:17 of SDR1 (the high-order 14 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies the “secondary PTEG”.

3. As many as 8 PTEs in the primary PTEG and, if the secondary Page Table search is enabled, 8 PTEs in the secondary PTEG are tested to determine if any translate the given virtual address. Let q = minimum(54, 77-b). For a match to exist, the

following conditions must be satisfied, where SLBE is the SLBE used to form the virtual address.

- PTE_H=0 for the primary PTEG, 1 for the secondary PTEG
- PTE_V=1
- PTE_B=$SLBE_B$
- $PTE_{AVA[0:q]}$=$VA_{0:q}$
- if b = 12 then
  (PTE_L = 0) | (PTE_{LP} specifies the 4KB base page size)
  else
  (PTE_L = 1) & (PTE_{LP} specifies the base page size specified by $SLBE_{L||LP}$)

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the Page Table search succeeds, the real address (RA) is formed by concatenating bits 0:59-p of ARPN||LP from the matching PTE with bits 64-p:63 of the effective address (the byte offset), where the p value is the $\log_2$ (actual page size specified by $PTE_{L\ LP}$).

$$RA=(ARPN \parallel LP)_{0:59-p} \parallel EA_{64-p:63}$$

Programming Note

If PTE_L = 0, the actual page size (and base page size) are 4 KB. Otherwise the actual page size and base page size are specified by PTE_{LP}.

Since hardware searches the Page Table using a value of b equal to $\log_2$ (base page size specified in the SLBE that was used to translate the address) regardless of the actual page size, the hardware page table search will identify different PTEs for VAs in different 2^b-byte blocks of the virtual page if the actual page size is larger than the base page size. Therefore, there may need to be a valid PTE corresponding to each 2^b-byte block of the virtual page that is referenced. For an actual page size that is larger than 2^{23} (8 MB), the PTE_{AVA} will differ among some or all of these PTEs. Depending on the Page Table size, some or all of these PTEs may be in the same PTEG. Any such PTEs that are in the same PTEG will differ in the value of PTE_H or PTE_{AVA} or both.

All PTEs for the same virtual page should have the same values in the Page Protection, KEY, ARPN, WIMG, and N fields. A set of values from any one of the PTEs that maps the virtual page may be used for an access in the virtual page since lookaside buffer information may be used to translate the virtual address.

To avoid creating multiple matching PTEs, software should not create PTEs for each of two different virtual pages that overlap in the virtual address space. If the virtual page sizes differ, two virtual pages overlap if the values of virtual address bits 0:77-p for both virtual pages are the same, where 2^p is the actual virtual page size of the larger page.

The N (No-execute) value used for the storage access is the result of ORing the N bit from the matching PTE with the N bit from the SLB entry that was used to translate the effective address.

Programming Note

The value of b used when searching the Page Table to identify the PTEGs is $\log_2$(base page size). Since a segment may contain pages of different sizes, the hardware searches for PTEs specifying pages of any supported size greater than or equal to the base page size, and the real address is formed using the value of p specified by the matching PTE.

A virtual page of 2^p bytes in a segment with a base page size of 2^b bytes may be mapped by as many as $2^{(p-b)}$ PTEs.

If the Page Table search fails, a *page fault* occurs. This is a [Hypervisor] Instruction Storage exception or a [Hypervisor] Data Storage exception, depending on

whether the effective address is for an instruction fetch or for a data access. The N value used for the storage access is the N bit from the SLB entry that was used to translate the effective address.

Programming Note

To obtain the best performance, Page Table Entries should be allocated beginning with the first empty entry in the primary PTEG, or with the first empty entry in the secondary PTEG if the primary PTEG is full and the secondary Page Table search is enabled ($LPCR_{TC}$=0).

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. Even though multiple PTEs may be needed for a virtual page whose size is larger than the base page size, one TLB entry derived from a single PTE may be used to translate all of the virtual addresses in the entire virtual page. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table (see Section 5.10).

In the TLB search, the match criteria include virtual address bits 0:(77-q) where q is an implementation-dependent integer such that $b \le q \le p$. As a result of a Page Table search, multiple matching TLB entries are not created for the same virtual page, except that multiple matching TLB entries may be created if the Page Table contains PTEs that map different-sized virtual pages that overlap in the virtual address space. (If the virtual page sizes differ, two virtual pages overlap if the values of virtual address bits 0:77-p for both virtual pages are the same, where 2^p is the actual virtual page size of the larger page.) If a TLB search finds multiple matching TLB entries created from such PTEs, one of the matching TLB entries is used as if it were the only matching entry, or a Machine Check occurs.

As a result of a Page Table search in a Page Table that does not contain different-sized virtual pages that overlap, it is implementation-dependent whether multiple non-matching TLB entries are created for the same virtual page. However, in this case if multiple TLB entries are created for a given virtual page, at most one matching TLB entry is created for any given virtual address in that virtual page, and q for that TLB entry is less than p.

An implementation may associate each of its TLB entries with the partition for which the TLB entry was created, so that the entries can be retained while other partitions are executing. In this case, when a valid TLB entry is created, the LPID value from LPIDR is written into the TLB entry.

Programming Notes

1. Page Table Entries may or may not be cached in a TLB.
2. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein.
3. Use the ***tlbie*** or ***tlbia*** instruction to ensure that the TLB no longer contains a mapping for a particular virtual page.

5.7.7.4 Relaxed Page Table Alignment [Category: Server.Relaxed Page Table Alignment]

The Page Table can be aligned on any 2^{18} byte (256 KB) boundary regardless of the HTAB size.

Section 5.7.7.2 describes the Storage Description Register, which includes the HTABORG field. That description generally applies except for the following difference. As the Page Table size is increased beyond 256 KB, the value in HTABORG need not have more of its low-order bits equal to 0. Instead, (HTABORG || $^{18}0$) is the real address of the start of the Page Table regardless of the Page Table size.

A Page Table search is performed as described in Section 5.7.7.3 except the 60-bit real address of a PTEG for both the primary and, if the secondary Page Table search is enabled, the secondary hash is formed by concatenating the following values:

- Bits 0:27 of the 39-bit appropriate primary or secondary hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then added to the value of bits 4:45 of SDR1 (HTABORG). This part of the real address differs from Section 5.7.7.2.
- Bits 28:38 of the 39-bit hash value.
- Seven 0-bits.

An outline of the PTEG real address computation is shown in Figure 23.

5.7.8 Reference and Change Recording

If address translation is enabled, Reference (R) and Change (C) bits are updated in any one of what could be multiple Page Table Entries that map the virtual page that is being accessed. If the storage operand of a *Load* or *Store* instruction crosses a virtual page boundary, the accesses to the components of the operand in each page are treated as separate and independent

accesses to each of the pages for the purpose of setting the Reference and Change bits.

Reference and Change bits are set by the hardware as described below. Setting the bits need not be atomic with respect to performing the access that caused the bits to be updated. An attempt to access storage may cause one or more of the bits to be set (as described below) even if the access is not performed. The bits are updated in the Page Table Entry if the new value would otherwise be different from the old value for the virtual page, as determined by examining either the Page Table Entry or any lookaside information for the virtual page (e.g., TLB) maintained by the hardware.

Reference Bit

The Reference bit is set to 1 if the corresponding access (load, store, or instruction fetch) is required by the sequential execution model and is performed. Otherwise the Reference bit may be set to 1 if the corresponding access is attempted, either in-order or out-of-order, even if the attempt causes an exception, except that the Reference bit is not set to 1 for the access caused by an indexed *Move Assist* instruction for which the XER specifies a length of zero.

Change Bit

The Change bit is set to 1 if a *Store* instruction is executed and the store is performed. Otherwise in general the Change bit may be set to 1 if a *Store* instruction is executed and the store is permitted by the storage protection mechanism and, if the *Store* instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the following kinds of interrupts:

- system-caused interrupts (see Section 6.4 on page 946)
- Floating-Point Enabled Exception type Program interrupts when the thread is in an Imprecise mode.

The only exception to the preceding statement is that the Change bit is not set to 1 if the instruction is a *Store String Indexed* instruction for which the XER specifies a length of zero.

Programming Note

A virtual page in a segment with a smaller base page size may be mapped by multiple PTEs. For each access of a virtual page, hardware may search the Page Table to update the R and C bits. If lookaside buffer information for the virtual page already indicates that all such bits to be set have already been set in a PTE that maps the virtual page, hardware need not make an update. Consider the following sequence of events:

1. A virtual page is mapped by 2 PTEs A and B and the R and C bits in both PTEs are 0.
2. A Load instruction accesses the virtual page and the R bit is updated in PTE A.
3. A Load instruction accesses the virtual page and the R bit is updated in PTE B.
4. A Store instruction accesses the virtual page and the C bit is updated in PTE B.
5. The virtual page is paged out. Software must examine both PTE A and B to get the state of the R and C bits for the virtual page.

Furthermore, if in event 2, PTE A was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur. Subsequently, if in event 3 or 4, PTE B was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur.

Programming Note

Even though the execution of a *Store* instruction causes the Change bit to be set to 1, the store might not be performed or might be only partially performed in cases such as the following.

- A *Store Conditional* instruction (***stwcx.*** or ***stdcx.***) is executed, but no store is performed.
- The *Store* instruction causes a Data Storage exception (for which setting the Change bit is not prohibited).
- The *Store* instruction causes an Alignment exception.
- The Page Table Entry that translates the virtual address of the storage operand is altered such that the new contents of the Page Table Entry preclude performing the store (e.g., the PTE is made invalid, or the PP bits are changed).

  For example, when executing a *Store* instruction, the thread may search the Page Table for the purpose of setting the Change bit and then re-execute the instruction. When reexecuting the instruction, the thread may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another thread, the second search may obtain the new contents, which may preclude the store.
- A system-caused interrupt occurs before the store has been performed.

When the hardware updates the Reference and Change bits in the Page Table Entry, the accesses are performed as described in Section 5.7.3.5, “Storage Control Attributes for Implicit Storage Accesses” on page 895. The accesses may be performed using operations equivalent to a store to a byte, halfword, word, or doubleword, and are not necessarily performed as an atomic read/modify/write of the affected bytes.

These Reference and Change bit updates are not necessarily immediately visible to software. Executing a ***sync*** instruction ensures that all Reference and Change bit updates associated with address translations that were performed, by the thread executing the ***sync*** instruction, before the ***sync*** instruction is executed will be performed with respect to that thread before the ***sync*** instruction’s memory barrier is created. There are additional requirements for synchronizing Reference and Change bit updates in multi-threaded systems; see Section 5.10, “Page Table Update Synchronization Requirements” on page 935.

Programming Note

Because the ***sync*** instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the thread executing the ***sync*** instruction before the memory barrier is created includes all Reference and Change bit updates associated with instructions preceding the ***sync*** instruction.

If software refers to a Page Table Entry when MSR_{DR}=1, the Reference and Change bits in the associated Page Table Entry are set as for ordinary loads and stores. See Section 5.10 for the rules software must follow when updating Reference and Change bits.

Figure 27 on page 907 summarizes the rules for setting the Reference and Change bits. The table applies to each atomic storage reference. It should be read from the top down; the first line matching a given situation applies. For example, if ***stwcx.*** fails due to both a storage protection violation and the lack of a reservation, the Change bit is not altered.

In the figure, the “*Load*-type” instructions are the *Load* instructions described in Books I, II, and III-S, ***eciwx***, and the *Cache Management* instructions that are treated as *Loads*. The “*Store*-type” instructions are the *Store* instructions described in Books I, II, and III-S, ***ecowx***, and the *Cache Management* instructions that are treated as *Stores*. The “ordinary” *Load* and *Store* instructions are those described in Books I, II, and III-S. “set” means “set to 1”.

Status of Access	R	C
Indexed *Move Assist* insn w 0 len in XER	No	No
Storage protection violation	Acc[1]	No
Out-of-order I-fetch or Load-type Inst'n (including transactional Load-type inst'n or ***dcbtst***)	Acc	No
Out-of-order Store-type inst'n, including transactional Store-type inst'n, excluding ***dcbtst***		
Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts[3], or transaction failure	Acc	Acc[1 2]
All other cases	Acc	No
In-order *Load*-type or *Store*-type insn, access not performed[4]		
Load-type insn	Acc	No
Store-type insn	Acc	Acc[2]
Other in-order access		
I-fetch	Yes	No
Ordinary *Load*, ***eciwx***	Yes	No
Other ordinary *Store*, ***ecowx***, ***dcbz***	Yes	Yes
icbi, ***icbt***, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***[***l***]	Acc	No

"Acc" means that it is acceptable to set the bit.

1 It is preferable not to set the bit.

2 If C is set, R is also set unless it is already set.

3 For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by $MSR_{FE0\ FE1}$.

4 This case does not apply to the *Touch* instructions, because they do not cause a storage access.

Figure 27. Setting the Reference and Change bits

5.7.9 Storage Protection

The storage protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the storage protection mechanism depends on the contents of one or more of the following.

- MSR bits HV, IR, DR, PR
- the key bits in the associated SLB entry
- the page protection bits and key bits in the associated PTE
- the AMR, IAMR, AMOR, and UAMOR
- LPCR bit VPM_0

The storage protection mechanism consists of the Virtual Page Class Key Protection mechanism, described in Section 5.7.9.1, and the Basic Storage Protection mechanism, described in Section 5.7.9.2 and Section 5.7.9.3.

When address translation is enabled for an access, the access is permitted if and only if the access is permitted by both the Virtual Page Class Key Protection mechanism and the Basic Storage Protection mechanism. When address translation is disabled for an access, the access is permitted if and only if the access is permitted by the Basic Storage Protection mechanism. If an instruction fetch is not permitted, an Instruction Storage exception or a Hypervisor Instruction Storage exception is generated. If a data access is not permitted, a Data Storage exception or a Hypervisor Data Storage exception is generated.

A *protection domain* is a maximal range of effective addresses for which variables related to storage protection can be independently specified (including by default, as in real and hypervisor real addressing modes), or a maximal range of addresses, effective or virtual, for which variables related to storage protection cannot be specified. Examples include: a segment, a virtual page (including for a virtualized Real Mode Area), the Real Mode Area (regardless of whether the RMA is virtualized), the effective address range $0:2^{60}-1$ in hypervisor real addressing mode, and a maximal range of effective or virtual addresses that cannot be mapped to real addresses. A *protection boundary* is a boundary between protection domains.

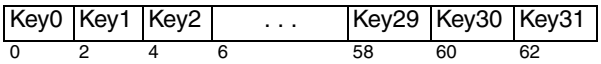
5.7.9.1 Virtual Page Class Key Protection

The Virtual Page Class Key Protection mechanism provides the means to assign virtual pages to one of 32 classes, and to modify data access permissions for each class by modifying the Authority Mask Register (AMR), shown in Figure 28, and to modify instruction access permissions for each class by modifying the Instruction Authority Mask Register (IAMR) shown in Figure 29.

Programming Note

If address translation is disabled for a given access, the access is not affected by the Virtual Page Class Key Protection mechanism even if the access is made in virtual real addressing mode.

Authority Mask Register



Bits	Name	Description
0:1	Key0	Access mask for class number 0
2:3	Key1	Access mask for class number 1
…	…	…
2n:2n+1	Keyn	Access mask for class number n
…	…	…
62:63	Key31	Access mask for class number 31

Figure 28. Authority Mask Register (AMR)

The access mask for each class defines the access permissions that apply to loads and stores for which the virtual address is translated using a Page Table Entry that contains a KEY field value equal to the class number. The access permissions associated with each class are defined as follows, where AMR_{2n} and AMR_{2n+1} refer to the first and second bits of the access mask corresponding to class number n.

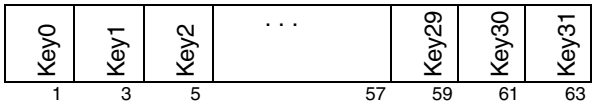
- A store is permitted if AMR_{2n}=0b0; otherwise the store is not permitted.
- A load is permitted if AMR_{2n+1}=0b0; otherwise the load is not permitted.

The AMR can be accessed using either SPR 13 or SPR 29. Access to the AMR using SPR 29 is privileged.

Programming Note

Because the AMR is part of the program context (if address translation is enabled), and because it is desirable for most application programmers not to have to understand the software synchronization requirements for context alterations (or the nuances of address translation and storage protection), operating systems should provide a system library program that application programs can use to modify the AMR.

Instruction Authority Mask Register



Bits	Name	Description
0	Resv'd.	
1	Key0	Access mask for class number 0
2	Resv'd	
3	Key1	Access mask for class number 1
…	…	…
2n	Resv'd	
2n+1	Keyn	Access mask for class number n
…	…	…
62	Resv'd.	
63	Key31	Access mask for class number 31

Figure 29. Instruction Authority Mask Register (IAMR)

The access mask for each class defines the access permissions that apply to instruction fetches for which the virtual address is translated using a Page Table Entry that contains a KEY field value equal to the class number. The access permission associated with each class is defined as follows, where $IAMR_{2n+1}$ refers to the bit of the access mask corresponding to class number n.

- An instruction fetch is permitted if $IAMR_{2n+1}$=0b0; otherwise the instruction fetch is not permitted.

Access to the IAMR is privileged.

The Authority Mask Override Register (AMOR) and the User Authority Mask Override Register (UAMOR), shown in Figure 30 and Figure 31 respectively, can be used to restrict modifications (***mtspr***) of the AMR. Also, the AMOR can be used to restrict modifications of the UAMOR and IAMR. Access to both the AMOR and UAMOR is privileged. The AMOR is a hypervisor resource.

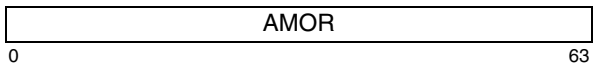


Figure 30. Authority Mask Override Register (AMOR)

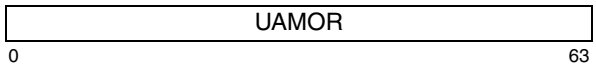


Figure 31. User Authority Mask Override Register (UAMOR)

The bits of the AMOR and UAMOR are in 1-1 correspondence with the bits of the AMR (i.e., $[U]AMOR_i$ corresponds to AMR_i). The AMOR affects modifications of the AMR and UAMOR in privileged but non hypervisor state; the UAMOR affects modifications of the AMR in problem state.

Similarly, the odd bits of the AMOR are in 1-1 correspondence with the odd bits of the IAMR (i.e., $AMOR_{2j+1}$ corresponds to $IAMR_{2j+1}$). The AMOR affects modifications of the IAMR in privileged but non hypervisor state; the IAMR cannot be accessed in problem state.

- When ***mtspr*** specifying the AMR (using either SPR 13 or SPR 29) or the IAMR is executed in privileged but non-hypervisor state, the AMOR is used as a mask that controls which bits of the resulting AMR or IAMR contents come from register RS and which AMR or IAMR bits are not modified.
- Similarly, when ***mtspr*** specifying the AMR (using SPR 13) is executed in problem state, the UAMOR is used as a mask that controls which bits of the resulting AMR contents come from register RS and which AMR bits are not modified.
- When ***mtspr*** specifying the UAMOR is executed in privileged but non-hypervisor state, the AMOR is ANDed with the contents of register RS and the result is placed into the UAMOR; the AMOR thereby controls which bits of the resulting UAMOR contents come from register RS and which UAMOR bits are set to zero.

A complete description of these effects can be found in the description of the ***mtspr*** instruction on page 1053.

Software must ensure that both bits of each even/odd bit pair of the AMOR contain the same value. — i.e., the contents of register RS for ***mtspr*** specifying the AMOR must be such that $(RS)_{2n} = (RS)_{2n+1}$ for every n in the range 0:31 — and likewise for the UAMOR. If this requirement is violated for the UAMOR the results of accessing the UAMOR (including implicitly by the hardware as described in the second item of the preceding list) are boundedly undefined; if the requirement is violated for the AMOR the results of accessing the AMOR (including implicitly by the hardware as described in the first and third items of the list) are undefined.

Programming Note

The preceding requirement permits designs to implement the AMOR and/or UAMOR as 32-bit registers — specifically, to implement only the even-numbered bits (or only the odd-numbered bits) of the register — in a manner such that the reduction, from the architecturally-required 64 bits to 32 bits, is not visible to (correct) software. This implementation technique saves space in the hardware. (A design that uses this technique does the appropriate "fan in/out" when the register is accessed, to provide the appearance, to (correct) software, of supporting all 64 bits of the register.)

Permitting designs to implement the [U]AMOR as 32-bit registers by virtue of the software requirement specified above, rather than by defining the [U]AMOR as 32-bit registers, permits the architecture to be extended in the future to support controlling modification of the "read access" AMR bits (the odd-numbered bits) independently from the "write access" AMR bits (the even-numbered bits), if that proves desirable. If this independent control does prove desirable, the only architecture change would be to eliminate the software requirement.

Programming Note

When modifying the AMOR and/or UAMOR, the hypervisor should ensure that the two registers are consistent with one another before giving control to a non-hypervisor program. In particular, the hypervisor should ensure that if $AMOR_i$=0 then $UAMOR_i$=0, for all i in the range 0:63. (Having $AMOR_i$=0 and $UAMOR_i$=1 would permit problem state programs, but not the operating system, to modify AMR bit i.)

Programming Note

The Virtual Page Class Key Protection mechanism replaces the Data Address Compare mechanism that was defined in versions of the architecture that precede Version 2.04 (e.g., the two facilities use some of the same resources, as described below). However, the Virtual Page Class Key Protection mechanism can be used to emulate the Data Address Compare mechanism. Moreover, programs that use the Data Address Compare mechanism can be modified in a manner such that they will work correctly both on implementations that comply with versions of the architecture that precede Version 2.04 (and hence implement the Data Address Compare mechanism) and on implementations that comply with Version 2.04 of the architecture or with any subsequent version (and hence instead implement the Virtual Page Class Key Protection mechanism). The technique takes advantage of the facts that the SPR number for privileged access to the AMR (29) is the same as the SPR number for the Data Address Compare mechanism's ACCR (Address Compare Control Register), that KEY_4 occupies the same bit in the PTE as the Data Address Compare mechanism's AC (Address Compare) bit, and that the definition of $ACCR_{62:63}$ is very similar to the definition of each even-odd pair of AMR bits. The technique is as follows, where PTE1 refers to doubleword 1 of the PTE.

- Set bits 2:3 and 62:63 of SPR 29 (which is either the ACCR or the AMR) to x, where x is the desired 2-bit value for controlling Data Address Compare matches, and set bits 0:1 to 0s.
- Set $PTE1_{54}$ (which is either the AC bit or KEY_4) to the same value that the AC bit would be set to, and set $PTE1_{2:3}$ (which are either RPN bits, that correspond to a real address size larger than the size supported by any implementation that supports the Data Address Compare mechanism, or $KEY_{0:1}$) and $PTE1_{52:53}$ (which are either reserved bits or $KEY_{2:3}$) to 0s.
- Use PTE_{KEY} values 0 and 1 only for purposes of emulating the Data Address Compare mechanism, except that PTE_{KEY} value 0 may also be used for any virtual pages for which it is desired that the Virtual Page Class Key Protection mechanism permit all accesses. Do not use PTE_{KEY} =31.
- When a Hypervisor Data Storage interrupt occurs, if $HDSISR_{42}$=1 then ignore the interrupt for *Cache Management* instructions other than ***dcbz***. (These instructions can cause a virtual page class key protection violation but cannot cause a Data Address Compare match.) Otherwise forward the interrupt to the operating system, which will treat the interrupt as if a Data Address Compare match had occurred. (Note: Cases for which it is undefined whether a Data Address Compare match occurs do not necessarily cause a virtual page class key protection violation.)

(Because privileged software can access the AMR using either SPR 13 or SPR 29, it might seem that, when SPR 13 was added to the architecture (in Version 2.06), SPR 29 should have been removed. SPR 29 is retained for two reasons: first, to avoid requiring privileged software to change to use the newer SPR number; and second, to retain the ability to emulate the Data Address Compare mechanism as described above.)

Programming Note

An example of the use of the AMOR (and UAMOR) is to support lightweight partitions, here called "adjunct" partitions, that provide services (e.g., device drivers) to "client" partitions. The adjunct partition would be managed by the hypervisor. It would run in problem state with $MSR_{HV\ PR}$=0b11, thereby restricting the resources it can modify (MSR_{PR}=1) and causing its interrupts to go to the hypervisor (MSR_{HV}=1), and it would share a Page Table with the client partition it serves. Typically, each of the two partitions would have data storage that the other partition must not be able to access. The hypervisor can use the AMOR, UAMOR, AMR, and PTE KEY field to provide the required protection. (The adjunct partition's lightness of weight derives from not requiring an operating system, and especially from not requiring a full partition context switch (SLB flush, TLB flush, SDR1 change, etc.) when the client partition invokes the services of the adjunct partition.)

For example, suppose each of the two partitions must not be able to access any of the other partition's data storage. The hypervisor could use KEY value j for all data virtual pages that only the adjunct partition must be able to access. Before dispatching the client partition for the first time, the hypervisor would initialize the three registers as follows.

AMR: all 0s except bits 2j and 2j+1, which would contain 1s

UAMOR: all 0s

AMOR: all 1s except bits 2j and 2j+1, which would contain 0s

Before dispatching the adjunct partition, the hypervisor would set UAMOR to all 0s, and would set the AMR to all 1s except bits 2j and 2j+1, which would be set to 0s. (Because the adjunct partition would run in problem state, there is no need for the hypervisor to modify the AMOR, and the adjunct partition cannot modify the UAMOR.) In addition, the hypervisor would prevent the client partition from modifying or deleting PTEs that contain translations used by the adjunct partition.

(It may be desirable to avoid using KEY values 0, 1, and 31 for storage that only the adjunct partition can access, because these KEY values may be needed by the client partition to emulate the Data Address Compare mechanism, as described above. Also, old software, that was written for an implementation that complies with a version of the architecture that precedes Version 2.04 (the version in which virtual page class keys were added), effectively uses KEY 0 for all virtual pages.)

Programming Note

Initialization of the UAMOR to all 0s, by the hypervisor before dispatching a partition for the first time, as described in the preceding Programming Note, permits operating systems (in partitions that run in a compatibility mode corresponding to Version 2.06 of the architecture or a subsequent version) to migrate gradually to supporting problem state access to the AMR — specifically, to avoid having to be changed immediately to modify the UAMOR and to save the AMR contents when an interrupt occurs from problem state. Relatedly, having the UAMOR contain all 0s while an application program is running protects old application programs that are "AMR-unaware". In the absence of programming errors, such application programs would not attempt to read or modify the AMR. However, having the UAMOR contain all 0s protects such programs against modifying the AMR inadvertently.

Permitting an "AMR-unaware" application program to modify the AMR (inadvertently) is potentially harmful for the obvious reasons. (The program might set to 1 an AMR bit corresponding to accesses that are necessary in order for the program to work correctly.) Moreover, even for an operating system that includes support for problem state modification of the AMR, having the UAMOR contain all 0s allows the operating system to avoid saving and restoring the AMR for "AMR-unaware" application programs. Such an operating system would provide a system service program that allows an application program to declare itself to be "AMR-aware" — i.e., potentially to need to modify the AMR. When an application program invokes this service, the operating system would set the UAMOR to the non-zero value appropriate to the access authorities (load and/or store, for one or more key values) that the application program is allowed to modify, and thereafter would save and restore the AMR (and preserve the UAMOR) for this application program. (Having the UAMOR contain all 0s does not prevent an "AMR-unaware" program from reading the AMR, but inadvertent reading of the AMR is likely to be much less harmful than inadvertently modifying it.)

(For partitions that run in a compatibility mode corresponding to a version of the architecture that precedes Version 2.06, the PCR provides sufficient protection to application programs.)

5.7.9.2 Basic Storage Protection, Address Translation Enabled

When address translation is enabled, the Basic Storage Protection mechanism is controlled by the following.

- MSR_{PR}, which distinguishes between supervisor (privileged) state and problem state

- K_s and K_p, the supervisor (privileged) state and problem state storage key bits in the SLB entry used to translate the effective address
- PP, page protection bits 0:2 in the Page Table Entry used to translate the effective address
- For instruction fetches only:
  - the N (No-execute) value used for the access (see Sections 5.7.6.1 and 5.7.7.3)
  - PTE_G, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.

1. For an instruction fetch, the access is not permitted if the N value is 1 or if PTE_G=1.
2. For any access except an instruction fetch that is not permitted by rule 1, a "Key" value is computed using the following formula:

   $Key \leftarrow (K_p \ \& \ MSR_{PR}) \ | \ (K_s \ \& \ \neg MSR_{PR})$

   Using the computed Key, Figure 32 is applied. An instruction fetch is permitted for any entry in the figure except "no access". A load is permitted for any entry except "no access". A store is permitted only for entries with "read/write".

Key	PP	Access Authority
0	000	read/write
0	001	read/write
0	010	read/write
0	011	read only
0	110	read only
1	000	no access
1	001	read only
1	010	read/write
1	011	read only
1	110	no access
All PP encodings not shown above are reserved. The results of using reserved PP encodings are boundedly undefined.		

Figure 32. PP bit protection states, address translation enabled

5.7.9.3 Basic Storage Protection, Address Translation Disabled

When address translation is disabled, the Basic Storage Protection mechanism is controlled by the following (see Chapter 2 and Section 5.7.3, "Real And Virtual Real Addressing Modes").

- MSR_{HV}, which (when MSR_{PR}=0) distinguishes between hypervisor state and privileged but non-hypervisor state
- VPM_0, which distinguishes between real addressing mode and virtual real addressing mode
- RMLS, which specifies the real mode limit value

Using the above values, the following rules are applied.

1. If MSR_{HV}=0 and VPM_0=1, access authority is determined as described in Section 5.7.3.4.
2. If MSR_{HV}=1 or VPM_0=0, Figure 33 is applied. The access is permitted for any entry in the figure except "no access".

HV	Access Authority
0	read/write or no access [1]
1	read/write
[1] If the effective address for the access is less than the value specified by the RMLS, the access authority is read/write; otherwise the access is not permitted.	

Figure 33. Protection states, address translation disabled

Programming Note

The comparison described in note 1 in Figure 33 ignores bits 0:3 of the effective address and may ignore bits 4:63-m; see Section 5.7.3.

5.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage control attributes may be found in Section 1.6 of Book II and subsections.

5.8.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if any of the following conditions is satisfied.

- MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, and the G bit is 1 in the relevant Page Table Entry.
- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, MSR_{HV}=1, and the storage is outside the range(s) specified by the Hypervisor Real Mode Storage Control facility (see Section 5.7.3.3.1).

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

The following rules apply to in-order execution of *Load* and *Store* instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- *Load* or *Store* instruction that causes an atomic access

  If any portion of the storage operand has been accessed and an External, Decrementer, Hypervisor Decrementer, Performance Monitor, or Imprecise mode Floating-Point Enabled exception is pending, the instruction completes before the interrupt occurs.

- *Load* or *Store* instruction that causes an Alignment exception, or that causes a [Hypervisor] Data Storage exception for reasons other than Data Address Watchpoint match.

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

  (The corresponding rules for instructions that cause a Data Address Watchpoint match are given in Section 8.1.3.)

5.8.1.1 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exceptions to this rule are the following.

Load Instruction

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

Instruction Fetch

If $MSR_{HV\ IR}$=0b10 then an instruction may be fetched if any of the following conditions are met.

1. The instruction is in a cache. In this case it may be fetched from the cache or from main storage.
2. The instruction is in a real page from which an instruction has previously been fetched, except that if that previous fetch was based on condition 1 then the previously fetched instruction must have been in the instruction cache.
3. The instruction is in the same real page as an instruction that is required by the sequential execution model, or is in the real page immediately following such a page.

Programming Note

Software should ensure that only well-behaved storage is copied into a cache, either by accessing as Caching Inhibited (and Guarded) all storage that may not be well-behaved, or by accessing such storage as not Caching Inhibited (but Guarded) and referring only to cache blocks that are well-behaved.

If a real page contains instructions that will be executed when MSR_{IR}=0 and MSR_{HV}=1, software should ensure that this real page and the next real page contain only well-behaved storage (or that the Hypervisor Real Mode Storage Control facility specifies that this real page is not Guarded).

5.8.2 Storage Control Bits

When address translation is enabled, each storage access is performed under the control of the Page Table Entry used to translate the effective address. Each Page Table Entry contains storage control bits that specify the presence or absence of the corre-

sponding storage control for all accesses translated by the entry as shown in Figure 34.

Bit	Storage Control Attribute
W[1,3]	0 - not Write Through Required 1 - Write Through Required
I[3]	0 - not Caching Inhibited 1 - Caching Inhibited
M[2]	0 - not Memory Coherence Required 1 - Memory Coherence Required
G	0 - not Guarded 1 - Guarded

1 Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.

2 [Category: Memory Coherence] Support for the 0 value of the M bit is optional, implementations that do not support the 0 value assume the value of the bit to be 1, and may either preserve the value of the bit or write it as 1.

3 [Category: SAO] The combination WIMG = 0b1110 has behavior unrelated to the meanings of the individual bits. See see Section 5.8.2.1, "Storage Control Bit Restrictions" for additional information.

Figure 34. Storage control bits

When address translation is enabled, instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1; see Section 5.7.9.

When address translation is disabled, the storage control attributes are implicit; see Section 5.7.3.3.

In Sections 5.8.2.1 and 5.8.2.2, "access" includes accesses that are performed out-of-order, and references to W, I, M, and G bits include the values of those bits that are implied when address translation is disabled.

Programming Note

In a system consisting of only a single-threaded processor which has caches, correct coherent execution does not require storage to be accessed as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

5.8.2.1 Storage Control Bit Restrictions

All combinations of W, I, M, and G values are permitted except those for which both W and I are 1 and M||G ≠ 0b10.

The combination WIMG = 0b1110 is used to identify the Strong Access Ordering (SAO) storage attribute (see Section 1.7.1, "Storage Access Ordering", in Book II). Because this attribute is not intended for general purpose programming, it is provided only for a single combination of the attributes normally identified using the WIMG bits. That combination would normally be indicated by WIMG = 0b0010.

References to Caching Inhibited storage (or storage with I=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the encoding using I=1. Conversely, references to storage that is not Caching Inhibited (or storage with I=0) apply to SAO storage or its WIMG encoding. References to Write Through Required storage (or storage with W=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the fact that the encoding uses W=1. Conversely, references to storage that is not Write Through Required (or storage with W=0) apply to SAO storage or its WIMG encoding.

If a given real page is accessed concurrently as SAO storage and as non-SAO storage, the result may be characteristic of the weakly consistent model.

Programming Note

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0. For implementations that support the SAO category, the operating system should provide a means by which application programs can request SAO storage, in order to avoid confusion with the preceding guideline (since SAO is encoded using WI=0b11).

At any given time, the value of the W bit must be the same for all accesses to a given real page.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

5.8.2.2 Altering the Storage Control Bits

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no thread modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using ***dcbst*** or ***dcbf***[***l***].

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using ***dcbf***[***l***] and ***icbi*** before permitting any other accesses to the page. Note that similar cache management is required before using the Fixed-Point Load and Store Caching Inhibited instructions to

access storage that has formerly been cached. (See Section 4.4.1 on page 875.)

Programming Note

The storage control bit alterations described above are examples of cases in which the directives for application of statements about the W and I bits to SAO given in the third paragraph of the preceding subsection must be applied. A transition from the typical WIMG=0b0010 for ordinary storage to WIMG=0b1110 for SAO storage does not require the flush described above because both WIMG combinations indicate storage that is not Caching Inhibited.

Programming Note

It is recommended that ***dcbf*** be used, rather than ***dcbfl***, when changing the value of the I or W bit from 0 to 1. (***dcbfl*** would have to be executed on all threads for which the contents of the data cache may be inconsistent with the new value of the bit, whereas, if the M bit for the page is 1, ***dcbf*** need be executed on only one thread in the system.)

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this are system-dependent.

Programming Note

For example, when changing the M bit in some directory-based systems, software may be required to execute ***dcbf***[***l***] on each thread to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

Additional requirements for changing the storage control bits in the Page Table are given in Section 5.10.

5.9 Storage Control Instructions

5.9.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a ***dcbz*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the hardware need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 5.6) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the thread enters any power conserving mode in which data cache contents are not maintained.

5.9.2 Synchronize Instruction

The *Synchronize* instruction is described in Section 4.4.3 of Book II, but only at the level required by an application programmer. This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers.

When L=0, the ***sync*** instruction also provides an ordering function for the operations caused by the *Message Send* instruction and previous *Stores*. The stores must be performed with respect to the processor receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt occurs.

When L=1, the ***sync*** instruction provides an ordering function for the operations caused by the *Message Send* instruction and previous *Stores* for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The stores must be performed with respect to the processor receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt occurs.

Another variant of the *Synchronize* instruction is described below. It is designated the Page Table Entry Synchronize instruction, and is specified by the extended mnemonic ***ptesync*** (equivalent to ***sync*** with L=2).

The ***ptesync*** instruction has all of the properties of ***sync*** with L=0 and also the following additional properties.

- The memory barrier created by the ***ptesync*** instruction provides an ordering function for the storage accesses associated with all instructions that are executed by the thread executing the ***ptesync*** instruction and, as elements of set A, for all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed. The applicable pairs are all pairs a_i,b_j in which b_j is a data access and a_i is not an instruction fetch.
- The ***ptesync*** instruction causes all Reference and Change bit updates associated with address translations that were performed, by the thread executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, to be performed with respect to that thread before the ***ptesync*** instruction's memory barrier is created.
- The ***ptesync*** instruction provides an ordering function for all stores to the Page Table caused by *Store* instructions preceding the ***ptesync*** instruction with respect to searches of the Page Table that are performed, by the thread executing the ***ptesync*** instruction, after the ***ptesync*** instruction completes. Executing a ***ptesync*** instruction ensures that all such stores will be performed, with respect to the thread executing the ***ptesync*** instruction, before any implicit accesses to the affected Page Table Entries, by such Page Table searches, are performed with respect to that thread.
- In conjunction with the ***tlbie*** and ***tlbsync*** instructions, the ***ptesync*** instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the ***tlbsync*** instruction description on page 934.

Programming Note

For instructions following a ***ptesync*** instruction, the memory barrier need not order implicit storage accesses for purposes of address translation and reference and change recording.

The functions performed by the ***ptesync*** instruction may take a significant amount of time to complete, so this form of the instruction should be used only if the functions listed above are needed. Otherwise ***sync*** with L=0 should be used (or ***sync*** with L=1, or ***eieio***, if appropriate).

Section 5.10, "Page Table Update Synchronization Requirements" on page 935 gives examples of uses of ***ptesync***.

5.9.3 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB). For performance reasons, most implementations also have implementation-specific lookaside information that is used in address translation. This lookaside information may be: a Translation Lookaside Buffer (TLB) which is a cache of recently used Page Table Entries (PTEs); a cache of recently used translations of effective addresses to real addresses; etc.; or any combination of these. Lookaside information, including the SLB, is managed using the instructions described in the subsections of this section.

Lookaside information derived from PTEs is not necessarily kept consistent with the Page Table. When software alters the contents of a PTE, in general it must also invalidate all corresponding implementation-specific lookaside information; exceptions to this rule are described in Section 5.10.1.2.

The effects of the ***slbie***, ***slbia***, and *TLB Management* instructions on address translations, as specified in Sections 5.9.3.1 and 5.9.3.3 for the SLB and TLB respectively, apply to all implementation-specific lookaside information that is used in address translation. Unless otherwise stated or obvious from context, references to SLB entry invalidation and TLB entry invalidation elsewhere in the Books apply also to all implementation-specific lookaside information that is derived from SLB entries and PTEs respectively.

The ***tlbia*** instruction is optional. However, all implementations provide a means by which software can invalidate all implementation-specific lookaside information that is derived from PTEs.

Implementation-specific lookaside information that contains translations of effective addresses to real addresses may include "translations" that apply in real addressing mode. Because such "translations" are affected by the contents of the LPCR, RMOR, and HRMOR, when software alters the contents of these registers it must also invalidate the corresponding implementation-specific lookaside information. Software can invalidate all such lookaside information by using the ***slbia*** instruction with IH=0b000. However, performance is likely to be better if other, appropriate, IH values are used to limit the amount of lookaside information that invalidated.

All implementations that have such lookaside information provide a means by which software can invalidate all such lookaside information.

For simplicity, elsewhere in the Books it is assumed that the TLB exists.

Programming Note

Because the instructions used to manage implementation-specific lookaside information that is derived from PTEs may be changed in a future version of the architecture, it is recommended that software "encapsulate" uses of the *TLB Management* instructions into subroutines.

Programming Note

The function of all the instructions described in Sections 5.9.3.1 - 5.9.3.3 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 12.

5.9.3.1 SLB Management Instructions

Programming Note

Accesses to a given SLB entry caused by the instructions described in this section obey the sequential execution model with respect to the contents of the entry and with respect to data dependencies on those contents. That is, if an instruction sequence contains two or more of these instructions, when the sequence has completed, the final contents of the SLB entry and of General Purpose Registers is as if the instructions had been executed in program order.

However, software synchronization is required in order to ensure that any alterations of the entry take effect correctly with respect to address translation; see Chapter 12.

Programming Note

Changes to the segment mappings in the presence of active transactions may compromise transactional semantics if the transaction has accessed a segment that is assigned a new VSID. Consequently, when modifying segment mappings, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified segment is terminated, using a ***tabort.*** or ***treclaim.*** instruction.

SLB Invalidate Entry *X-form*

slbie RB

31	///	///	RB	434	/
0	6	11	16	21	31

```
ea0:35 ← (RB)0:35
if, for SLB entry that translates
  or most recently translated ea,
    entry_class = (RB)36 and
    entry_seg_size = size specified in (RB)37:38
then for SLB entry (if any) that translates ea
  SLBEV ← 0
  all other fields of SLBE ← undefined
else
   s ← log_base_2(entry_seg_size)
   esid ← (RB)0:63-s
   u ← undefined 1-bit value
   if u then
     if an SLB entry translates esid
        SLBEV ← 0
        all other fields of SLBE ← undefined
```

Let the Effective Address (EA) be any EA for which $EA_{0:35} = (RB)_{0:35}$. Let the class be $(RB)_{36}$. Let the segment size be equal to the segment size specified in $(RB)_{37:38}$; the allowed values of $(RB)_{37:38}$, and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 21 on page 897).

The class value and segment size must be the same as the class value and segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for a real mode address “translation” is invalidated. If the SLB contains more than one such entry, then zero or more such entries are invalidated, and similarly for any implementation-specific lookaside information used in address translation; additionally, a machine check may occur.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

The hardware ignores the contents of RB listed below and software must set them to 0s.

- $(RB)_{37}$
- $(RB)_{39}$
- $(RB)_{40:63}$
- If s = 40, $(RB)_{24:35}$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

slbie does not affect SLBs on other threads.

Programming Note

The reason the class value specified by ***slbie*** must be the same as the Class value that is or was in the relevant SLB entry is that the hardware may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by ***slbie*** differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some implementations maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.)

When switching tasks in certain cases, it may be advantageous to preserve some implementation-specific lookaside entries while invalidating others. The IH=0b001 invalidation hint of the ***slbia*** instruction can be used for this purpose if SLB class values are appropriately assigned, i.e. a class value of 0 gives the hint that the entry should be preserved and a class value of 1 indicates the entry must be invalidated. Also, it is advantageous to assign a class value of 1 to entries that need to be invalidated via an ***slbie*** instruction while preserving implementation-specific lookaside entries that are not derived from an SLB entry since such entries are assigned a class value of 0.

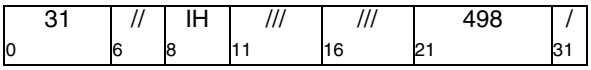
The *Move To Segment Register* instructions (see Section 5.9.3.2.1) create SLB entries in which the Class value is 0.

Programming Note

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

SLB Invalidate All *X-form*

slbia IH




```
for each SLB entry except SLB entry 0
   SLBE_V ← 0
   all other fields of SLBE ← undefined
```

For all SLB entries except SLB entry 0, the V bit in the entry is set to 0, making the entry invalid, and the remaining fields of the entry are set to undefined values. SLB entry 0 is not altered.

On implementations that have implementation-specific lookaside information for effective to real address translations, the IH field provides a hint that can be used to invalidate entries selectively in such lookaside information. The defined values for IH are as follows.

0b000 All such implementation-specific lookaside information is invalidated. (This value is not a hint.)

0b001 Preserve such implementation-specific lookaside information having a Class value of 0.

0b010 Preserve such implementation-specific lookaside information created when $MSR_{IR/DR}$=0.

0b110 Preserve such implementation-specific lookaside information created when MSR_{HV}=1, MSR_{PR}=0, and $MSR_{IR/DR}$=0.

All other IH values are reserved. If the IH field contains a reserved value, the hint provided by the IH field is undefined.

Implementation specific lookaside information for which preservation is not requested is invalidated. Implementation specific lookaside information for which preservation is requested may be invalidated.

When IH=0b000, execution of this instruction has the side effect of clearing the storage access history associated with the Hypervisor Real Mode Storage Control facility. See Section 5.7.3.3.1, “Hypervisor Real Mode Storage Control” for more details.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

slbia does not affect SLBs on other threads.

Programming Note

If ***slbia*** is executed when instruction address translation is enabled, software can ensure that attempting to fetch the instruction following the ***slbia*** does not cause an Instruction Segment interrupt by placing the ***slbia*** and the subsequent instruction in the effective segment mapped by SLB entry 0. (The preceding assumes that no other interrupts occur between executing the ***slbia*** and executing the subsequent instruction.)

If it is desired to invalidate the entire SLB and all associated implementation-specific lookaside information, the following sequence can be used. The sequence assumes that address translation is disabled.

```
li     r0,0
slbmte r0,r0    # clear SLBE 0
slbia  0b000    # invalidate all other
                # SLBEs, and all ERATEs
```

Programming Note

The defined values for IH are as follows.

0b000 All ERAT entries are invalidated. (This value is not a hint.) This value should be used by the hypervisor when relocating itself (i.e. when modifying the HRMOR) or when reconfiguring real storage.

0b001 Preserve ERAT entries with a Class value of 0. This value should be used by an operating system when switching tasks in certain cases; for example, if $SLBE_C$=0 is used for SLB translations shared between the tasks.

0b010 Preserve ERAT entries created when $MSR_{IR/DR}$=0. This value should generally be used by an operating system when switching tasks.

0b110 Preserve ERAT entries created when MSR_{HV}=1 and $MSR_{IR/DR}$=0. This value should be used by the hypervisor when switching partitions.

Programming Note

slbia serves as both a basic and an extended mnemonic. The Assembler will recognize an ***slbia*** mnemonic with one operand as the basic form, and an ***slbia*** mnemonic with no operand as the extended form. In the extended form the IH operand is omitted and assumed to be 0.

SLB Move To Entry *X-form*

slbmte RS,RB

31	RS	///	RB	402	/
0	6	11	16	21	31

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown in Figure 35.

RS

B	VSID	K_sK_pNLC	0	LP	0s
0	2	52	57	58	60 63

RB

ESID	V	0s	index
0	36	37 52	63

$RS_{0:1}$ B
$RS_{2:51}$ VSID
RS_{52} K_s
RS_{53} K_p
RS_{54} N
RS_{55} L
RS_{56} C
RS_{57} must be 0b0
$RS_{58:59}$ LP
$RS_{60:63}$ must be 0b0000
$RB_{0:35}$ ESID
RB_{36} V
$RB_{37:51}$ must be 0b000 || 0x000
$RB_{52:63}$ index, which selects the SLB entry

Figure 35. GPR contents for slbmte

On implementations that support a virtual address size of only n bits, n<78, $(RS)_{2:79-n}$ must be zeros.

$(RS)_{57}$ and $(RS)_{60:63}$ are ignored by the hardware.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0:15).

This instruction cannot be used to invalidate the translation contained in an SLB entry.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

The reason ***slbmte*** cannot be used to invalidate an SLB entry is that it does not necessarily affect implementation-specific address translation lookaside information. ***slbie*** (or ***slbia***) must be used for this purpose.

SLB Move From Entry VSID *X-form*

slbmfev RT,RB

31	RT	///	RB	851	/
0	6	11	16	21	31

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the B, VSID, K_s, K_p, N, L, C, and LP fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 36.

RT

B	VSID	K_sK_pNLC	0	LP	0s
0	2	52	57	58	60 63

RB

0s	index
0	52 63

$RT_{0:1}$	B
$RT_{2:51}$	VSID
RT_{52}	K_s
RT_{53}	K_p
RT_{54}	N
RT_{55}	L
RT_{56}	C
RT_{57}	set to 0b0
$RT_{58:59}$	LP
$RT_{60:63}$	set to 0b0000

$RB_{0:51}$	must be 0x0_0000_0000_0000
$RB_{52:63}$	index, which selects the SLB entry

Figure 36. GPR contents for slbmfev

On implementations that support a virtual address size of only n bits, n<78, $RT_{2:79-n}$ are set to zeros.

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

Special Registers Altered:

None

SLB Move From Entry ESID *X-form*

slbmfee RT,RB

31	RT	///	RB	915	/
0	6	11	16	21	31

If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the ESID and V fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 37.

RT

ESID	V	0s
0	36 37	63

RB

0s	index
0	52 63

$RT_{0:35}$ ESID
RT_{36} V
$RT_{37:63}$ set to 0b000 || 0x00_0000
$RB_{0:51}$ must be 0x0_0000_0000_0000
$RB_{52:63}$ index, which selects the SLB entry

Figure 37. GPR contents for slbmfee

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

Special Registers Altered:

None

SLB Find Entry ESID *X-form*

slbfee. RT,RB

31	RT	///	RB	979	1
0	6	11	16	21	31

The SLB is searched for an entry that matches the effective address specified by register RB. The search is performed as if it were being performed for purposes of address translation. That is, in order for a given entry to satisfy the search, the entry must be valid (V=1), and $(RB)_{0:63-s}$ must equal $SLBE[ESID_{0:63-s}]$ (where 2^s is the segment size selected by the B field in the entry).If exactly one matching entry is found, the contents of the B, VSID, K_s, K_p, N, L, C, and LP fields of the entry are placed into register RT. If no matching entry is found, register RT is set to 0. If more than one matching entry is found, either one of the matching entries is used, as if it were the only matching entry, or a Machine Check occurs. If a Machine Check occurs, register RT, and CR Field 0 are set to undefined values, and the description below of how this register and this field is set does not apply.

The contents of registers RT and RB are interpreted as shown in Figure 38.

RT

B	VSID	K_sK_pNLC	0	LP	0s
0	2	52	57	58	60 63

RB

ESID	0000	0s
0	36	40 63

$RT_{0:1}$ B
$RT_{2:51}$ VSID
RT_{52} K_s
RT_{53} K_p
RT_{54} N
RT_{55} L
RT_{56} C
RT_{57} set to 0b0
$RT_{58:59}$ LP
$RT_{60:63}$ set to 0b0000
$RB_{0:35}$ ESID
$RB_{36:39}$ must be 0b0000
$RB_{40:63}$ must be 0x000000

Figure 38. GPR contents for slbfee.

If s > 28, $RT_{80-s:51}$ are set to zeros. On implementations that support a virtual address size of only n bits, n < 78, $RT_{2:79-n}$ are set to zeros.

CR Field 0 is set as follows. j is a 1-bit value that is equal to 0b1 if a matching entry was found. Otherwise, j is 0b0.

$$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid j \mid\mid XER_{SO}$$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0-15).

This instruction is privileged.

Special Registers Altered:

CR0

5.9.3.2 Bridge to SLB Architecture [Category:Server.Phased-Out]

The facility described in this section can be used to ease the transition to the current Power ISA software-managed Segment Lookaside Buffer (SLB) architecture, from the Segment Register architecture provided by 32-bit PowerPC implementations. A complete description of the Segment Register architecture may be found in "Segmented Address Translation, 32-Bit Implementations," Section 4.5, Book III of Version 1.10 of the PowerPC architecture, referenced in the introduction to this architecture.

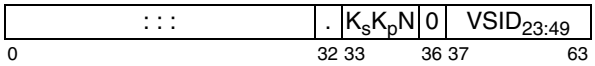
The facility permits the operating system to continue to use the 32-bit PowerPC implementation's *Segment Register Manipulation* instructions.

5.9.3.2.1 Segment Register Manipulation Instructions

The instructions described in this section -- ***mtsr***, ***mtsrin***, ***mfsr***, and ***mfsrin*** -- allow software to associate effective segments 0 through 15 with any of virtual segments 0 through 2^{27}-1. SLB entries 0:15 serve as virtual Segment Registers, with SLB entry i used to emulate Segment Register i. The ***mtsr*** and ***mtsrin*** instructions move 32 bits from a selected GPR to a selected SLB entry. The ***mfsr*** and ***mfsrin*** instructions move 32 bits from a selected SLB entry to a selected GPR.

The contents of the GPRs used by the instructions described in this section are shown in Figure 39. Fields shown as zeros must be zero for the *Move To Segment Register* instructions. Fields shown as hyphens are ignored. Fields shown as periods are ignored by the *Move To Segment Register* instructions and set to zero by the *Move From Segment Register* instructions. Fields shown as colons are ignored by the *Move To Segment Register* instructions and set to undefined values by the *Move From Segment Register* instructions.

RS/RT



RB

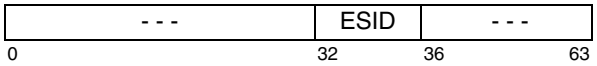


Figure 39. GPR contents for *mtsr*, *mtsrin*, *mfsr*, and *mfsrin*

Programming Note

The "Segment Register" format used by the instructions described in this section corresponds to the low-order 32 bits of RS and RT shown in the figure. This format is essentially the same as that for the Segment Registers of 32-bit PowerPC implementations. The only differences are the following.

- Bit 36 corresponds to a reserved bit in Segment Registers. Software must supply 0 for the bit because it corresponds to the L bit in SLB entries, and large pages are not supported for SLB entries created by the *Move To Segment Register* instructions.
- VSID bits 23:25 correspond to reserved bits in Segment Registers. Software can use these extra VSID bits to create VSIDs that are larger than those supported by the *Segment Register Manipulation* instructions of 32-bit PowerPC implementations.

Bit 32 of RS and RT corresponds to the T (direct-store) bit of early 32-bit PowerPC implementations. No corresponding bit exists in SLB entries.

Programming Note

The Programming Note in the introduction to Section 5.9.3.1 applies also to the *Segment Register Manipulation* instructions described in this section, and to any combination of the instructions described in the two sections, except as specified below for ***mfsr*** and ***mfsrin***.

The requirement that the SLB contain at most one entry that translates a given effective address (see Section 5.7.6.1) applies to SLB entries created by ***mtsr*** and ***mtsrin***. This requirement is satisfied naturally if only ***mtsr*** and ***mtsrin*** are used to create SLB entries for a given ESID, because for these instructions the association between SLB entries and ESID values is fixed (SLB entry i is used for ESID i). However, care must be taken if ***slbmte*** is also used to create SLB entries for the ESID, because for ***slbmte*** the association between SLB entries and ESID values is specified by software.

Move To Segment Register X-form

mtsr SR,RS

31	RS	/	SR	///	210	/
0	6	11	12	16	21	31

The SLB entry specified by SR is loaded from register RS, as follows.

SLBE Bit(s)	Set to	SLB Field(s)
0:31	0x0000_0000	$ESID_{0:31}$
32:35	SR	$ESID_{32:35}$
36	0b1	V
37:38	0b00	B
39:61	0b000||0x0_0000	$VSID_{0:22}$
62:88	$(RS)_{37:63}$	$VSID_{23:49}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C
94	0b0	reserved
95:96	0b00	LP

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:

None

Move To Segment Register Indirect X-form

mtsrin RS,RB

31	RS	///	RB	242	/
0	6	11	16	21	31

The SLB entry specified by $(RB)_{32:35}$ is loaded from register RS, as follows.

SLBE Bit(s)	Set to	SLB Field(s)
0:31	0x0000_0000	$ESID_{0:31}$
32:35	$(RB)_{32:35}$	$ESID_{32:35}$
36	0b1	V
37:38	0b00	B
39:61	0b000||0x0_0000	$VSID_{0:22}$
62:88	$(RS)_{37:63}$	$VSID_{23:49}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C
94	0b0	reserved
95:96	0b00	LP

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

Special Registers Altered:

None

Move From Segment Register X-form

mfsr RT,SR

31	RT	/	SR	///	595	/
0	6	11	12	16	21	31

The contents of the low-order 27 bits of the VSID field and the contents of the K_s, K_p, N, and L fields of the SLB entry specified by SR are placed into register RT as follows.

SLBE Bit(s)	Copied to	SLB Field(s)
62:88	$RT_{37:63}$	$VSID_{23:49}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). If the SLB entry is invalid (V=0), $RT_{33:63}$ are set to 0. Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:

None

Move From Segment Register Indirect X-form

mfsrin RT,RB

31	RT	///	RB	659	/
0	6	11	16	21	31

The contents of the low-order 27 bits of the VSID field and the contents of the K_s, K_p, N, and L fields of the SLB entry specified by $(RB)_{32:35}$ are placed into register RT as follows.

SLBE Bit(s)	Copied to	SLB Field(s)
62:88	$RT_{37:63}$	$VSID_{23:49}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID<16, V=1, VSID<2^{27}, L=0, and C=0). If the SLB entry is invalid (V=0), $RT_{33:63}$ are set to 0. Otherwise the contents of register RT are undefined.

This instruction is privileged.

Special Registers Altered:

None

5.9.3.3 TLB Management Instructions

Programming Note

Changes to the page table in the presence of active transactions may compromise transactional semantics if a page accessed by a translation is remapped within the lifetime of the transaction. Through the use of a ***tlbie*** instruction to the unmapped page, an operating system or hypervisor can ensure that any transaction that has touched the affected page is terminated.

Changes to local translation lookaside buffers, through the ***tlbia*** and ***tlbiel*** instructions have no effect on transactions. Consequently, if these instructions are used to invalidate TLB entries after the unmapping of a page, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified page is terminated, using a ***tabort.*** or ***treclaim*** instruction.

TLB Invalidate Entry ***X-form***

tlbie RB,RS

31	RS	///	RB	306	/
0	6	11	16	21	31

```
L ← (RB)63
if L = 0
  then
    base_pg_size = 4K
    actual_pg_size =
       page size specified in (RB)56:58
    i = 51
  else
    base_pg_size =
       base page size specified in (RB)44:51
    actual_pg_size =
       actual page size specified in (RB)44:51
    i = max(min(43,63-b),63-p)
b ← log_base_2(base_pg_size)
p ← log_base_2(actual_pg_size)
sg_size ← segment size specified in (RB)54:55
for each thread
  for each TLB entry
    if (entry_VA14:i+14 = (RB)0:i) &
       (entry_sg_size = sg_size) &
       (entry_base_pg_size = base_pg_size) &
       (entry_actual_pg_size = actual_pg_size) &
       ( ( TLBEs contain LPID &
           (TLBELPID = (RS)32:63) ) |
         ( TLBEs do not contain LPID &
           (LPIDRLPID = (RS)32:63) ) )
    then
      if ((L = 0)|(b ≥ 20))
        then
          TLB entry ← invalid
        else
          if (entry_VA58:77-b = (RB)56:75-b)
then
  TLB entry ← invalid
```

The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below, where L is $(RB)_{63}$.

RS:

0s	LPID
0	32 63

RB if L=0:

AVA	0s	B	AP	0s	L
0	52	54	56	59	63

RB if L=1:

AVA	LP	0s	B	AVAL	L
0	44	52	54	56	63

$RS_{32:63}$ contains an LPID value. The supported $(RS)_{32:63}$ values are the same as the LPID values supported in LPIDR. $RS_{0:31}$ must contain zeros and are ignored by the hardware.

If the L field in RB contains 0, the base page size is 4 KB and $RB_{56:58}$ (AP - Actual Page size field) must be set to the $SLBE_{L||LP}$ encoding for the page size corresponding to the actual page size specified by the PTE that was used to create the TLB entry to be invalidated. Thus, b is equal to 12 and p is equal to $\log_2$ (actual page size specified by $(RB)_{56:58}$). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between $(RB)_{44:51}$ (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_{LP} and the base page size and actual page size, except for the “r” bits (see Section 5.7.7.1 on page 900 and Figure 25 on page 901). Thus, b is equal to $\log_2$ (base page size specified by $(RB)_{44:51}$) and p is equal to $\log_2$ (actual page size specified by $(RB)_{44:51}$). Specifically, $(RB)_{44+c:51}$ must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the maximum of 0 and (20-p).
- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). $(RB)_{0:i}$ must contain bits 14:(i+14) of the virtual address translated by the TLB to be invalidated. If b>20, $RB_{64-b:43}$

may contain any value and are ignored by the hardware.

- If b<20, $(RB)_{56:75-b}$ must contain bits 58:77-b of the virtual address translated by the TLB to be invalidated, and other bits in $(RB)_{56:62}$ may contain any value and are ignored by the hardware.
- If b≥20, $(RB)_{56:62}$ (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in $(RB)_{54:55}$ (B field). The contents of $RB_{54:55}$ must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

$RB_{52:53}$ and $RB_{59:62}$ (when $(RB)_{63}$ = 0) must contain zeros and are ignored by the hardware.

All TLB entries on all threads that have all of the following properties are made invalid.

- The entry translates a virtual address for which all the following are true.
  - $VA_{14:14+i}$ is equal to $(RB)_{0:i}$.
  - L=0 or b≥20 or, if L=1 and b<20, $VA_{58:77-b}$ is equal to $(RB)_{56:75-b}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The L field in RB is 0, the base page size of the entry is 4 KB, and the actual page size of the entry matches the actual page size specified in $(RB)_{56:58}$.
  - The L field in RB is 1, the base page size of the entry matches the base page size specified in $(RB)_{44:51}$, and the actual page size of the entry matches the actual page size specified in $(RB)_{44:51}$.
- Either of the following is true:
  - The implementation's TLB entries contain LPID values and $TLBE_{LPID} = (RS)_{32:63}$.
  - The implementation's TLB entries do not contain LPID values, and $LPIDR_{LPID} = (RS)_{32:63}$. The LPIDR used for this comparison is in the same thread as the TLB entry being tested.

If the implementation's TLB entries contain LPID values, additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches $(RS)_{32:63}$. If the implementation's TLB entries do not contain LPID values, additional TLB entries may also be made invalid on any thread that is in the partition specified by $(RS)_{32:63}$.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

If the value specified in $RS_{32:63}$, $RB_{54:55}$, $RB_{56:58}$ (when RB_{63}=0), or $RB_{44:51}$ (when RB_{63}=1) is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the thread executing the ***tlbie*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a fourth set of operations, which is independent of the other three sets that ***eieio*** orders.

This instruction is hypervisor privileged.

See Section 5.10, "Page Table Update Synchronization Requirements" for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

For ***tlbie***[***l***] instructions in which $(RB)_{63}$=0, the AP value in RB is provided to make it easier for the hardware to locate address translations, in lookaside buffers, corresponding to the address translation being invalidated.

For ***tlbie***[***l***] instructions the AP specification is not binary compatible with versions of the architecture that precede Version 2.06. As an example, for an actual page size of 64 KB AP=0b101, whereas software written for an implementation that complies with a version of the architecture that precedes V. 2.06 would have AP=100 since AP was a 1 bit value followed by 0s in $RB_{57:58}$. If binary compatibility is important, for a 64 KB page software can use AP=0b101 on these earlier implementations since these implementations were required to ignore $RB_{57:58}$.

Programming Note

For ***tlbie***[***l***] instructions the AVA and AVAL fields in RB contain different VA bits from those in PTE_{AVA}.

TLB Invalidate Entry Local *X-form*

tlbiel RB

31	///	///	RB	274	/
0	6	11	16	21	31

```
IS ← (RB)52:53
switch(IS)
  case (0b00):
    L ← (RB)63
    if L = 0
      then
        base_pg_size = 4K
        actual_pg_size =
                    page size specified in (RB)56:58
        i = 51
      else
        base_pg_size =
               base page size specified in (RB)44:51
        actual_pg_size =
             actual page size specified in (RB)44:51
        i = max(min(43,63-b),63-p)
    b ← log_base_2(base_pg_size)
    p ← log_base_2(actual_pg_size)
    sg_size ← segment size specified in (RB)54:55
    for each TLB entry
      if (entry_VA14:i+14 = (RB)0:i) &
         (entry_sg_size = segment_size) &
         (entry_base_pg_size = base_pg_size) &
         (entry_actual_pg_size = actual_pg_size) &
         (TLBEs do not contain LPID |
         (TLBEs contain LPID & (TLBELPID=LPIDRLPID)))
        then
          if ((L = 0)|(b ≥ 20))
            then
              TLB entry ← invalid
            else
              if (entry_VA58:77-b = (RB)56:75-b)
                then
                  TLB entry ← invalid
  case (0b10):
    i ← implementation-dependent number, 40≤i≤51
    for each TLB entry in set (RB)i:51
      if (TLBEs do not contain LPID |
         (TLBEs contain LPID & (TLBELPID=LPIDRLPID)))
        then TLB entry ← invalid
  case (0b11):
    i ← implementation-dependent number, 40≤i≤51
    if MSRHV then
      TLBEs in set (RB)i:51 ← invalid
    else
      for each TLB entry in set (RB)i:51
        if (TLBEs do not contain LPID |
         (TLBEs contain LPID & (TLBELPID=LPIDRLPID)))
          then TLB entry ← invalid
```

The operation performed by this instruction is based on the contents of register RB. The contents of RB are shown below, where IS is $(RB)_{52:53}$ and L is $(RB)_{63}$.

IS=0b00 and L=0:

AVA	IS	B	AP	0s	L
0	52	54	56	59	63

IS=0b00 and L=1:

AVA	LP	IS	B	AVAL	L
0	44	52	54	56	63

IS=0b10 or 0b11:

0s	SET	IS	0s
0	40	52	54 63

The Invalidation Selector (IS) field in RB has three defined values (0b00, 0b10, and 0b11). The IS value of 0b01 is reserved and is treated in the same manner as the corresponding case for instruction fields (see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” on page 5 in Book I).

Engineering Note
IS field in RB contains 0b00

If the L field in RB contains 0, the base page size is 4 KB and $RB_{56:58}$ (AP - Actual Page size field) must be set to the $SLBE_{L||LP}$ encoding for the page size corresponding to the actual page size specified by the PTE that was used to create the TLB entry to be invalidated. Thus, b is equal to 12 and p is equal to $\log_2$ (actual page size specified by $(RB)_{56:58}$). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between $(RB)_{44:51}$ (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_{LP} and the base page size and actual page size, except for the “r” bits (see Section 5.7.7.1 on page 900 and Figure 25 on page 901). Thus, b is equal to $\log_2$ (base page size specified by $(RB)_{44:51}$) and p is equal to $\log_2$ (actual page size specified by $(RB)_{44:51}$). Specifically, $(RB)_{44+c:51}$ must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the maximum of 0 and (20-p).
- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). $(RB)_{0:i}$ must contain bits 14:(i+14) of the virtual address translated by the TLB to be invalidated. If b>20, $RB_{64-b:43}$ may contain any value and are ignored by the hardware.
- If b<20, $(RB)_{56:75-b}$ must contain bits 58:77-b of the virtual address translated by the TLB to be invalidated, and other bits in $(RB)_{56:62}$ may

contain any value and are ignored by the hardware.

- If b≥20, $(RB)_{56:62}$ (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in $(RB)_{54:55}$ (B field). The contents of $RB_{54:55}$ must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

Let the segment size be equal to the segment size specified in $(RB)_{54:55}$ (B field). The contents of $RB_{54:55}$ must be the same as the contents of PTE_B used to create the TLB entry to be invalidated.

All TLB entries that have all of the following properties are made invalid on the thread executing the ***tlbiel*** instruction.

- The entry translates a virtual address for which all the following are true.
  - $VA_{14:14+i}$ is equal to $(RB)_{0:i}$.
  - L=0 or b≥20 or, if L=1 and b<20, $VA_{58:77-b}$ is equal to $(RB)_{56:75-b}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The L field in RB is 0, the base page size of the entry is 4 KB, and the actual page size of the entry matches the actual page size specified in $(RB)_{56:58}$.
  - The L field in RB is 1, the base page size of the entry matches the base page size specified in $(RB)_{44:51}$, and the actual page size of the entry matches the actual page size specified in $(RB)_{44:51}$.
- Either of the following is true:
  - The implementation's TLB entries do not contain LPID values.
  - The implementation's TLB entries contain LPID values and $TLBE_{LPID}$ = $LPIDR_{LPID}$.

IS field in RB contains 0b10 or 0b11

$(RB)_{i:51}$ (bits i-40:11 of the SET field in (RB)) specify a set of TLB entries, where i is an implementation-dependent value in the range 40:51. Each entry in the set is invalidated if any of the following conditions are met for the entry.

- The implementation's TLB entries do not contain an LPID value.
- The IS field in RB contains 0b10 or MSR_{HV}=0, the implementation's TLB entries contain an LPID value, and $TLBE_{LPID}$ = $LPIDR_{LPID}$.
- The IS field in RB contains 0b11 and MSR_{HV}=1.

How the TLB is divided into the 2^{52-i} sets is implementation-dependent. The relationship of virtual addresses to these sets is also implementation-dependent. However, if, in an implementation, there can be multiple TLB entries for the same virtual address and same partition, then all these entries must be in a single set.

If the IS field in RB contains 0b10 or 0b11, it is implementation-dependent whether implementation-specific lookaside information that contains translations of effective addresses to real addresses is invalidated.

$RB_{0:39}$ (when $(RB)_{52:53}$ = 0b10 or 0b11), $RB_{59:62}$(when $(RB)_{52:53}$ = 0b00 and $(RB)_{63}$=0), and $RB_{54:63}$ (when $(RB)_{52:53}$ = 0b10 or 0b11) must contain 0s and are ignored by the hardware. When i>40 and $(RB)_{52:53}$ = 0b10 or 0b11, $RB_{40:i-1}$ may contain any value and are ignored by the hardware.

Only TLB entries on the thread executing the ***tlbiel*** instruction are affected.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are boundedly undefined.

If the value specified in $RB_{54:55}$, $RB_{56:58}$ (when $RB_{52:53}$=0b00 and $(RB)_{63}$ is 0), or $RB_{44:51}$ (when $RB_{52:53}$=0b00 and $(RB)_{63}$ is 1) is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

This instruction is privileged.

See Section 5.10, "Page Table Update Synchronization Requirements" on page 935 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

The primary use of this instruction by hypervisor software is to invalidate TLB entries prior to reassigning a thread to a new logical partition.

For IS = 0b10 or 0b11, it is implementation-dependent whether ERAT entries are invalidated. If the ***tlbiel*** instruction is being executed due to a partition swap, an ***slbia*** instruction can be used to invalidate the pertinent ERAT entries. If the ***tlbiel*** instruction is being executed to invalidate TLB entries with parity or ECC errors, the fact that the corresponding ERAT entries are not invalidated is immaterial. If the ***tlbiel*** instruction is being executed to invalidate multiple matching TLB entries, the fact that the corresponding ERAT entries are not invalidated is immaterial for implementations that never create multiple matching ERAT entries.

The primary use of this instruction by operating system software is to invalidate TLB entries that were created by the hypervisor using an implementation-specific hypervisor-managed TLB facility, if such a facility is provided.

tlbiel may be executed on a given thread even if the sequence ***tlbie*** - ***eieio*** - ***tlbsync*** - ***ptesync*** is concurrently being executed on another thread.

See also the Programming Notes with the description of the ***tlbie*** instruction.

TLB Invalidate All *X-form*

tlbia

31	///	///	///	370	/
0	6	11	16	21	31

```
all TLB entries ← invalid
```

All TLB entries are made invalid on the thread executing the ***tlbia*** instruction.

This instruction is hypervisor privileged.

This instruction is optional, and need not be implemented.

Special Registers Altered:

None

Programming Note

tlbia does not affect TLBs on other threads.

TLB Synchronize *X-form*

tlbsync

31	///	///	///	566	/
0	6	11	16	21	31

The ***tlbsync*** instruction provides an ordering function for the effects of all ***tlbie*** instructions executed by the thread executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***ptesync*** instruction executed by the same thread. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All TLB invalidations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction will have completed on any other thread before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that thread.
- All storage accesses by other threads for which the address was translated using the translations being invalidated, and all Reference and Change bit updates associated with address translations that were performed by other threads using the translations being invalidated, will have been performed with respect to the thread executing the ***ptesync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***ptesync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to preceding ***tlbie*** instructions executed by the thread executing the ***tlbsync*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a fourth set of operations, which is independent of the other three sets that ***eieio*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction have been performed.

This instruction is hypervisor privileged.

See Section 5.10 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

tlbsync should not be used to synchronize the completion of ***tlbiel***.

5.10 Page Table Update Synchronization Requirements

This section describes rules that software must follow when updating the Page Table, and includes suggested sequences of operations for some representative cases.

In the sequences of operations shown in the following subsections, the Page Table Entry is assumed to be for a virtual page for which the base page size is equal to the actual page size. If these page sizes are different, multiple ***tlbie*** instructions are needed, one for each PTE corresponding to the virtual page.

In the sequences of operations shown in the following subsections, any alteration of a Page Table Entry (PTE) that corresponds to a single line in the sequence is assumed to be done using a *Store* instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a store doubleword operation is done using two *Store Word* instructions).

Stores are not performed out-of-order, as described in Section 5.5, “Performing Operations Out-of-Order” on page 890. Moreover, address translations associated with instructions preceding the corresponding Store instructions are not performed again after the stores have been performed. (These address translations must have been performed before the store was determined to be required by the sequential execution model, because they might have caused an exception.) As a result, an update to a PTE need not be preceded by a context synchronizing operation.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the PTE are to be used for address translations associated with subsequent instructions.

As noted in the description of the *Synchronize* instruction in Section 4.4.3 of Book II, address translation associated with instructions which occur in program order subsequent to the *Synchronize* (and this includes the ***ptesync*** variant) may be performed prior to the completion of the *Synchronize*. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

Programming Note

In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the ***rfid*** or ***hrfid*** instruction that returns from the interrupt handler may provide the required context synchronization.

Page Table Entries must not be changed in a manner that causes an implicit branch.

5.10.1 Page Table Updates

TLBs are non-coherent caches of the HTAB. TLB entries must be invalidated explicitly with one of the *TLB Invalidate* instructions.

Unsynchronized lookups in the HTAB continue even while it is being modified. Any thread, including a thread on which software is modifying the HTAB, may look in the HTAB at any time in an attempt to translate a virtual address. When modifying a PTE, software must ensure that the PTE's V bit is 0 if the PTE is inconsistent (e.g., if the RPN field is not correct for the current AVA field).

Updates of Reference and Change bits by the hardware are not synchronized with the accesses that cause the updates. When modifying doubleword 1 of a PTE, software must take care to avoid overwriting a hardware update of these bits and to avoid having the value written by a *Store* instruction overwritten by a hardware update.

Software must execute ***tlbie*** and ***tlbsync*** instructions only as part of the following sequence, and must ensure that no other thread will execute a “conflicting instruction” while the instructions in the sequence are executing on the given thread. In addition to achieving the required system synchronization, the sequence will cause transactions that include accesses to the affected page(s) to fail.

tlbie instruction(s) specifying the same LPID operand value
eieio
tlbsync
ptesync

Let L be the LPID value specified by the above ***tlbie*** instruction(s). The “conflicting instructions” in this case are the following.

- a ***tlbie*** instruction that specifies an LPID value that matches the value L
- a ***tlbsync*** instruction that is part of a ***tlbie***-***eieio***-***tlbsync***-***ptesync*** sequence in which the ***tlbie*** instruction(s) specify an LPID value that matches the value L
- an ***mtspr*** instruction that modifies the LPIDR, if the modification has either of the following properties.
  - The old LPID value (i.e., the contents of the LPIDR just before the ***mtspr*** instruction is executed) is the value L
  - The new LPID value (i.e., the value specified by the ***mtspr*** instruction) is the value L

Other instructions (excluding ***mtspr*** instructions that modify the LPIDR as described above, and excluding

tlbie instructions except as shown) may be interleaved with the instruction sequence shown above, but the instructions in the sequence must appear in the order shown. On systems consisting of only a single-threaded processor, the ***eieio*** and ***tlbsync*** instructions can be omitted.

Programming Note

The ***eieio*** instruction prevents the reordering of the preceding ***tlbie*** instructions with respect to the subsequent ***tlbsync*** instruction. The ***tlbsync*** instruction and the subsequent ***ptesync*** instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated (by the ***tlbie*** instructions), and all Reference and Change bit updates associated with address translations that were performed using the translations being invalidated, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ptesync instruction are performed with respect to that thread or mechanism.

For Page Table update sequences that mark the PTE invalid (see Section 5.10.1.2, "Modifying a Page Table Entry"), Reference and Change bit updates can continue to be performed in the invalid PTE until the ***ptesync*** at the end of the ***tlbie***/***eieio***/***tlbsync***/***ptesync*** sequence has completed. Any access to the PTE, by software, that should be performed after all such implicit PTE updates have completed, such as reading the final values of the Reference and Change bits or modifying PTE bytes that contain those bits, must be placed after this ***ptesync***.

Before permitting an ***mtspr*** instruction that modifies the LPIDR to be executed on a given thread, software must ensure that no other thread will execute a "conflicting instruction" until after the ***mtspr*** instruction followed by a context synchronizing instruction have been executed on the given thread (a context synchronizing event can be used instead of the context synchronizing instruction; see Chapter 12).

The "conflicting instructions" in this case are the following.

- a ***tlbie*** instruction specifying an LPID operand value that matches either the old or the new $LPIDR_{LPID}$ value
- a ***tlbsync*** instruction that is part of a ***tlbie***-***eieio***-***tlbsync***-***ptesync*** sequence in which the ***tlbie*** instruction(s) specify an LPID value that matches either the old or the new $LPIDR_{LPID}$ value

Programming Note

The restrictions specified above regarding modifying the LPIDR apply even on systems consisting of only a single-threaded processor, and even if the new LPID value is equal to the old LPID value.

The sequences of operations shown in the following subsections assume a multi-threaded environment. In an environment consisting of only a single-threaded processor, the ***tlbsync*** must be omitted, and the ***eieio*** that separates the ***tlbie*** from the ***tlbsync*** can be omitted. In a multi-threaded environment, when ***tlbiel*** is used instead of ***tlbie*** in a Page Table update, the synchronization requirements are the same as when ***tlbie*** is used in an environment consisting of only a single-threaded processor.

Programming Note

For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another thread, the ***ptesync*** instruction at the end of the sequence should be followed by a *Store* instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the ***ptesync*** instruction ensures that if a *Load* instruction executed by another thread returns the chosen value from location X, the sequence's stores to the Page Table have been performed with respect to that other thread. The *Load* instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE.)

This Note assumes that the Page Table and location X are in storage that is Memory Coherence Required.

5.10.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The V bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes

```
PTE_ARPN,LP,AC,R,C,WIMG,N,PP ← new values
eieio     /* order 1st update before 2nd */
PTE_B,AVA,SW,L,H,V ← new values (V=1)
ptesync   /* order updates before next
             Page Table search and before
```

```
            next data access                */
```

5.10.1.2 Modifying a Page Table Entry

General Case

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. (The sequence is equivalent to deleting the PTE and then adding a new one; see Sections 5.10.1.1 and 5.10.1.3.)

```
PTE_V ← 0  /* (other fields don't matter)*/
ptesync    /* order update before tlbie and
              before next Page Table search */
tlbie(old_B,old_VA_14:77-b,old_L,old_LP,old_AP,
         old_LPID)
         /*invalidate old translation*/
eieio      /* order tlbie before tlbsync     */
tlbsync    /* order tlbie before ptesync     */
ptesync    /* order tlbie, tlbsync and 1st
              update before 2nd update       */
PTE_ARPN,LP,AC,R,C,WIMG,N,PP ← new values
eieio      /* order 2nd update before 3rd */
PTE_B,AVA,SW,L,H,V ← new values (V=1)
ptesync    /* order 2nd and 3rd updates before
              next Page Table search and
              before next data access       */
```

Resetting the Reference Bit

If the only change being made to a valid entry is to set the Reference bit to 0, a simpler sequence suffices because the Reference bit need not be maintained exactly.

```
oldR ← PTE_R /* get old R                          */
if oldR = 1 then
  PTE_R ← 0        /* store byte (R=0, other bits
                      unchanged)                  */
  tlbie(B,VA_14:77-b,L,LP,AP,LPID) /* invalidate
                                      entry      */
  eieio    /* order tlbie before tlbsync           */
  tlbsync  /* order tlbie before ptesync           */
  ptesync  /* order tlbie, tlbsync, and update
              before next Page Table search
              and before next data access          */
```

Modifying the SW field

If the only change being made to a valid entry is to modify the SW field, the following sequence suffices, because the SW field is not used by the hardware and doubleword 0 of the PTE is not modified by the hardware.

```
loop: ldarx r1 ← PTE_dwd_0 /* load dwd 0 of PTE */
      r1_57:60 ← new SW value /* replace SW, in r1 */
      stdcx. PTE_dwd_0 ←  r1 /* store dwd 0 of PTE
             if still reserved (new SW value, other
             fields unchanged) */
      bne-   loop    /* loop if lost reservation */
```

A ***lbarx/stbcx.***, ***lharx***/***sthcx.***, or ***lwarx***/***stwcx.*** pair (specifying the low-order byte, halfword, or word respectively of doubleword 0 of the PTE) can be used instead of the ***ldarx*** /***stdcx.*** pair shown above.

Modifying the Virtual Address

If the virtual address translated by a valid PTE is to be modified and the new virtual address hashes to the same PTEG (or the same two PTEGs if the secondary Page Table search is enabled) as does the old virtual address, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE_AVA,SW,L,H,V ← new values (V=1)
ptesync    /* order update before tlbie and
              before next Page Table search       */
tlbie(old_B,old_VA_14:77-b,old_L,old_LP,old_AP,
         old_LPID) /*invalidate old translation*/
eieio      /* order tlbie before tlbsync          */
tlbsync    /* order tlbie before ptesync          */
ptesync    /* order tlbie, tlbsync, and update
              before next data access             */
```

5.10.1.3 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
PTE_V ← 0  /* (other fields don't matter)         */
ptesync    /* order update before tlbie and
              before next Page Table search  */
tlbie(old_B,old_VA_14:77-b,old_L,old_LP,old_AP,
         old_LPID)  /*invalidate old translation*/
eieio      /* order tlbie before tlbsync          */
tlbsync    /* order tlbie before ptesync          */
ptesync    /* order tlbie, tlbsync, and update
              before next data access             */
```

Chapter 6. Interrupts

6.1 Overview

The Power ISA provides an interrupt mechanism to allow the thread to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

6.2 Interrupt Registers

6.2.1 Machine Status Save/ Restore Registers

When various interrupts occur, the state of the machine is saved in the Machine Status Save/Restore registers (SRR0 and SRR1). Section 6.5 describes which registers are altered by each interrupt.

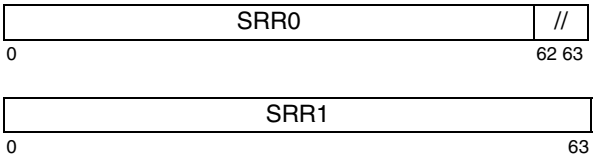


Figure 40. Save/Restore Registers

SRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for SRR1 bits in the range 33:36, 42:43, and 45:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set SRR1 (including implementation-dependent setting, e.g. by the Machine Check interrupt or by implementation-specific interrupts). $SRR1_{44}$ cannot be treated as reserved, regardless of how it is set by interrupts, because it is used by software, as described in a Programming Note near the end of Section 6.5.9, “Program Interrupt” on page 959.

6.2.2 Hypervisor Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Hypervisor Machine Status Save/ Restore registers (HSRR0 and HSRR1). Section 6.5 describes which registers are altered by each interrupt.

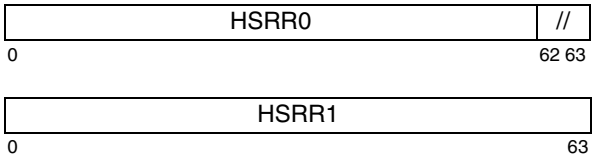


Figure 41. Hypervisor Save/Restore Registers

HSRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for HSRR1 bits in the range 33:36 and 42:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set HSRR1 (including implementation-dependent setting, e.g. by implementation-specific interrupts).

The HSRR0 and HSRR1 are hypervisor resources; see Chapter 2.

Programming Note

Execution of some instructions, and fetching instructions when MSR_{IR}=1, may have the side effect of modifying HSRR0 and HSRR1; see Section 6.4.4.

6.2.3 Data Address Register

The Data Address Register (DAR) is a 64-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 6.5.2, 6.5.3, 6.5.4, and 6.5.8. In general, when one of these interrupts occurs the DAR is set to an effective address associated with the storage access that caused the

interrupt, with the high-order 32 bits of the DAR set to 0 if the interrupt occurs in 32-bit mode.

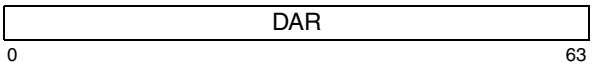


Figure 42. Data Address Register

6.2.4 Hypervisor Data Address Register

The Hypervisor Data Address Register (HDAR) is a 64-bit register that is set by the Hypervisor Data Storage Interrupt; see Section 6.5.16. In general, when this interrupt occurs, the HDAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the HDAR set to 0 if the interrupt occurs in 32-bit mode.

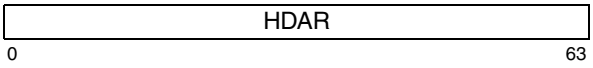


Figure 43. Hypervisor Data Address Register

6.2.5 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 6.5.2, 6.5.3, 6.5.4, and 6.5.8. In general, when one of these interrupts occurs the DSISR is set to indicate the cause of the interrupt.

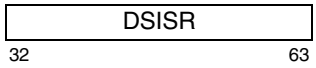


Figure 44. Data Storage Interrupt Status Register

DSISR bits may be treated as reserved in a given implementation if they are specified as being set either to 0 or to an undefined value for all interrupts that set the DSISR (including implementation-dependent setting, e.g. by the Machine Check interrupt or by implementation-specific interrupts).

6.2.6 Hypervisor Data Storage Interrupt Status Register

The Hypervisor Data Storage Interrupt Status Register (HDSISR) is a 32-bit register that is set by the Hypervisor Data Storage interrupt. In general, when one of these interrupts occurs the HDSISR is set to indicate the cause of the interrupt.

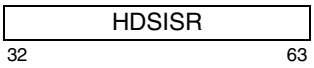


Figure 45. Hypervisor Data Storage Interrupt Status Register

6.2.7 Hypervisor Emulation Instruction Register

The Hypervisor Emulation Instruction Register (HEIR) is a 32-bit register that is set by the Hypervisor Emulation Assistance interrupt; see Section 6.5.18. The image of the instruction that caused the interrupt is loaded into the register.

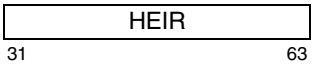


Figure 46. Hypervisor Emulation Instruction Register

6.2.8 Hypervisor Maintenance Exception Register

Each bit in the Hypervisor Maintenance Exception Register (HMER) is associated with one or more causes of the Hypervisor Maintenance exception, and is set when the associated exception(s) occur. If the corresponding bit in the Hypervisor Maintenance Exception Enable Register (HMEER) is set, a Hypervisor Maintenance Interrupt (HMI) may occur. If the thread is in a power-saving mode when the interrupt would have occurred, the thread will exit the power-saving mode; see Section 6.5.19 and Section 3.3.2.

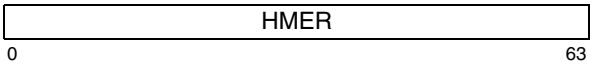


Figure 47. Hypervisor Maintenance Exception Register

The contents of the HMER are as follows:

- **0** Set to 1 for a Malfunction Alert.
- **1** Set to 1 when performance is degraded for thermal reasons.
- **2** Set to 1 when thread recovery is invoked.
- **Others** Implementation-specific.

When the ***mtspr*** instruction is executed with the HMER as the encoded Special Purpose Register, the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mthmer*** instruction.

Programming Note

An access to the HMER is likely to be very slow. Software should access it sparingly.

6.2.9 Hypervisor Maintenance Exception Enable Register

The Hypervisor Maintenance Exception Enable Register (HMEER) is a 64-bit register in which each bit enables the corresponding exception in the HMER to cause the Hypervisor Maintenance interrupt, potentially causing exit from power-saving mode; see Section 6.5.19 and Section 3.3.2.

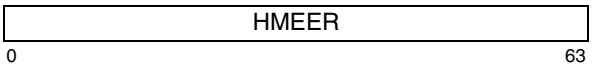


Figure 48. Hypervisor Maintenance Exception Enable Register

6.2.10 Facility Status and Control Register

The Facility Status and Control Register (FSCR) controls the availability of various facilities in problem state and indicates the cause of a Facility Unavailable interrupt.

When the FSCR makes a facility unavailable, attempted usage of the facility in problem state is treated as follows:

- Execution of an instruction causes a Facility Unavailable interrupt.
- Access of an SPR using ***mfspr/mtspr*** causes a Facility Unavailable interrupt
- ***rfebb***, ***rfid***, ***rfscv***, ***hrfid*** and ***mtmsr[d]*** instructions have the same effect on bits in system registers as they would if the bits were available.

Programming Note

The FSCR does not prevent ***rfebb*** instructions from attempting to set bits in System Registers that the FSCR makes unavailable. Thus changes to $BESCR_{TS}$ made by the operating system have the potential to result in an illegal transaction state transition when ***rfebb*** is subsequently executed in problem state, resulting in the occurrence of a TM Bad Thing type Program interrupt.

The MSR can also make the Transactional Memory facility unavailable in any privilege state, and $MMCR0_{PMCC}$ can make various components of the Performance Monitor unavailable when accessed problem state. An access to one of these facilities when it is unavailable causes a Facility Unavailable interrupt.

When the PCR makes a facility unavailable in problem state, the facility is treated as not implemented in problem state; any Facility Unavailable interrupt that would occur if the facility were not made unavailble by the PCR does not occur.

When a Facility Unavailable interrupt occurs, the unavailable facility that was accessed is indicated in the most-significant byte of the FSCR.

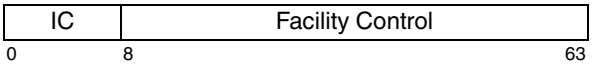


Figure 49. Facility Status and Control Register

The contents of the FSCR are specified below.

Value **Meaning**

0:7 **Interruption Cause** (IC)

When a Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the facility for which access was attempted. The values and their meanings are specified below.

02 Access to the DSCR at SPR 3
03 Access to a Performance Monitor SPR in group A or B when $MMCR0_{PMCC}$ is set to a value for which the access results in a Facility Unavailable interrupt (See the definition of $MMCR0_{PMCC}$ in Section 9.4.4.)
04 Execution of a BHRB Instruction
05 Access to a Transactional Memory SPR or execution of a Transactional Memory Instruction
06 Reserved
07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch Instruction
08 Access to the Target Address Register

All other values are reserved.

8:63 **Facility Enable** (FE)

The FE field controls the availability of various facilities in problem state as specified below.

8:54 Reserved

55 **Target Address Register** (TAR)

0 The TAR and ***bctar*** instruction are not available in problem state.
1 The TAR and ***bctar*** instruction are available in problem state unless made unavailable by another register.

56 **Event-Based Branch Facility** (EBB)

0 The Event-Based Branch facility SPRs and instructions are not available in problem state, and event-based exceptions and branches do not occur.
1 The Event-Based Branch facility SPRs and instructions are available in problem state unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other registers.

57 Reserved

58:60 Reserved

Programming Note

$HFSCR_{58:60}$ are used to control the availability of Transactional Memory, the Performance Monitor, and the BHRB in problem and privileged states. $FSCR_{58:60}$ are reserved since the availability of Transactional Memory is controlled by the MSR, and the availability of the Performance Monitor and BHRB is controlled by MMCR0.

61 **Data Stream Control Register at SPR 3** (DSCR)

0 SPR 3 is not available in problem state.
1 SPR 3 is available in problem state unless made unavailable by another register.

62:63 Reserved

Programming Note

When an OS has set the FSCR such that a facility is unavailable, the OS should either emulate the facility when it is accessed or provide an application interface that requires the application to request use of the facility before it accesses the facility.

6.2.11 Hypervisor Facility Status and Control Register

The Hypervisor Facility Status and Control Register (HFSCR) controls the available of various facilities in problem and privileged states, and indicates the cause of a Hypervisor Facility Unavailable interrupt.

When the HFSCR makes a facility unavailable, attempted usage of the facility in problem or privileged states is treated as follows:

- Execution of an instruction causes a Hypervisor Facility Unavailable interrupt.
- Access of an SPR using ***mfspr/mtspr*** causes a Hypervisor Facility Unavailable interrupt
- ***rfebb***, ***rfid***, ***rfscv***, ***hrfid*** and ***mtmsr[d]*** instructions have the same effect on bits in system registers as they would if the bits were available.

Programming Note

Because the HFSCR does not prevent ***mtspr***, ***rfscv***, ***[h]rfid***, and ***mtmsr[d]*** instructions from setting bits in system registers that the HFSCR will make unavailable after a transition to a lower privilege state, these instructions may cause interrupts in a variety of unexpected ways. For example, consider a hypervisor that sets HSRR1 such that ***hrfid*** returns to a lower privilege state with MSR[TS] nonzero. A TM Bad Thing interrupt will result, despite that TM is made unavailable by the HFSCR.

Similarly, the HFSCR does not prevent ***rfebb*** instructions from attempting to set bits in System Registers that the HFSCR makes unavailable. Thus changes to $BESCR_{TS}$ made by the hypervisor have the potential to result in an illegal transaction state transition when ***rfebb*** is subsequently executed in problem or privileged state, resulting in the occurrence of a TM Bad Thing type Program interrupt.

When the PCR makes a facility unavailable in problem state, the facility is treated as not implemented in problem state; any Hypervisor Facility Unavailable interrupt that would occur if the facility were not made unavailble by the PCR does not occur as a result of problem-state access. See Section 2.6 for additional information.)

When a Hypervisor Facility Unavailable interrupt occurs, the facility that was accessed is indicated in the most-significant byte of the HFSCR.

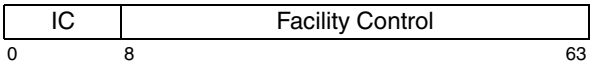


Figure 50. Hypervisor Facility Status and Control Register

The contents of the HFSCR are specified below.

Value **Meaning**

0:7 **Interruption Cause** (IC)

When a Hypervisor Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the access that was attempted. The values and their meanings are specified below.
00 Access to a Floating Point register or execution of a Floating Point instruction
01 Access to a Vector or VSX register or execution of a Vector or VSX instruction
02 Access to the DSCR at SPRs 3 or 17
03 Read or write access of a Performance Monitor SPR in group A, or read access of a Performance Monitor SPR in group B

(See the definition of $MMCR0_{PMCC}$ in Section 9.4.4 for a definition of groups A and B.)

04 Execution of a BHRB Instruction

05 Access to a Transactional Memory SPR or execution of a Transactional Memory instruction

06 Reserved

07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch Instruction

08 Access to the Target Address Register

All other values are reserved.

8:63 **Facility Enable** (FE)

The FE field controls the availability of various facilities in problem and privileged states as specified below.

8:54 Reserved

55 **Target Address Register** (TAR)

0 The TAR and ***bctar*** instruction are not available in problem and privileged state.

1 The TAR and ***bctar*** instruction are available in problem and privileged state unless made unavailable by another register.

56 **Event-Based Branch Facility** (EBB)

0 The Event-Based Branch facility SPRs and instructions are not available in problem and privileged states, and event-based exceptions and branches do not occur.

1 The Event-Based Branch facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other bits.

58 **Transactional Memory Facility** (TM)

0 The Transactional Memory Facility SPRs and instructions are not available in problem and privileged states.

1 The Transactional Memory Facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register.

59 **BHRB Instructions** (BHRB)

0 The BHRB instructions (***mfbhrb***, ***clrbhrb***) are not available in problem and privileged states.

1 The BHRB instructions (***mfbhrb***, ***clrbhrb***) are available in problem and privileged states unless made unavailable by another register.

60 **Performance Monitor Facility SPRs** (PM)

0 Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are not available in problem and privileged states; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are not available in privileged state. (See the definition of $MMCR0_{PMCC}$ in Section 9.4.4 for a definition of groups A and B.)

1 Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are available in problem and privileged states unless made unavailable by another register; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are available in privileged state.

61 **Data Stream Control Register** (DSCR)

0 SPR 3 is not available in problem or privileged states and SPR 17 is not available in privileged state.

1 SPR 3 is available in problem and privileged states and SPR 17 is available in privileged state unless made unavailable by another register.

62 **Vector and VSX Facilities** (VECVSX)

0 The facilities whose availability is controled by either MSR_{VEC} or MSR_{VSX} are not available in problem and privileged states.

1 The acilities whose availability is controled by either MSR_{VEC} or MSR_{VSX} are available in problem and privileged states unless made unavailable by another register.

63 **Floating Point Facility** (FP)

0 The facilities whose availability is controlled by MSR_{FP} are not available in problem and privileged states.

1 The facilities whose availability is controlled by MSR_{FP} are available in problem and privileged states unless made unavailable by another register.

Programming Note

The FSCR can be used to determine whether a particular facility is being used by an application, and the HFSCR can be used to determine whether a particular facility is being used by either an application or by an operating system. This is done by disabling the facility initially, and enabling it in the interrupt handler upon first usage. The information about the usage of a particular facility can be used to determine whether that facility's state must be saved and restored when changing program context.

Programming Note

The following tables summarize the interrupts that occur as a result of accessing the non-privileged Performance Monitor registers in problem state when $MMCR0_{PMCC}$, PCR, and HFSCR are set to various values. (Accesses to privileged Performance Monitor SPRs (SPRs 784-792, 795-798) in problem state result in Privileged Instruction Type Program interrupts.)

			mfspr				***mtspr***			
			PMCC				**PMCC**			
	SPR	**#**	**00**	**01**	**10**	**11**	**00**	**01**	**10**	**11**
Group A	MMCR2[6]	769	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	MMCRA	770	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC1	771	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC2	772	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC3	773	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC4	774	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC5	775	HU[4]	FU, HU[4]	HU[4]	FU, HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	FU, HU[4]
	PMC6	776	HU[4]	FU, HU[4]	HU[4]	FU, HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	FU, HU[4]
	MMCR0	779	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
Group B	SIER[6]	768	HU[4]	FU, HU[4]	HU[4]	HU[4]	See 2.	See 2.	See 2.	See 2.
	SIAR	780	HU[4]	FU, HU[4]	HU[4]	HU[4]	See 2.	See 2.	See 2.	See 2.
	SDAR	781	HU[4]	FU, HU[4]	HU[4]	HU[4]	See 2.	See 2.	See 2.	See 2.
	MMCR1	782	HU[4]	FU, HU[4]	FU, HU[4]	FU, HU[4]	See 2.	See 2.	See 2.	See 2.

Notes:

4. Terminology:
   FU: Facility Unavailable interrupt
   HE: Hypervisor Emulation Assistance interrupt
   HU: Hypervisor Facility Unavailable interrupt
5. Write operations to this SPR are not implemented for all privilege states. (See the ***mtspr*** instruction description in Section 4.4.4 for additional information.) FU or HU interrupts do not occur regardless of the value of $MMCR0_{PMCC}$ or $HFSCR_{PM}$.
6. When the PCR indicates a version of the architecture prior to V 2.07, this SPR is treated as not implemented in problem state; no FU or HU interrupts occur regardless of the value of $MMCR0_{PMCC}$ or $HFSCR_{PM}$.
7. An HU interrupt occurs if $HFSCR_{PM}$=0 when this register is accessed in either problem state or privileged (but not hypervisor) state.

6.3 Interrupt Synchronization

When an interrupt occurs, SRR0 or HSRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0 or HSRR0 may or may not have completed execution, depending on the interrupt type.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.5.1. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a non-recoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.5.1, but does satisfy items 4 and 5.

6.4 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are:

- System Reset
- Machine Check
- External
- Decrementer
- Directed Privileged Doorbell
- Hypervisor Decrementer
- Hypervisor Maintenance
- Directed Hypervisor Doorbell
- Performance Monitor

External, Decrementer, Hypervisor Decrementer, Directed Privileged Doorbell, Directed Hypervisor Doorbell, and Hypervisor Maintenance interrupts are maskable interrupts. Therefore, software may delay the generation of these interrupts. System Reset and Machine Check interrupts are not maskable.

"Instruction-caused" interrupts are further divided into two classes, *precise* and *imprecise*.

6.4.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise.

When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.
2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing thread.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type.
4. Architecturally, no subsequent instruction has begun execution.

6.4.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When an Imprecise Mode Floating-Point Enabled Exception type Program interrupt occurs, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or some instruction following that instruction; see Section 6.5.9, "Program Interrupt" on page 959.
2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing thread.
3. The instruction addressed by SRR0 may appear not to have begun execution (except, in some cases, for causing the interrupt to occur), may have been partially executed, or may have completed; see Section 6.5.9.
4. No instruction following the instruction addressed by SRR0 appears to have begun execution.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of all other maskable interrupts is controlled by either the operating system or the hypervisor.

6.4.3 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the thread's state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check interrupts (see Section 6.5.2) differs from the description given below in several respects.

1. SRR0 or HSRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 33:36 and 42:47 of SRR1 or HSRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1 or HSRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 51 on page 951. In particular, MSR bits IR and DR are set as specified by $LPCR_{AIL}$ (see Section 2.2), and MSR bit SF is set to 1, selecting 64-bit mode. The new values take effect beginning with the first instruction executed following the interrupt.
5. Instruction fetch and execution resumes, using the new MSR value, at the effective address specific to the interrupt type. These effective addresses are shown in Figure 52 on page 952. An offset may be applied to get the effective addresses, as specified by $LPCR_{AIL}$ (see Section 2.2).

Interrupts do not clear reservations obtained with ***lbarx***, ***lharx***, ***lwarx, ldarx***, or ***lqarx***.

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the operating system before dispatching a "new" program.

- ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** to clear the reservation if one is outstanding, to ensure that a ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, or ***lqarx*** in the interrupted program is not paired with a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** on the "new" program.
- ***sync***, to ensure that all storage accesses caused by the interrupted program will be performed with respect to another thread before the program is resumed on that other thread.
- ***isync*** or ***rfid***, to ensure that the instructions in the "new" program execute in the "new" context.
- ***treclaim***, to ensure that any previous use of the transactional facility is terminated.

Programming Note

For instruction-caused interrupts, in some cases it may be desirable for the operating system to emulate the instruction that caused the interrupt, while in other cases it may be desirable for the operating system not to emulate the instruction. The following list, while not complete, illustrates criteria by which decisions regarding emulation should be made. The list applies to general execution environments; it does not necessarily apply to special environments such as program debugging, bring-up, etc.

In general, the instruction should be emulated if:

- The interrupt is caused by a condition for which the instruction description (including related material such as the introduction to the section describing the instruction) implies that the instruction works correctly. Example: Alignment interrupt caused by ***lmw*** for which the storage operand is not aligned, or by ***dcbz*** for which the storage operand is in storage that is Write Through Required or Caching Inhibited.
- The instruction is an illegal instruction that should appear, to the program executing it, as if it were supported by the implementation. Example: A Hypervisor Emulation Assistance interrupt is caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports, or by an instruction that is in a category that the implementation does not support but is used by some programs that the operating system supports.

In general, the instruction should not be emulated if:

- The purpose of the instruction is to cause an interrupt. Example: System Call interrupt caused by ***sc***.
- The interrupt is caused by a condition that is stated, in the instruction description, potentially to cause the interrupt. Example: Alignment interrupt caused by ***lwarx*** for which the storage operand is not aligned.
- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should be permitted to access but for which there currently is no PTE that satisfies the Page Table search.)

Programming Note

If a program modifies an instruction that it or another program will subsequently execute and the execution of the instruction causes an interrupt, the state of storage and the content of some registers may appear to be inconsistent to the interrupt handler program. For example, this could be the result of one program executing an instruction that causes a Hypervisor Emulation Assistance interrupt just before another instance of the same program stores an *Add Immediate* instruction in that storage location. To the interrupt handler code, it would appear that a hardware generated the interrupt as the result of executing a valid instruction.

Programming Note

In order to handle Machine Check and System Reset interrupts correctly, the operating system should manage MSR_{RI} as follows.

- In the Machine Check and System Reset interrupt handlers, interpret SRR1 bit 62 (where MSR_{RI} is placed) as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable
- In each interrupt handler, when enough state has been saved that a Machine Check or System Reset interrupt can be recovered from, set MSR_{RI} to 1.
- In each interrupt handler, do the following (in order) just before returning.
  1. Set MSR_{RI} to 0.
  2. Set SRR0 and SRR1 to the values to be used by ***rfid***. The new value of SRR1 should have bit 62 set to 1 (which will happen naturally if SRR1 is restored to the value saved there by the interrupt, because the interrupt handler will not be executing this sequence unless the interrupt is recoverable).
  3. Execute ***rfid***.

For interrupts that set the SRRs other than Machine Check or System Reset, MSR_{RI} can be managed similarly when these interrupts occur within interrupt handlers for other interrupts that set the SRRs.

This Note does not apply to interrupts that set the HSRRs because these interrupts put the thread into hypervisor state, and either do not occur or can be prevented from occurring within interrupt handlers for other interrupts that set the HSRRs.

6.4.4 Implicit alteration of HSRR0 and HSRR1

Executing some of the more complex instructions may have the side effect of altering the contents of HSRR0 and HSRR1. The instructions listed below are guaranteed not to have this side effect. Any omission of instruction suffixes is significant; e.g., ***add*** is listed but ***add.*** is excluded.

1. *Branch* instructions

   b[***l***][***a***], ***bc***[***l***][***a***], ***bclr***[***l***], ***bcctr***[***l***]

2. *Fixed-Point Load* and *Store* Instructions

   lbz, ***lbzx***, ***lhz***, ***lhzx***, ***lwz***, ***lwzx***, ***ld*<64>**, ***ldx*<64>**, ***stb***, ***stbx***, ***sth***, ***sthx***, ***stw***, ***stwx***, ***std*<64>**, ***stdx*<64>**

   Execution of these instructions is guaranteed not to have the side effect of altering HSRR0 and HSRR1 only if the storage operand is aligned and MSR_{DR}=0.

3. *Arithmetic* instructions

   addi, ***addis***, ***add***, ***subf***, ***neg***

4. *Compare* instructions

   cmpi, ***cmp***, ***cmpli***, ***cmpl***

5. *Logical* and *Extend Sign* instructions

   ori, ***oris***, ***xori***, ***xoris***, ***and***, ***or***, ***xor***, ***nand***, ***nor***, ***eqv***, ***andc***, ***orc, extsb***, ***extsh***, ***extsw***

6. *Rotate and Shift* instructions

   rldicl*<64>**, ***rldicr*<64>**, ***rldic*<64>**, ***rlwinm, ***rldcl*<64>**, ***rldcr*<64>**, ***rlwnm***, ***rldimi*<64>**, ***rlwimi***, ***sld*<64>**, ***slw***, ***srd*<64>**, ***srw***

7. Other instructions

   isync

   rfid, ***hrfid***

   mtspr, ***mfspr***, ***mtmsrd***, ***mfmsr***

Programming Note

Instructions excluded from the list include the following.

- instructions that set or use XER_{CA}
- instructions that set XER_{OV} or XER_{SO}
- ***andi.***, ***andis.***, and fixed-point instructions with Rc=1 (Fixed-point instructions with Rc=1 can be replaced by the corresponding instruction with Rc=0 followed by a *Compare* instruction.)
- all floating-point instructions
- ***mftb***

These instructions, and the other excluded instructions, may be implemented with the assistance of the Hypervisor Emulation Assistance interrupt, or of implementation-specific interrupts that modify HSRR0 and HSRR1. The included instructions are guaranteed not to be implemented thus. (The included instructions are sufficiently simple as to be unlikely to need such assistance. Moreover, they are likely to be needed in interrupt handlers before HSRR0 and HSRR1 have been saved or after HSRR0 and HSRR1 have been restored.)

Similarly, fetching instructions may have the side effect of altering the contents of HSRR0 and HSRR1 unless MSR_{IR}=0.

6.5 Interrupt Definitions

Figure 51 shows all the types of interrupts and the values assigned to the MSR for each. Figure 52 shows the effective address of the interrupt vector for each interrupt type. (Section 5.7.4 on page 895 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 52.)

Interrupt Type	**MSR Bit**							
	IR	DR	FE0	FE1	EE	RI	ME	HV
System Reset	0	0	0	0	0	0	p	1
Machine Check	0	0	0	0	0	0	0	1
Data Storage	r	r	0	0	0	0	-	-
Data Segment	r	r	0	0	0	0	-	-
Instruction Storage	r	r	0	0	0	0	-	-
Instruction Segment	r	r	0	0	0	0	-	-
External	r	r	0	0	0	h	-	e
Alignment	r	r	0	0	0	0	-	-
Program	r	r	0	0	0	0	-	-
FP Unavailable[3]	r	r	0	0	0	0	-	-
Decrementer	r	r	0	0	0	0	-	-
Directed Privileged Doorbell Interrupt[4]	r	r	0	0	0	0	-	-
Hypervisor Decrementer	r	r	0	0	0	-	-	1
System Call	r	r	0	0	0	0	-	s
Trace	r	r	0	0	0	0	-	-
Hypervisor Data Storage	0	0	0	0	0	-	-	1
Hypervisor Instr. Storage.	0	0	0	0	0	-	-	1
Hypv Emulation Assistance	r	r	0	0	0	-	-	1
Hypervisor Maintenance	0	0	0	0	0	-	-	1
Directed Hypervisor Doorbell Interrupt[4]	r	r	0	0	0	-	-	1
Performance Monitor	r	r	0	0	0	0	-	-
Vector Unavailable[1]	r	r	0	0	0	0	-	-
VSX Unavailable[2]	r	r	0	0	0	0	-	-
Facility Unavailable	r	r	0	0	0	0	-	-
Hypervisor Facility Unavailable	r	r	0	0	0	-	-	1

Interrupt Type	MSR Bit IR DR FE0 FE1 EE RI ME HV

0 bit is set to 0
1 bit is set to 1
- bit is not altered
r if $LPCR_{AIL}$=2 or 3, $MSR_{IR\ DR}$ = 0b11, and the interrupt does not cause a transition from MSR_{HV}=0 to MSR_{HV}=1, set to 1; otherwise set to 0
p if the interrupt occurred while the thread was in power-saving mode, set to 1; otherwise not altered
e if LPES=0, set to 1; otherwise not altered
h if LPES=1, set to 0; otherwise not altered
s if LEV=1, set to 1; otherwise not altered

Settings for Other Bits

Bits BE, FP, PR, SE, TM[5], SLE, VEC[1], VSX[2], and PMM are set to 0.

If the interrupt results in HV being equal to 1, the LE bit is copied from the HILE bit; otherwise the LE bit is copied from the $LPCR_{ILE}$ bit.

The SF bit is set to 1.

If the TS field contained 0b10 (Transactional) when the interrupt occurred, the TS field is set to 0b01 (Suspended); otherwise the TS field is not altered.

Reserved bits are set as if written as 0.

[1] Category: Vector
[2] Category: Vector Scalar Emulation
[3] Category: Floating-Point
[4] Category: Server.Processor Control
[5] Category: Transactional Memory

Figure 51. MSR setting due to interrupt

Effective Address[1]	Interrupt Type
00..0000_0100	System Reset
00..0000_0200	Machine Check
00..0000_0300	Data Storage
00..0000_0380	Data Segment
00..0000_0400	Instruction Storage
00..0000_0480	Instruction Segment
00..0000_0500	External
00..0000_0600	Alignment
00..0000_0700	Program
00..0000_0800	Floating-Point Unavailable[5]
00..0000_0900	Decrementer
00..0000_0980	Hypervisor Decrementer
00..0000_0A00	Directed Privileged Doorbell[6]
00..0000_0B00	Reserved
00..0000_0C00	System Call
00..0000_0D00	Trace
00..0000_0E00	Hypervisor Data Storage
00..0000_0E20	Hypervisor Instruction Storage
00..0000_0E40	Hypervisor Emulation Assistance
00..0000_0E60	Hypervisor Maintenance
00..0000_0E80	Directed Hypervisor Doorbell[6]
00..0000_0EA0	Reserved
00..0000_0EC0	Reserved
00..0000_0EE0	Reserved for implementation-dependent interrupt for performance monitoring
00..0000_0F00	Performance Monitor
00..0000_0F20	Vector Unavailable[3]
00..0000_0F40	VSX Unavailable[4]
00..0000_0F60	Facility Unavailable
00..0000_0F80	Hypervisor Facility Unavailable
. . .	. . .
00..0000_0FFF	Reserved

1 The values in the Effective Address column are interpreted as follows.
- 00...0000_0nnn *means* 0x0000_0000_0000_0nnn unless the values of $LPCR_{AIL}$ and $MSR_{HV\ IR\ DR}$ cause the application of an effective address offset. See the description of $LPCR_{AIL}$ in Section 2.2 for more details.

2 Effective addresses 0x0000_0000_0000_0000 through 0x0000_0000_0000_00FF are used by software and will not be assigned as interrupt vectors.

3 Category: Vector.

4 Category: Vector Scalar Extension

5 Category: Floating Point

6 Category: Server.Processor Control

Figure 52. Effective address of interrupt vector by interrupt type

exception that caused exit from power-sav-

Programming Note

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 52 risks incompatibility with future implementations.

6.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing or causes the loss of a Machine Check exception or a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

When the thread is in any power-saving level, a System Reset interrupt occurs when a System Reset exception exists. When the thread is in doze or nap power-saving levels, a System Reset interrupt occurs when any of the following exceptions exists provided that the exception is enabled to cause exit from power saving mode (see Section 2.2, "Logical Partitioning Control Register (LPCR)"). When the thread is in sleep or rvwinkle power-saving level, it is implementation-specific whether the following exceptions, when enabled, cause exit, or whether only a system-reset causes exit.

- External
- Decrementer
- Directed Privileged Doorbell <S.PC>
- Directed Hypervisor Doorbell <S.PC>
- Hypervisor Maintenance
- Implementation-specific

SRR1 indicates the exception that caused exit from power-saving mode as specified below.

The following registers are set:

SRR0 If the interrupt did not occur when the thread was in power-saving mode, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present; otherwise, set to an undefined value.

SRR1

33 Implementation-dependent.

34:36 Set to 0.

42:45 If the interrupt did not occur when the thread was in power-saving mode, set to an implementation-specific value. If the interrupt occurred when the thread was in power-saving mode, set to indicate the

ing mode as shown below:

$SRR1_{42:45}$	Exception
0000	Reserved
0001	Reserved
0010	Implementation specific
0011	Directed Hypvsr Doorbell
0100	System Reset
0101	Directed Privlgd Doorbell
0110	Decrementer
0111	Reserved
1000	External
1001	Reserved
1010	Hypervisor Maintenance
1011	Reserved
1100	Implementation specific
1101	Reserved
1110	Implementation specific
1111	Reserved

If multiple exceptions that cause exit from power-saving mode exist, the exception reported is the exception corresponding to the interrupt that would have occurred if the same exceptions existed and the thread was not in power-saving mode.

46:47 Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows:

00 The interrupt did not occur when the thread was in power-saving mode.

01 The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.

10 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution.

11 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

Programming Note

Although the resources that are maintained in power-saving mode (except in doze power-saving level) are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of $SRR1_{46:47}$, to determine what state to restore. (To avoid implementation-dependence in the portion of the hypervisor that enters power-saving mode, the hypervisor must use the specification of the four instructions to determine what state to save.)

62 If the interrupt did not occur while the thread was in power-saving mode, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in power-saving mode, set to 1 if the thread is in a recoverable state; otherwise set to 0.

Others Loaded from the MSR.

MSR See Figure 51 on page 951.

In addition, if the interrupt occurs when the thread is in power-saving mode and is caused by an exception other than a System Reset exception, all other registers, except HSRR0 and HSRR1, that would be set by the corresponding interrupt if the exception occurred when the thread was not in power-saving mode are set by the System Reset interrupt, and are set to the values to which they would be set if the exception occurred when the thread was not in power-saving mode.

Execution resumes at effective address 0x0000_0000_0000_0100.

The means for software to distinguish between power-on Reset and other types of System Reset are implementation-dependent.

6.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check

interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 5.6), or by an error in the storage subsystem.

When the thread is not in power-saving mode, Machine Check interrupts are enabled when MSR_{ME}=1; if MSR_{ME}=0 and a Machine Check exception occurs, the thread enters the Checkstop state. When the thread is in doze or nap power-saving levels, Machine Check interrupts are treated as enabled when $LPCR_{PECE[2]}$=1 and cannot occur when $LPCR_{PECE[2]}$=0. When the thread is in sleep or rvwinkle power-saving level, it is implementation-specific whether Machine Check interrupts are treated as enabled under the same conditions as in doze and nap power-saving level or if they cannot occur. If a Machine Check exception occurs while the thread is in power-saving mode and the Machine Check exception is not enabled to cause exit from power-saving mode, the result is implementation specific

The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 5.6), or if an implementation-dependant hardware error occurs that prevents continued operation.

Disabled Machine Check (Checkstop State)

When a thread is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the thread. Some implementations may preserve some or all of the internal state of the thread when entering Checkstop state, so that the state can be analyzed as an aid in problem determination.

Enabled Machine Check

If a Machine Check exception causes an interrupt that is not context synchronizing or causes the loss of a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

In some systems, the operating system may attempt to identify and log the cause of the Machine Check.

The following registers are set:

SRR0 If the interrupt did not occur while the thread was in power-saving mode, set on a "best effort" basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred; otherwise set to an undefined value.

Programming Note

Since the hypervisor can save the address of the instruction following the *Power-Saving Mode* instruction if needed, there is no need for the thread to preserve it and store it into SRR0. Therefore, for ease of implementation, the contents of SRR0 upon exit from power-saving mode are specified to be undefined.

SRR1

46:47 Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows.

00 The interrupt did not occur when the thread was in power-saving mode.

01 The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.

10 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution.

11 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

Programming Note

Although the resources that are maintained in power-saving mode (except in the doze power-saving level) are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of $SRR1_{46:47}$, to determine what state to restore. (To avoid implementation-dependence in the portion of the hypervisor that enters power-saving mode, the hypervisor must use the specification of the four instructions to determine what state to save.)

62 If the interrupt did not occur while the thread was in power-saving mode, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in power-saving mode, set to 1 if the thread is in a recoverable state; otherwise set to 0.

Others Set to an implementation-dependent value.

MSR See Figure 51.

DSISR Set to an implementation-dependent value.

DAR Set to an implementation-dependent value.

Execution resumes at effective address 0x0000_0000_0000_0200.

A Machine Check interrupt caused by the existence of multiple SLB entries or TLB entries (or similar entries in implementation-specific translation caches) which translate a given effective or virtual address (see Sections 5.7.6.2 and 5.7.7.3.) must occur while still in the context of the partition that caused it. The interrupt must be presented in a way that permits continuing execution, with damage limited to the causing partition. Treating the exception as instruction-caused will achieve these requirements.

Programming Note

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, which may be placed into registers. This corruption of register contents may occur even if the interrupt is recoverable.

6.5.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists, the value of the expression

$(MSR_{HV\ PR} = 0b10) \mid (\neg VPM_0\ \&\ \neg MSR_{DR})$

$\mid (\neg VPM_1\ \&\ MSR_{DR})$

is 1, and a data access cannot be performed for any of the following reasons.

- Data address translation is enabled (MSR_{DR}=1) and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a real address.
- The effective address specified by a ***lq***, ***stq***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited.
- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection and $LPCR_{KBV}$=0.
- A Data Address Watchpoint match occurs.
- Execution of an ***eciwx*** or ***ecowx*** instruction is disallowed because EAR_E=0.
- An attempt is made to execute a *Fixed-Point Load or Store Caching Inhibited* instruction with MSR_{DR}=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

If a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, a Data Storage interrupt does not occur.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51.

DSISR

32 Set to 0.

33 Set to 1 if MSR_{DR}=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0..

34:35 Set to 0.

36 Set to 1 if the access is not permitted by Figure 32 or 33, as appropriate; otherwise set to 0.

37 Set to 1 if the access is due to a ***lq***, ***stq***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruc-

tion that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.

38 Set to 1 for a *Store*, ***dcbz***, or ***ecowx*** instruction; otherwise set to 0.

39:40 Set to 0.

41 Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43 Set to 1 if execution of an ***eciwx*** or ***ecowx*** instruction is attempted when EAR_E=0; otherwise set to 0.

44:61 Set to 0.

62 Set to 1 if an attempt is made to execute a *Fixed-Point Load or Store Caching Inhibited* instruction with MSR_{DR}=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

63 Set to 0.

DAR Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the DAR is set as described by the first item that corresponds to an exception that is reported in the DSISR. For example, if a *Load Word* instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the DSISR), the DAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- a Data Storage exception occurs for reasons other than a Data Address Watchpoint match or, for ***eciwx*** and ***ecowx***, EAR_E=0
  - a byte in the block that caused the exception, for a *Cache Management* instruction
  - a byte in the first aligned quadword for which access was attempted in the page that caused the exception, for a quadword *Load* or *Store* instruction (i.e., a *Load* or *Store* instruction for which the storage operand is a quadword; "first" refers to address order: see Section 6.7)
  - a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a non-quadword *Load* or *Store* instruction or an ***eciwx*** or ***ecowx*** instruction
- undefined, for a Data Address Watchpoint match, or if ***eciwx*** or ***ecowx*** is executed when EAR_E=0

For the cases in which the DAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR. However, if one or more DSI-causing exceptions occur together with a Virtualized Page Class Key Storage Protection exception that occurs when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM_1=0, an HDSI results, and all of the exceptions are reported in the HDSISR.

Execution resumes at effective address 0x0000_0000_0000_0300, possibly offset as specified in Figure 52.

6.5.4 Data Segment Interrupt

A Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled and the effective address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a virtual address.

If a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** would not perform its store in the absence of a Data Segment interrupt and a non-conditional *Store* to the specified effective address would cause a Data Segment interrupt, it is implementation-dependent whether a Data Segment interrupt occurs.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, a Data Segment interrupt does not occur.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51.

DSISR Set to an undefined value.

DAR Set to the effective address of a storage element as described in the following list.

- a byte in the block that caused the exception, for a *Cache Management* instruction
- a byte in the first aligned quadword for which access was attempted in the

segment that caused the exception, for a quadword *Load* or *Store* instruction (i.e., a *Load* or *Store* instruction for which the storage operand is a quadword; "first" refers to address order: see Section 6.7)

- a byte in the first aligned doubleword for which access was attempted in the segment that caused the exception, for a non-quadword *Load* or *Store* instruction or an ***eciwx*** or ***ecowx*** instruction

If the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address 0x0000_0000_0000_0380, possibly offset as specified in Figure 52.

Programming Note

A Data Segment interrupt occurs if MSR_{DR}=1 and the translation of the effective address of any byte of the specified storage location is not found in the SLB (or in any implementation-specific address translation lookaside information).

6.5.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists, the value of the expression

$$(MSR_{HV\ PR} = 0b10) \mid (\neg VPM_0\ \&\ \neg MSR_{IR}) \mid (\neg VPM_1\ \&\ MSR_{IR})$$

is 1, and the next instruction to be executed cannot be fetched for any of the following reasons.

- Instruction address translation is enabled and the virtual address cannot be translated to a real address.
- The fetch access violates storage protection.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33 Set to 1 if MSR_{IR}=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34 Set to 0.

35 Set to 1 if the access is to No-execute or Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 32 or 33, as appropriate; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51.

If multiple Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in SRR1.

Execution resumes at effective address 0x0000_0000_0000_0400, possibly offset as specified in Figure 52.

6.5.6 Instruction Segment Interrupt

An Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled and the effective address cannot be translated to a virtual address.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0480, possibly offset as specified in Figure 52.

Programming Note

An Instruction Segment interrupt occurs if MSR_{IR}=1 and the translation of the effective address of the next instruction to be executed is not found in the SLB (or in any implementation-specific address translation lookaside information).

6.5.7 External Interrupt

An External interrupt is classified as being either a Direct External interrupt or a Mediated External interrupt. Throughout this Book, usage of the phrase "External interrupt', without further classification, refers to

both a Direct External interrupt and a Mediated External interrupt.

6.5.7.1 Direct External Interrupt

A Direct External interrupt occurs when no higher priority exception exists, a Direct External exception exists, and the value of the expression

MSR_{EE} | (¬($LPES_0$) & (¬(MSR_{HV}) | MSR_{PR}))

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

When $LPES_0$=0, the following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

When $LPES_0$=1, the following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 52.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

$$MSR_{EE} \mid \neg(LPES_0 \mid MSR_{HV})$$

is equivalent to the expression given above.

Programming Note

The Direct External exception has the same meaning as the External exception in versions of the architecture prior to Version 2.05.

6.5.7.2 Mediated External Interrupt

A Mediated External interrupt occurs when no higher priority exception exists, a Mediated External exception exists (see the definition of $LPCR_{MER}$ in Section 2.2), and the value of the expression

$$MSR_{EE} \,\&\, (\neg(MSR_{HV}) \mid MSR_{PR})$$

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

When $LPES_0$=0, the following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1
- **33:36** Set to 0.
- **42** Set to 1.
- **43:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

When $LPES_0$=1, the following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 52.

6.5.8 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The operand of a *Floating-Point Load*, a *Floating-Point Store*, a *Vector-Scalar Extension Load*, or a *Vector-Scalar Extension Store* is not word-aligned, or crosses a virtual page boundary.
- The operand of ***lq***, ***stq***, ***lmw***, ***stmw***, ***lharx***, ***lwarx***, ***ldarx***, ***sthcx.***, ***stwcx.***, ***stdcx.***, ***eciwx***, ***ecowx, lfdp***, ***lfdpx***, ***stfdp***, or ***stfdpx*** is not aligned.
- The operand of a single-register *Load* or *Store* is not aligned and the thread is in Little-Endian mode.
- The instruction is ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, or ***stswx***, and the operand is in storage that is Write Through Required or Caching Inhibited, or the thread is in Little-Endian mode.
- The operand of a *Load* or *Store* crosses a segment boundary, or crosses a boundary between virtual pages that have different storage control attributes.
- The operand of a *Load* or *Store* is not aligned and is in storage that is Write Through Required or Caching Inhibited.

- The operand of ***lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx***, or ***dcbz*** is in storage that is Write Through Required or Caching Inhibited.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, an Alignment interrupt does not occur.

Setting the DSISR and DAR as described below is optional for implementations on which Alignment interrupts occur rarely, if ever, for cases that the Alignment interrupt handler emulates. For such implementations, if the DSISR and DAR are not set as described below they are set to undefined values.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51.

DSISR

32:43 Set to 0.

44:45 Set to bits 30:31 of the instruction if DS-form. Set to 0b00 if D-, or X-form.

46 Set to 0.

47:48 Set to bits 29:30 of the instruction if X-form. Set to 0b00 if D- or DS-form.

49 Set to bit 25 of the instruction if X-form. Set to bit 5 of the instruction if D- or DS-form.

50:53 Set to bits 21:24 of the instruction if X-form. Set to bits 1:4 of the instruction if D- or DS-form.

54:58 Set to bits 6:10 of the instruction (RT/RS/FRT/FRS), except undefined for ***dcbz***.

59:63 Set to bits 11:15 of the instruction (RA) for update form instructions; set to either bits 11:15 of the instruction or to any register number not in the range of registers to be loaded for a valid form ***lmw***, a valid form ***lswi***, or a valid form ***lswx*** for which neither RA nor RB is in the range of registers to be loaded; otherwise undefined.

DAR Set to the effective address computed by the instruction, except that if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

For an X-form *Load* or *Store*, it is acceptable for the thread to set the DSISR to the same value that would have resulted if the corresponding D- or DS-form instruction had caused the interrupt. Similarly, for a D- or DS-form *Load* or *Store*, it is acceptable for the thread to set the DSISR to the value that would have resulted for the corresponding X-form instruction. For example, an unaligned ***lwax*** (that crosses a protection boundary) would normally, following the description above, cause the DSISR to be set to binary:

```
000000000000 00 0 01 0 0101 ttttt ?????
```

where "ttttt" denotes the RT field, and "?????" denotes an undefined 5-bit value. However, it is acceptable if it causes the DSISR to be set as for ***lwa***, which is

```
000000000000 10 0 00 0 1101 ttttt ?????
```

If there is no corresponding alternative form instruction (e.g., for ***lwaux***), the value described above is set in the DSISR.

The instruction pairs that may use the same DSISR value are.

lhz/lhzx	lhzu/lhzux	lha/lhax	lhau/lhaux
lwz/lwzx	lwzu/lwzux	lwa/lwax	
ld/ldx	ldu/ldux		
lsth/sthx	sthu/sthux	stw/stwx	stwu/stwux
std/stdx	stdu/stdux		
lfs/lfsx	lfsu/lfsux	lfd/lfdx	lfdu/lfdux
stfs/stfsx	stfsu/stfsux	stfd/stfdx	stfdu/stfdux

Execution resumes at effective address 0x0000_0000_0000_0600, possibly offset as specified in Figure 52.

Programming Note

The architecture does not support the use of an unaligned effective address by ***lharx***, ***lwarx***, **ldarx**, ***sthcx.***, ***stwcx.***, ***stdcx.***, ***eciwx***, and ***ecowx***.

If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to simulate the instruction, but instead should treat the instruction as a programming error.

6.5.9 Program Interrupt

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

Floating-Point Enabled Exception

A Floating-Point Enabled Exception type Program interrupt is generated when the value of the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

is 1. $FPSCR_{FEX}$ is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a *Move To FPSCR*

instruction that causes an exception bit and the corresponding enable bit both to be 1.

TM Bad Thing Exception [Category: Transactional Memory]

A TM Bad Thing exception is generated when any of the following occurs.

- An ***rfebb***, ***rfid***, ***hrfid***, or ***mtmsrd*** instruction attempts to cause an illegal state transition (see Section 3.2.2).
- An ***rfid***, ***hrfid***, or ***mtmsrd*** instruction attempts to cause a transition to Problem state with an active transaction (Transactional or Suspended state) when TM is disabled by the PCR (PCR_{TM}=1 or $PCR_{v2.06}$=1).
- An ***rfebb*** instruction in Problem state attempts to cause a transition to Transactional or Suspended state when PCR_{TM}=1 (i.e. a latent non-zero TS value was in the BESCR).
- An attempt is made to execute ***trechkpt.*** in Transactional or Suspended state or when $TEXASR_{FS}$=0.
- An attempt is made to execute ***tend.*** in Suspended state.
- An attempt is made to execute ***treclaim.*** in Non-transactional state.
- An attempt is made to execute a ***mtspr*** targeting a TM register in other than Non-transactional state.
- An attempt is made to execute a power saving instruction in Suspended state.

Privileged Instruction

The following applies if the instruction is executed when MSR_{PR} = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an ***mtspr*** or ***mfspr*** instruction with an SPR field that contains a value having spr_0=1.

The following applies if the instruction is executed when $MSR_{HV\ PR}$ = 0b00.

A Privileged Instruction type Program interrupt is generated when execution is attempted of an ***mtspr*** or ***mfspr*** instruction with an SPR field that designates an SPR that is accessible by the instruction only when the thread is in hypervisor state, or when execution of a hypervisor-privileged instruction is attempted.

Programming Note

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR_{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining $SRR1_{49}$ (the bit in which MSR_{PR} was saved by the interrupt).

Trap

A Trap type Program interrupt is generated when any of the conditions specified in a *Trap* instruction is met.

The following registers are set:

SRR0 For all Program interrupts except a Floating-Point Enabled Exception type Program interrupt, set to the effective address of the instruction that caused the corresponding exception.

For a Floating-Point Enabled Exception type Program interrupt, set as described in the following list.

- If $MSR_{FE0\ FE1}$ = 0b00, $FPSCR_{FEX}$ = 1, and an instruction is executed that changes $MSR_{FE0\ FE1}$ to a nonzero value, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

Programming Note

Recall that all instructions that can alter $MSR_{FE0\ FE1}$ are context synchronizing, and therefore are not initiated until all preceding instructions have reported all exceptions they will cause.

- If $MSR_{FE0\ FE}$ = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If $MSR_{FE0\ FE}$ = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time $FPSCR_{FEX}$ was changed from 1 to 0 or of some subsequent instruction.

Programming Note

If SRR0 is set to the effective address of a subsequent instruction, that instruction will not be beyond the first such instruction at which synchronization of floating-point instructions occurs. (Recall that such synchronization is caused by *Floating-Point Status and Control Register* instructions, as well as by execution synchronizing instructions and events.)

SRR1

33:36 Set to 0.

42 Set to 1 for a TM Bad Thing Exception type Program interrupt; otherwise set to 0.

43 Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.

44 Set to 0.

45 Set to 1 for a Privileged Instruction type Program interrupt; otherwise set to 0.

46 Set to 1 for a Trap type Program interrupt; otherwise set to 0.

47 Set to 0 if SRR0 contains the address of the instruction causing the exception and there is only one such instruction; otherwise set to 1.

Programming Note

$SRR1_{47}$ can be set to 1 only if the exception is a Floating-Point Enabled Exception and either $MSR_{FE0\ FE1}$ = 0b01 or 0b10 or $MSR_{FE0\ FE1}$ has just been changed from 0b00 to a nonzero value. ($SRR1_{47}$ is always set to 1 in the last case.)

Others Loaded from the MSR.

Exactly one of bits 42, 43, 45, and 46 is set to 1.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0700, possibly offset as specified in Figure 52.

Programming Note

In versions of the architecture that precede V. 2.05, the conditions that now cause a Hypervisor Emulation Assistance interrupt instead caused an “Illegal Instruction type Program interrupt”. This was a Program interrupt for which registers (SRR0, SRR1, and the MSR) were set as described above for the Privileged Instruction type Program interrupt, except that $SRR1_{44}$ was set to 1 and $SRR1_{45}$ was set to 0. Thus operating systems have code to handle these conditions, at the Program interrupt vector location. For this reason, if a Hypervisor Emulation Assistance interrupt occurs, when the thread is not in hypervisor state, for an instruction that the hypervisor does not emulate, the hypervisor should pass control to the operating system at the operating system's Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt, except with $SRR1_{44:45}$ set to 0b10. (The Hypervisor Emulation Assistance interrupt was added to the architecture in V. 2.05, and the Illegal Instruction type Program interrupt was removed from the architecture in V. 2.06. In V. 2.05 the Hypervisor Emulation Assistance interrupt was optional: implementations that supported it generated it as described in V. 2.06, and never generated an Illegal Instruction type Program interrupt; implementations that did not support it generated an Illegal Instruction type Program interrupt as described above.)

6.5.10 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR_{FP}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0800, possibly offset as specified in Figure 52.

6.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists, and MSR_{EE}=1.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0900, possibly offset as specified in Figure 52.

6.5.12 Hypervisor Decrementer Interrupt

A Hypervisor Decrementer interrupt occurs when no higher priority exception exists, a Hypervisor Decrementer exception exists, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$ & HDICE

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0980, possibly offset as specified in Figure 52.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

$(MSR_{EE} \mid \neg(MSR_{HV})) \& HDICE$

is equivalent to the expression given above.

6.5.13 Directed Privileged Doorbell Interrupt [Category: Server.Processor Control

A Directed Privileged Doorbell interrupt occurs when no higher priority exception exists, a Directed Privileged Doorbell exception is present, and MSR_{EE}=1. Directed Privileged Doorbell exceptions are generated when Directed Privileged Doorbell messages (see Chapter 11) are received and accepted by the thread.

Save/Restore Register 0, Save/Restore Register 1 and Machine State Register are updated as follows:

Save/Restore Register 0 Set to the effective address of the next instruction to be executed.

Save/Restore Register 1 Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0A00, possibly offset as specified in Figure 52.

6.5.14 System Call Interrupt

A System Call interrupt occurs when a *System Call* instruction is executed.

The following registers are set:

SRR0 Set to the effective address of the instruction following the System Call instruction.

SRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0C00, possibly offset as specified in Figure 52.

Programming Note

An attempt to execute an ***sc*** instruction with LEV=1 in problem state should be treated as a programming error.

6.5.15 Trace Interrupt [Category: Trace]

A Trace interrupt occurs when no higher priority exception exists and any instruction except ***rfid***, ***hrfid,*** or a *Power-Saving Mode* instruction is successfully completed, provided any of the following is true:

- the instruction is ***mtmsr[d]*** and MSR_{SE}=1 when the instruction was initiated,
- the instruction is not ***mtmsr[d]*** and MSR_{SE}=1,
- the instruction is a *Branch* instruction and MSR_{BE}=1, or
- a CIABR match occurs.

Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state <TM>, did not cause the transaction to fail in such a way that the instruction did not complete. (See Section 5.3.1 of Book II). Thus a Trace interrupt never occurs for a *System Call* instruction, or for a *Trap* instruction that traps, or for a ***dcbf*** that is executed in Transactional state. The instruction that causes a Trace interrupt is called the “traced instruction”.

When a Trace interrupt occurs, the following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1

33	Set to 1.
34	Set to 0.
35	Set to 1 if the traced instruction is a *Load* instruction or is specified to be treated as a *Load* instruction; otherwise set to 0.
36	Set to 1 if the traced instruction is a *Store* instruction or is specified to be treated as a *Store* instruction; otherwise set to 0.
43	Set to 1 if the traced instruction is the result of a CIABR match.
44:47	Set to 0.
Others	Loaded from the MSR.

Architecture Note

Only bit 43 is set in SRR1 for a CIABR match.

Programming Note

Bit 33 is set to 1 for historical reasons.

SIAR Set to the effective address of the traced instruction.

SDAR Set to the effective address of the storage operand (if any) of the traced instruction; otherwise undefined.

If the state of the Performance Monitor is such that the Performance Monitor may be altering the SIAR and SDAR (i.e., if $MMCR0_{PMAE}$=1), the contents of the SIAR and SDAR are undefined for the Trace interrupt and may change even when no Trace interrupt occurs.

Architecture Note

The SIAR and SDAR are not set when a Trace interrupt occurs for a CIABR match.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_00D0, possibly offset as specified in Figure 52. For a Trace interrupt resulting from execution of an instruction that modifies the value of MSR_{IR} or MSR_{DR}, the Trace interrupt vector location is based on the modified values.

Programming Note

The following instructions are not traced.

- ***rfid***
- ***hrfid***
- ***sc***, and *Trap* instructions that trap
- *Power-Saving Mode* instructions
- other instructions that cause interrupts (other than Trace interrupts)
- the first instructions of any interrupt handler
- instructions that are emulated by software
- instructions, executed in Transactional state, that are disallowed in Transactional state
- instructions, executed in Transactional state, that cause types of accesses that are disallowed in Transactional state
- ***mtspr***, executed in Transactional state, specifying an SPR that is not part of the Transactional Memory checkpointed registers
- ***tbegin.*** executed at maximum nesting depth

In general, interrupt handlers can achieve the effect of tracing these instructions.

6.5.16 Hypervisor Data Storage Interrupt

A Hypervisor Data Storage interrupt occurs when no higher priority exception exists, the thread is not in hypervisor state, and either (a) VPM_1=0, $LPCR_{KBV}$=1, and a Virtual Storage Page Class Key Protection exception exists or (b) the value of the expression

$(VPM_0 \,\&\, \neg MSR_{DR}) \mid (VPM_1 \,\&\, MSR_{DR})$

is 1, and a data access cannot be performed for any of the following reasons.

- Data address translation is enabled (MSR_{DR}=1) and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a real address.
- Data address translation is disabled (MSR_{DR}=0), and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, ***dcbf***[***l***], ***eciwx***, or ***ecowx*** instruction cannot be translated to a real address by means of the virtual real addressing mechanism.
- The effective address specified by a ***lq***, ***stq***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited.
- The access violates storage protection.
- A Data Address Watchpoint match occurs.
- Execution of an ***eciwx*** or ***ecowx*** instruction is disallowed because EA_{RE}=0.

If a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.,*** or ***stqcx.*** would not perform its store in the absence of a Hypervisor Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Hypervisor Data Storage interrupt, it is implementation-dependent whether a Hypervisor Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, a Hypervisor Data Storage interrupt does not occur.

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51.

HDSISR

32 Set to 0.

33 Set to 1 if the value of the expression $(MSR_{DR}) \mid (\neg MSR_{DR} \,\&\, VPM_0)$

is 1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34:35 Set to 0.

36 Set to 1 if the access is not permitted by Figure 32 or 33, as appropriate; otherwise set to 0.

37 Set to 1 if the access is due to a ***lq***, ***stq***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.

38 Set to 1 for a *Store*, ***dcbz***, or ***ecowx*** instruction; otherwise set to 0.

39:40 Set to 0.

41 Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43 Set to 1 if execution of an ***eciwx*** or ***ecowx*** instruction is attempted when EAR_E=0; otherwise set to 0.

44:63 Set to 0.

HDAR Set to the effective address of a storage element, as described in the following list. The list should be read from the top down; the HDAR is set as described by the first item that corresponds to an exception that is reported in the HDSISR. For example, if a *Load Word* instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the HDSISR), the HDAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- a Hypervisor Data Storage exception occurs for reasons other than a Data Address Watchpoint match or, for ***eciwx*** and ***ecowx***, EAR_E=0
  - a byte in the block that caused the exception, for a *Cache Management* instruction
  - a byte in the first aligned quadword for which access was attempted in the page that caused the exception, for a quadword *Load* or *Store* instruction (i.e., a *Load* or *Store* instruction for which the storage operand is a quadword; "first" refers to address order: see Section 6.7)
  - a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a non-quadword *Load* or *Store* instruction or an ***eciwx*** or ***ecowx*** instruction
- undefined, for a Data Address Watchpoint match, or if ***eciwx*** or ***ecowx*** is executed when EAR_E=0

For the cases in which the HDAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the HDAR are set to 0.

If multiple Hypervisor Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the HDSISR. If the HDSISR reports other exceptions together with a Virtualized Page Class Key Storage Protection exception that occurs when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM_1=0, the other exceptions are actually DSIs.

Programming Note

A Virtual Page Class Key Storage Protection exception that occurs with $LPCR_{KBV}$=1 and Virtualized Partition Memory disabled by VPM_1=0 identifies an access that must be emulated by the hypervisor. When it is reported together with other exceptions in the HDSISR, the hypervisor should service the Virtual Page Class Key Storage Protection exception first. This is in part because the operating system may be using some PTE fields for non-architected purposes, which could in turn cause spurious exceptions to be reported.

Execution resumes at effective address 0x0000_0000_0000_0E00, possibly offset as specified in Figure 52.

6.5.17 Hypervisor Instruction Storage Interrupt

A Hypervisor Instruction Storage interrupt occurs when the thread is not in hypervisor state, no higher priority exception exists, the value of the expression

$(VPM_0 \,\&\, \neg MSR_{IR}) \mid (VPM_1 \,\&\, MSR_{IR})$

is 1, and the next instruction to be executed cannot be fetched for any of the following reasons.

- Instruction address translation is enabled (MSR_{IR}=1) and the virtual address cannot be translated to a real address.
- Instruction address translation is disabled (MSR_{IR}=0), and the virtual address cannot be translated to a real address by means of the virtual real addressing mechanism.
- The fetch access violates storage protection.

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, HSRR0 is set to the branch target address).

HSRR1

33 Set to 1 if the value of the expression (MSR_{IR}) | (¬MSR_{IR} & VPM_0) is 1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34 Set to 0.

35 Set to 1 if the access is to No-execute or Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 32 or 33, as appropriate; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51.

If multiple Hypervisor Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in HSRR1.

Execution resumes at effective address 0x0000_0000_0000_0E10, possibly offset as specified in Figure 52.

6.5.18 Hypervisor Emulation Assistance Interrupt

A Hypervisor Emulation Assistance interrupt is generated when execution is attempted of an illegal instruction, or of a reserved instruction or an instruction that is not provided by the implementation. It is also generated under the following conditions.

- an ***mtspr*** or ***mfspr*** instruction is executed when MSR_{PR}=1 if the instruction specifies an SPR with spr_0=0 that is not provided by the implementation
- an ***mtspr*** or ***mfspr*** instruction is executed when MSR_{PR}=0 if the instruction specifies SPR 0
- an ***mfspr*** instruction is executed when MSR_{PR}=0 if the instruction specifies SPR 4, 5, or 6

A Hypervisor Emulation Assistance interrupt may be generated when execution is attempted of any of the following kinds of instruction.

- an instruction that is in invalid form
- an ***lswx*** instruction for which RA or RB is in the range of registers to be loaded

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51 on page 951.

HEIR Set to a copy of the instruction that caused the interrupt

Execution resumes at effective address 0x0000_0000_0000_0E40, possibly offset as specified in Figure 52.

Programming Note

If a Hypervisor Emulation Assistance interrupt occurs, when the thread is not in hypervisor state, for an instruction that the hypervisor does not emulate, the hypervisor should pass control to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt", as described in a Programming Note near the end of Section 6.5.9, “Program Interrupt” on page 959.

6.5.19 Hypervisor Maintenance Interrupt

A Hypervisor Maintenance interrupt occurs when no higher priority exception exists, a Hypervisor Maintenance exception exists (a bit in the HMER is set to one), the exception is enabled in the HMEER, and the value of the following expression is 1.

(MSR_{EE} | ¬(MSR_{HV}) | MSR_{PR})

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 51 on page 951.

HMER See Section 6.2.8 on page 940.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mthmer*** instruction.

Execution resumes at effective address 0x0000_0000_0000_0E60.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

$(MSR_{EE} \mid \neg(MSR_{HV}))$

is equivalent to the expression given above.

Programming Note

If an implementation uses the HMER to record that a readable resource, such as the Time Base, has been corrupted, then, because the HMI is disabled in the hypervisor state, it is necessary for the hypervisor to check HMER after reading that resource to be sure an error has not occurred.

6.5.20 Directed Hypervisor Doorbell Interrupt [Category: Server.Processor Control]

A Directed Hypervisor Doorbell interrupt occurs when no higher priority exception exists, a Directed Hypervisor Doorbell exception is present, and MSR_{HV}=0 | MSR_{EE}=1. Directed Hypervisor Doorbell exceptions are generated when Directed Hypervisor Doorbell messages (see Chapter 11) are received and accepted by the thread.

HSave/Restore Register 0, HSave/Restore Register 1 and Machine State Register are updated as follows:

HSRR0 Set to the effective address of the next instruction to be executed.

HSRR1 Set to the contents of the Machine State Register at the time of the interrupt.

Machine State Register See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0E80, possibly offset as specified in Figure 52.

6.5.21 Performance Monitor Interrupt [Category: Server.Performance Monitor]

A Performance Monitor interrupt occurs when no higher priority exception exists, a Performance Monitor exception exists, event-based branches are disabled, and MSR_{EE}=1.

If multiple Performance Monitor exceptions occur before the first causes a Performance Monitor interrupt, the interrupt reflects the most recent Performance Monitor exception and the preceding Performance Monitor exceptions are lost.

The following registers are set:

SRR0 Set to the effective address of the instruction that would have been attempted to be execute next if no interrupt conditions were present.

SRR1
- **33:36 and 42:47** Reserved.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0F00, possibly offset as specified in Figure 52.

6.5.22 Vector Unavailable Interrupt [Category: Vector]

A Vector Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a Vector instruction (including Vector loads, stores, and moves), and MSR_{VEC}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0F20, possibly offset as specified in Figure 52.

6.5.23 VSX Unavailable Interrupt [Category: VSX]

A VSX Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a VSX instruction (including VSX loads, stores, and moves), and MSR_{VSX}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

Execution resumes at effective address 0x0000_0000_0000_0F40, possibly offset as specified in Figure 52.

6.5.24 Facility Unavailable Interrupt

A Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem state when it has been made unavailable by the FSCR
- a Performance Monitor register or instruction (e.g. ***mfbhrbe***, ***clrbhrb***) is accessed in problem state when it has been made unavailable by MMCR0.
- the Transactional Memory Facility is accessed in any privilege state when it has been made unavailable by MSR_{TM}.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

FSCR
- **0:7** See Section 6.2.10 on page 941.
- **Others** Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F60, possibly offset as specified in Figure 52.

Programming Note

For the case of an outer ***tbegin.***, the interrupt handler should either return to the ***tbegin.*** with MSR_{TM} = 1 (allowing the program to use transactions), or treat the attempt to initiate an outer transaction as a program error.

6.5.25 Hypervisor Facility Unavailable Interrupt

A Hypervisor Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem or privileged states when it has been made unavailable by the HFSCR.

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 51 on page 951.

HFSCR
- **0:7** See Section 6.2.11 on page 942.
- **Others** Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F80, possibly offset as specified in Figure 52.

6.6 Partially Executed Instructions

If a Data Storage, Data Segment, Alignment, system-caused, or imprecise exception occurs while a *Load* or *Store* instruction is executing, the instruction may be aborted. In such cases the instruction is not completed, but may have been partially executed in the following respects.

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would violate storage protection, that byte is neither copied to a register by a *Load* instruction nor modified by a *Store* instruction. Also, the rules for storage accesses given in Section 5.8.1, "Guarded Storage" and in Section 2.1 of Book II are obeyed.
- Some registers may have been altered as described in the Book II section cited above.
- Reference and Change bits may have been updated as described in Section 5.7.8.
- For a ***stbcx.***, ***sthcx.***, ***stwcx., stdcx.***, or ***stqcx.*** instruction that is executed in-order, CR0 may have been set to an undefined value and the reservation may have been cleared.

The architecture does not support continuation of an aborted instruction but intends that the aborted instruction be re-executed if appropriate.

Programming Note

An exception may result in the partial execution of a *Load* or *Store* instruction. For example, if the Page Table Entry that translates the address of the storage operand is altered, by a program running on another thread, such that the new contents of the Page Table Entry preclude performing the access, the alteration could cause the *Load* or *Store* instruction to be aborted after having been partially executed.

As stated in the Book II section cited above, if an instruction is partially executed the contents of registers are preserved to the extent that the instruction can be re-executed correctly. The consequent preservation is described in the following list. For any given instruction, zero, one, or two items in the list apply.

- For a fixed-point *Load* instruction that is not a multiple or string form, or for an ***eciwx*** instruction, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an ***lq*** instruction, if RT+1 = RA then the contents of register RT+1 are not altered.
- For an update form *Load* or *Store* instruction, the contents of register RA are not altered.

6.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Some exceptions, such as the Mediated External exception, persist and can be deferred. However, other exceptions would be lost if they were not recognized and handled when they occur. For example, if an External interrupt was generated when a Data Storage exception existed, the Data Storage exception would be lost. If the Data Storage exception was caused by a *Store Multiple* instruction for which the storage operand crosses a virtual page boundary and the exception was a result of attempting to access the second virtual page, the store could have modified locations in the first virtual page even though it appeared that the *Store Multiple* instruction was never executed.

For the above reasons, all exceptions are prioritized with respect to other exceptions that may exist at the same instant to prevent the loss of any exception that is not persistent. Some exceptions cannot exist at the same instant as some others.

Data Storage, Hypervisor Data Storage, Data Segment, and Alignment exceptions and transaction failure due to attempted access of a disallowed type while in Transactional state occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0). (The required ordering of exceptions on components of non-atomic accesses does not extend to the performing of the component accesses in the event of an exception. For example, if byte n causes a data storage exception, it is not necessarily true that the access to byte n-1 has been performed.)

6.7.1 Unordered Exceptions

The exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt processing mechanism. These exceptions are recognized and processed when presented.

1. System Reset
2. Machine Check

6.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism. With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same ordering. The exception is that Virtual Page Class Key Storage Protection exceptions that occur when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM_1=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

System-Caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. Hypervisor Maintenance
3. External, [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell <S.PC>, Directed Hypervisor Doorbell <S.PC>

Instruction-Caused and Precise

1. Instruction Segment
2. [Hypervisor] Instruction Storage
3.a Hypervisor Emulation Assistance
3.b Program
   - Privileged Instruction
4. Function-Dependent
   4.a Fixed-Point and Branch
      1 Hypervisor Facility Unavailable
      2 Facility Unavailable
      3a Program
         - Trap
         - TM Bad Thing
      3b System Call
      3c.1 Data Storage for the case of *Fixed-Point Load or Store Caching Inhibited* instructions with MSR_{DR}=1
      3c.2 all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace
   4.b Floating-Point
      1 Hypervisor Facility Unavailable
      2 FP Unavailable
      3a Program
         - Precise Mode Floating-Pt Enabled Excep'n
      3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace
   4.c Vector
      1 Hypervisor Facility Unavailable
      2 Vector Unavailable
      3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace
   4.d VSX
      1 Hypervisor Facility Unavailable
      2 VSX Unavailable
      3a Program
         - Precise Mode Floating-Pt Enabled Excep'n
      3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace
   4.e Other Instructions
      1 Hypervisor Facility Unavailable
      2 Facility Unavailable
      3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace

For implementations that execute multiple instructions in parallel using pipeline or superscalar techniques, or combinations of these, it can be difficult to understand the ordering of exceptions.To understand this ordering it is useful to consider a model in which each instruction is fetched, then decoded, then executed, all before the next instruction is fetched. In this model, the exceptions a single instruction would generate are in the order shown in the list of instruction-caused exceptions. Exceptions with different numbers have different ordering. Exceptions with the same numbering but different lettering are mutually exclusive and cannot be caused by the same instruction. The External, [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell <S.PC>, and Directed Hypervisor Doorbell <S.PC> interrupts have equal ordering. Similarly, where Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal ordering.

Even on threads that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

6.8 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g., it does not include waiting for the Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority. The phrase "corresponding interrupt" means the interrupt having the same name as the exception unless the thread is in power-saving mode, in which case the phrase means the System Reset interrupt.

Unless otherwise stated or obvious from context, it is assumed below that one of the following conditions is satisfied.

- The thread is not in power-saving mode and the interrupt, unless it is the Machine Check interrupt, is not disabled. (For the Machine Check interrupt no assumption is made regarding enablement.)
- The thread is in power-saving mode and the exception is enabled to cause exit from the mode.

With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same priority. The exception is that exceptions caused by Virtual Page Class Key Storage Protection exceptions that occur when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM_1=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

1. System Reset

   System Reset exception has the highest priority of all exceptions. If this exception exists, the interrupt mechanism ignores all other exceptions and generates a System Reset interrupt.

Once the System Reset interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

2. Machine Check

   Machine Check exception is the second highest priority exception. If this exception exists and a System Reset exception does not exist, the interrupt mechanism ignores all other exceptions and generates a Machine Check interrupt.

   Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction-Caused and Precise

   This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where [Hypervisor] Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the hardware may generate any one of the three interrupts for which an exception exists). For instructions that are forbidden in Transactional state, transaction failure takes priority over all interrupts except Privileged Instruction type of Program Interrupts. For data accesses that are forbidden in Transactional state, transaction failure has the same priority as the group of “other” [Hypervisor] Data Storage, Data Segment, and Alignment exceptions. (See Section 5.3.1 of Book II).

   A. Fixed-Point Loads and Stores
      a. These exceptions are mutually exclusive and have the same priority:
         - Hypervisor Emulation Assistance
         - Program - Privileged Instruction
      b. Hypervisor Facility Unavailable
      c. Facility Unavailable
      d. Data Storage for the case of *Fixed-Point Load or Store Caching Inhibited* instructions with MSR_{DR}=1
      e. all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, or Alignment
      f. Trace

   B. Floating-Point Loads and Stores
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Floating-Point Unavailable
      d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      e Trace

   C. Vector Loads and Stores
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Vector Unavailable
      d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      e. Trace

   D. VSX Loads and Stores
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. VSX Unavailable
      d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      e. Trace

   E. Other Floating-Point Instructions
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Floating-Point Unavailable
      d. Program - Precise Mode Floating-Point Enabled Exception
      e. Trace

   F. Other Vector Instructions
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Vector Unavailable
      d. Trace

   G. Other VSX Instructions
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. VSX Unavailable
      d. Program - Precise Mode Floating-Point Enabled Exception
      e. Trace

   H. TM instruction, ***mt*/*fspr*** specifying TM SPR
      a. Program - Privileged Instruction (only for ***treclaim.***, ***trechkpt.***., and ***mtspr***)
      b Hypervisor Facility Unavailable
      c Facility Unavailable
      d Program - TM Bad Thing (only for ***treclaim.***, ***trechkpt.***, and ***mtspr***)
      e Trace

   I. ***rfid, hrfid, rfebb*** and ***mtmsr***[***d***]
      a. Program - Privileged Instruction for all except ***rfebb***
      b Program - TM Bad Thing exception for all except ***mtmsr***.
      c. Program - Floating-Point Enabled Exception for all except ***rfebb***
      d. Trace, for ***mtmsr***[***d***] and ***rfebb*** only

   J. Other Instructions
      a. These exceptions are mutually exclusive and have the same priority:
         - Program - Trap

- System Call
- Program - Privileged Instruction
- Hypervisor Emulation Assistance

b. Hypervisor Facility Unavailable
c. Facility Unavailable
d. Trace

K. [Hypervisor] Instruction Storage and Instruction Segment

These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The two exceptions are mutually exclusive.

The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. Hypervisor Maintenance

This exception is the fifth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Maintenance exception exists and each attempt to execute an instruction when the Hypervisor Maintenance interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Maintenance interrupt is not delayed indefinitely.

6. Direct External, Mediated External, and [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell <S.PC>, Directed Hypervisor Doorbell <S.PC>

These exceptions are the lowest priority exceptions. All have equal priority (i.e., the hardware may generate any one of the corresponding interrupts for which an exception exists). When one of these exceptions is created, the interrupt processing mechanism waits for all other possible exceptions to be reported. It then generates the corresponding interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Decrementer exception exists and each attempt to execute an instruction when the Hypervisor Decrementer interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Decrementer interrupt is not delayed indefinitely.

If LPES=1 and a Direct External exception exists and each attempt to execute an instruction when this interrupt is enabled causes an exception (see the Programming Note below), the Direct External interrupt is not delayed indefinitely.

Programming Note

An incorrect or malicious operating system could corrupt the first instruction in the interrupt vector location for an instruction-caused interrupt such that the attempt to execute the instruction causes the same exception that caused the interrupt (a looping interrupt; e.g., *Trap* instruction and Program interrupt). Similarly, the first instruction of the interrupt vector for one instruction-caused interrupt could cause a different instruction-caused interrupt, and the first instruction of the interrupt vector for the second instruction-caused interrupt could cause the first instruction-caused interrupt (e.g., Program interrupt and Floating-Point Unavailable interrupt). Similarly, if the Real Mode Area is virtualized and there is no PTE for the page containing the interrupt vectors, every attempt to execute the first instruction of the OS's Instruction Storage interrupt handler would cause a Hypervisor Instruction Storage interrupt; if the Hypervisor Instruction Storage interrupt handler returns to the OS's Instruction Storage interrupt handler without the relevant PTE having been created, another Hypervisor Instruction Storage interrupt would occur immediately. The looping caused by these and similar cases is terminated by the occurrence of a System Reset or Hypervisor Decrementer interrupt.

6.9 Relationship of Event-Based Branches to Interrupts

Event-based exceptions have a priority lower than all exceptions that cause interrupts. When an event-based exception is created, the Event-Based Branch facility waits for all other possible exceptions that would cause interrupts to be reported. It then generates the event-based branch if no exception that would cause an interrupt exists when the event-based branch is to be generated.

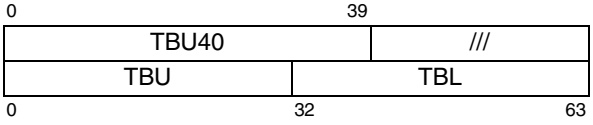
Chapter 7. Timer Facilities

7.1 Overview

The Time Base, Decrementer, Hypervisor Decrementer, Processor Utilization of Resources, and Scaled Processor Utilization of Resources registers provide timing functions for the system. The remainder of this section describes these registers and related facilities.

7.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 53) containing a 64-bit unsigned integer that is incremented periodically.



Field	Description
TBU40	Upper 40 bits of Time Base
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 53. Time Base

The Time Base is a hypervisor resource; see Chapter 2.

The SPRs TBU40, TBU, and TBL provide access to the fields of the Time Base shown in Figure 53. When a ***mtspr*** instruction is executed specifying one of these SPRs, the associated field of the Time Base is altered and the remaining bits of the Time Base are not affected.

See Chapter 6 of Book II for infromation about the update frequency of the Time Base.

The Time Base is implemented such that:

1. Loading a GPR from the Time Base has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Time Base replaces the contents of the Time Base with the contents of the GPR.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a Power ISA system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in problem state (MSR_{PR}=1). If the means is under software control, it must be privileged and, in implementations of the Server environment, must be accessible only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). There must be a method for getting all Time Bases in the system to start incrementing with values that are identical or almost identical.

Programming Note

If software initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from 2^{64}-1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

If Time Base bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0x0 only when bit 59 changes state regardless of whether or not they incremented to 0xF since they were previously set to 0x0.

See the description of the Time Base in Chapter 6 of Book II for ways to compute time of day in POSIX format from the Time Base.

7.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is not privileged; it is discussed in Chapter 6 of Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The ***mttbl*** and ***mttbu*** extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the ***mtspr*** instruction; see Appendix A, "Assembler Extended Mnemonics" on page 1015.

The Time Base can be written by a sequence such as:

```
lwz    Rx,upper # load 64-bit value for
lwz    Ry,lower # TB into Rx and Ry
li     Rz,0
mttbl  Rz       # set TBL to 0
mttbu  Rx       # set TBU
mttbl  Ry       # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

The preferred method of changing the Time Base utilizes the TBU40 facility. The following code sequence demonstrates the process. Assume the upper 40 bits of Rx contain the desired value upper 40 bits of the Time Base.

```
mftb   Ry      # Read 64-bit Time Base value
clrldi Ry,Ry,40# lower 24 bits of old TB
mttbu40Rx      # write upper 40 bits of TB
mftb   Rz      # read TB value again
clrldi Rz,Rz,40# lower 24 bits of new TB
cmpld  Rz,Ry   # compare new and old lwr 24
bge    done    # no carry out of low 24 bits
addis  Rx,Rx,0x0100#increment upper 40 bits
mttbu40 Rx     # update to adjust for carry
```

Programming Note

The instructions for writing the Time Base are mode-independent. Thus code written to set the Time Base will work correctly in either 64-bit or 32-bit mode.

7.3 Virtual Time Base

The Virtual Time Base (VTB) is a 64-bit incrementing counter.

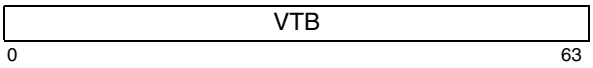


Figure 54. Virtual Time Base

Virtual Time Base increments at the same rate as the Time Base until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1); at the next increment its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The operation of the Virtual Time Base has the following additional properties.

1. Loading a GPR from the Virtual Time Base has no effect on the accuracy of the Virtual Time Base.
2. Copying the contents of a GPR to the Virtual Time Base replaces the contents of the Virtual Time Base with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Virtual Time Base input frequency will also change. Software must be aware of this in order to set interval timers.

Programming Note

In configurations in which the hypervisor allows multiple partitions to time-share a processor, the Virtual Time Base can be managed by the hypervisor such that it appears to each partition as if it counts only during the times that the partition is executing.

In order to do this, the hypervisor saves the value of the Virtual Time Base as part of the program context when removing a partition from the processor, and restores it to it's previous value when initiating the partition again on the same or another processor.

7.4 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

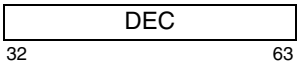


Figure 55. Decrementer

The Decrementer counts down until its value becomes 0x0000_0000; at the next decrement its value becomes 0xFFFF_FFFF.

The Decrementer is driven at the same frequency as the Time Base.

When the contents of DEC_{32} change from 0 to 1, a Decrementer exception will come into existence within a reasonable period of time. When the contents of DEC_{32} change from 1 to 0, the existing Decrementer exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DEC_{32} is the result of decrementation of the Decrementer by the hardware or of modification of the Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Decrementer has the following additional properties.

1. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Decrementer replaces the contents of the Decrementer with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

7.4.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are privileged when they refer to the Decrementer. Using an extended mnemonic (see Appendix A, “Assembler Extended Mnemonics” on page 1015), the Decrementer can be written from GPR Rx using:

```
mtdec  Rx
```

The Decrementer can be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

7.5 Hypervisor Decrementer

The Hypervisor Decrementer (HDEC) is a 32-bit decrementing counter that provides a mechanism for causing a Hypervisor Decrementer interrupt after a programmable delay. The contents of the Hypervisor Decrementer are treated as a signed integer.

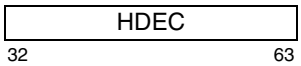


Figure 56. Hypervisor Decrementer

The Hypervisor Decrementer is a hypervisor resource; see Chapter 2.

The Hypervisor Decrementer counts down until its value becomes 0x0000_0000; at the next decrement its value becomes 0xFFFF_FFFF.

The Hypervisor Decrementer is driven at the same frequency as the Time Base.

When the contents of $HDEC_{32}$ change from 0 to 1 and the thread is not in a power-saving mode, a Hypervisor Decrementer exception will come into existence within a reasonable period of time. When a Hypervisor Decre-

menter interrupt occurs, the existing Hypervisor Decrementer exception will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event. Even if multiple $HDEC_{32}$ transitions from 0 to 1 occur before a Hypervisor Decrementer interrupt occurs, at most one Hypervisor Decrementer exception exists.

The preceding paragraph applies regardless of whether the change in the contents of $HDEC_{32}$ is the result of decrementation of the Hypervisor Decrementer by the hardware or of modification of the Hypervisor Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Hypervisor Decrementer has the following additional properties.

1. Loading a GPR from the Hypervisor Decrementer has no effect on the accuracy of the Hypervisor Decrementer.
2. Copying the contents of a GPR to the Hypervisor Decrementer replaces the contents of the Hypervisor Decrementer with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Hypervisor Decrementer update frequency will also change. Software must be aware of this in order to set interval timers.

If Hypervisor Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

Programming Note

A Hypervisor Decrementer exception is not created if the thread is in a power-saving mode when $HDEC_{32}$ changes from 0 to 1 because having a Hypervisor Decrementer interrupt occur almost immediately after exiting the power-saving mode in this case is deemed unnecessary. The hypervisor already has control, and if a timed exit from the power-saving mode is necessary and possible, the hypervisor can use the Decrementer to exit the power-saving mode at the appropriate time. For sleep and rvwinkle power-saving levels, the state of the Hypervisor Decrementer and Decrementer is not necessarily maintained and updated.

7.6 Processor Utilization of Resources Register (PURR)

The Processor Utilization of Resources Register (PURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the PURR are treated as a 64-bit unsigned integer.

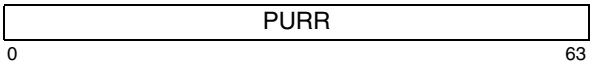


Figure 57. Processor Utilization of Resources Register

The PURR is a hypervisor resource; see Chapter 2.

The contents of the PURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1) at which point the contents are replaced by that sum modulo 2^{64}. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the PURR increases is an estimate of the portion of resources used by the thread per unit time with respect to other threads that share those resources monitored by the PURR. When the thread is idle, the rate at which the PURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be T_{ab}. Let the difference between the value represented by the contents of the PURR at time T_a and T_b be the value P_{ab}. The ratio of P_{ab}/T_{ab} is an estimate of the percentage of shared resources used by the thread during the interval T_{ab}. For the set {S} of threads that share the resources monitored by the PURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the PURR are implementation-specific.

The PURR is implemented such that:

1. Loading a GPR from the PURR has no effect on the accuracy of the PURR.
2. Copying the contents of a GPR to the PURR replaces the contents of the PURR with the contents of the GPR.

Programming Note

Estimates computed as described above may be useful for purposes related to resource utilization, including utilization-based system management and planning.

Because the rate at which the PURR accumulates resource usage estimates is dependent on the frequency at which the Time Base is incremented, and the frequency of the oscillator that drives instruction execution may vary independently from that of the Time Base, the interpretation of the contents of the PURR may be inaccurate as a measurement of capacity consumption for accounting purposes. The SPURR should be used for accounting purposes.

7.7 Scaled Processor Utilization of Resources Register (SPURR)

The Scaled Processor Utilization of Resources Register (SPURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the SPURR are treated as a 64-bit unsigned integer.

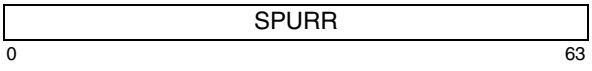


Figure 58. Scaled Processor Utilization of Resources Register

The SPURR is a hypervisor resource; see Section 2.7.

The contents of the SPURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1) at which point the contents are replaced by that sum modulo 2^{64}. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the SPURR increases is an estimate of the portion of resources used by the thread with respect to other threads that share those resources monitored by the SPURR, and relative to the computational capacity provided by those resources. The computational capacity provided by the shared resources may vary as a function of the frequency of the oscillator which drives the resources or as a result of deliberate delays in processing that are created to reduce power consumption. When the thread is idle, the rate at which the SPURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be T_{ab}. Let the ratio of the effective and nominal frequencies of the oscillator driving instruction execution f_e/f_n be f_r. Let the ratio of delay cycles created by power reduction circuitry and total cycles c_d/c_t be c_r. Let the difference between the value represented by the contents of the SPURR at time T_a and T_b be the value S_{ab}. The ratio of $S_{ab}/(T_{ab} \times f_r \times (1 - c_r))$ is an estimate of the percentage of shared resource capacity used by the thread during the interval T_{ab}. For the set {S} of threads that share the resources monitored by the SPURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the SPURR are implementation-specific.

The SPURR is implemented such that:

1. Loading a GPR from the SPURR has no effect on the accuracy of the SPURR.
2. Copying the contents of a GPR to the SPURR replaces the contents of the SPURR with the contents of the GPR.

Programming Note

Estimates computed as described above may be useful for purposes of resource use accounting, program dispatching, etc.

7.8 Instruction Counter

The Instruction Counter (IC) is a 64-bit incrementing counter that counts the number of instructions that the thread has completed (according to the sequential execution model; see Chapter 2.2 of Book I).

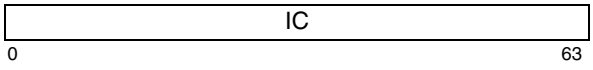


Figure 59. Instruction Counter

Chapter 8. Debug Facilities

8.1 Overview

Implementations provide debug facilities to enable hardware and software debug functions, such as control flow tracing, data address watchpoints, and program single-stepping. The debug facilities described in this section consist of the Come-From Address Register (see Section 8.1.1), Completed Instruction Address Breakpoint Register (see Section 8.1.2), and the Data Address Watchpoint Register (DAWRn) and Data Address Watchpoint Register Extension (DAWRXn) (see Section 8.1.3). The interrupt associated with the Data Address Breakpoint registers is described in Section 6.5.3. The interrupt associated with the Completed Instruction Address Breakpoint Register is described in Section 6.5.15. The Trace facility, which can be used for single-stepping as well as for control flow tracing, is described in Section 6.5.15.

The ***mfspr*** and ***mtspr*** instructions (see Section 4.4.4) provide access to the registers of the debug facilities.

In addition to the facilities mentioned above, implementations typically provide debug facilities, modes, and access mechanisms that are implementation-specific. For example, implementations typically provide facilities for instruction address tracing, and also access to certain debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

8.1.1 Come-From Address Register

The Come-From Address Register (CFAR) is a 64-bit register. When an ***rfebb***, ***rfid***, instruction is executed, the register is set to the effective address of the instruction. When a *Branch* instruction is executed and the branch is taken, the register is set to the effective address of an instruction in the instruction cache block containing the *Branch* instruction, except that if the *Branch* instruction is a B-form *Branch* (i.e. ***bc***, ***bca***, ***bcl***, or ***bcla***) for which the target address is in the instruction cache block containing the *Branch* instruction or is in the previous or next cache block, the register is not necessarily set. For *Branch* instructions, the setting need not occur until a subsequent context synchronizing operation has occurred.

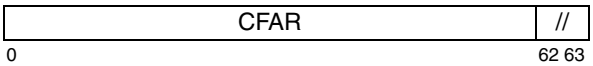


Figure 60. Come-From Address Register

The contents of the CFAR can be read and written using the ***mfspr*** and ***mtspr*** instructions. Acccess to the CFAR is privileged.

Programming Note

This register can be used for purposes of debugging software. For example, often a software bug results in the program executing a portion of the code that it should not have reached or causing an unexpected interrupt. In the former case, a breakpoint can be placed in the portion of the code that was erroneously reached and the program reexecuted. In either case, the interrupt handler can save the contents of the CFAR (before executing the first instruction that would modify the register), and then make the saved contents available for a debugger to use in determining the control flow path by which the exception was reached.

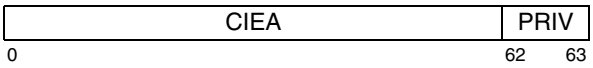
In order to preserve the CFAR's contents for each partition and to prevent it from being used to implement a "covert channel" between partitions, the hypervisor should initialize/save/restore the CFAR when switching partitions on a given thread.

8.1.2 Completed Instruction Address Breakpoint [Category: Trace]

The Completed Instruction Address Breakpoint mechanism provides a means of detecting an instruction completion at a specific instruction address. The address comparison is done on an effective address (EA).

The Completed Instruction Address Breakpoint mechanism is controlled by the Completed Instruction

Address Breakpoint Register (CIABR), shown in Figure 62.



Bit(s)	Name	Description
0:61	CIEA	Completed Instruction Effective Address
62:63	PRIV	Privilege 00: Disable matching 01: Match in problem state 10: Match in privileged (non-hypervisor) state 11: Match in hypervisor state

Figure 61. Completed Instruction Address Breakpoint Register

A Completed Instruction Address Breakpoint match occurs upon instruction completion if all of the following conditions are satisfied.

- the completed instruction address is equal to $CIEA_{0:61}$ || 0b00.
- the thread run level matches that specified in RLM.

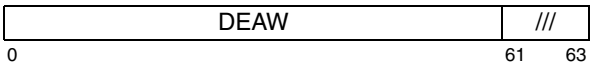
In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A Completed Instruction Address Breakpoint match causes a Trace exception provided that no higher priority interrupt occurs from the completion of the instruction (see Section 6.5.15).

8.1.3 Data Address Watchpoint

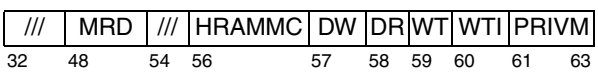
The Data Address Watchpoint mechanism provides a means of detecting load and store accesses to a range of addresses starting at a designated doubleword. The address comparison is done on an effective address (EA).

The Data Address Watchpoint mechanism is controlled by a single set of SPRs, numbered with n=0: the Data Address Watchpoint Register (DAWRn), shown in Figure 62, and the Data Address Watchpoint Register Extension (DAWRXn), shown in Figure 63.



Bit(s)	Name	Description
0:60	DEAW	Data Effective Address Watchpoint

Figure 62. Data Address Watchpoint Register



Bit(s)	Name	Description
48:53	MRD	Match Range in Doublewords biased by -1. (0b000000 = 1 DW, 0b111111 = 64 DW)
56	HRAMMC	Hypervisor Real Addressing Mode Match Control 0: $DEAW_0$ and EA_0 are used during matching in hypervisor real addressing mode 1: $DEAW_0$ and EA_0 are ignored during matching in hypervisor real addressing mode
57	DW	Data Write
58	DR	Data Read
59	WT	Watchpoint Translation
60	WTI	Watchpoint Translation Ignore
61:63	PRIVM	Privilege Mask
61	HYP	Hypervisor state
62	PNH	Privileged but Non-Hypervisor state
63	PRO	Problem state

All other fields are reserved.

Figure 63. Data Address Watchpoint Register Extension

The supported PRIVM values are 0b000, 0b001, 0b010, 0b011, 0b100, and 0b111. If the PRIVM field does not contain one of the supported values, then whether a match occurs for a given storage access is undefined. Elsewhere in this section it is assumed that the PRIVM field contains one of the supported values.

Architecture Note

Future versions of the architecture may allow n to be nonzero, thus providing more Data Address Watchpoints. The SPR numbers for DAWRn and DAWRXn have been defined to support this possible future expansion.

Programming Note

PRIVM value 0b000 causes matches not to occur regardless of the contents of other DAWRn and DAWRXn fields. PRIVM values 0b101 and 0b110 are not supported because a storage location that is shared between the hypervisor and non-hypervisor software is unlikely to be accessed using the same EA by both the hypervisor and the non-hypervisor software. (PRIVM value 0b111 is supported primarily for reasons of software compatibility with respect to emulation of the DABR facility as described in a subsequent Programming Note.)

A Data Address Watchpoint match occurs for a Load or Store instruction if, for any byte accessed, all of the following conditions are satisfied.

- the access is

  - a quadword access and located in the range $(DEAW_{0:59} \| 0) \le (EA_{0:59} \| 0) \le ((DEAW_{0:59} \| 0) + (^{55}0 \| MRD_{0:4}))$ such that $(EA_{0:60}$ AND $(^{55}1 \| {}^{6}0)) = (DEAW_{0:60}$ AND $(^{55}1 \| {}^{6}0))$.
  - not a quadword access and located in the range $DEAW_{0:60} \le EA_{0:60} \le (DEAW_{0:60} + (^{55}0 \| MRD_{0:5}))$ such that $(EA_{0:60}$ AND $(^{55}1 \| {}^{6}0)) = (DEAW_{0:60}$ AND $(^{55}1 \| {}^{6}0))$.
- $(MSR_{DR} = DAWRXn_{WT}) \mid DAWRXn_{WTI}$
- the thread is in
  - hypervisor state and $DAWRXn_{HYP} = 1$, or
  - privileged but non-hypervisor state and $DAWRXn_{PNH} = 1$, or
  - problem state and $DAWRXn_{PR} = 1$
- the instruction is a *Store* and $DAWRXn_{DW} = 1$, or the instruction is a *Load* and $DAWRXn_{DR} = 1$.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

If the above conditions are satisfied, a match also occurs for ***eciwx*** and ***ecowx***. For the purpose of determining whether a match occurs, ***eciwx*** is treated as a *Load*, and ***ecowx*** is treated as a *Store*.

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is *Store Conditional* but the store is not performed
- The instruction is ***dcbz***. (For the purpose of determining whether a match occurs, ***dcbz*** is treated as a *Store*.)

The *Cache Management* instructions other than ***dcbz*** never cause a match.

A Data Address Watchpoint match causes a Data Storage exception or a Hypervisor Data Storage exception (see Section 6.5.3, “Data Storage Interrupt” on page 955 and Section 6.5.16, “Hypervisor Data Storage Interrupt” on page 963). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a *Store* or ***ecowx*** instruction causes the match, the storage operand is not modified if the instruction is one of the following:

- any *Store* instruction that causes an atomic access
- ***ecowx***

Architecture Note

A Data Address Watchpoint match will not cross a 512 byte boundary. This is governed by the size of the MRD field which is 6 bits: $2^{6+3} = 512$ bytes. The definition of the MRD field supports that future versions of the architecture may increase the size of the MRD field, thus increasing the match range and also the granularity of the boundary that is not crossed.

Architecture Note

For a quadword access, it is possible that the address reported in the DAR/HDAR can point to a byte outside of the range of the watchpoint if the starting address of the watchpoint is not quadword aligned.

Programming Note

The Data Address Watchpoint mechanism does not apply to instruction fetches.

Programming Note

Implementations that comply with versions of the architecture that precede Version 2.02 do not provide the DABRX (now replaced by DAWRXn). Forward compatibility for software that was written for such implementations (and uses the Data Address Breakpoint facility) can be obtained by setting $DAWRXn_{60:63}$ to 0b0111.

Chapter 9. Performance Monitor Facility

9.1 Overview

The Performance Monitor facility provides a means of collecting information about program and system performance.

9.2 Performance Monitor Operation

The Performance Monitor facility is controlled by the following features (described in detail in subsequent sections).

- an MSR bit
  - PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring
- registers
  - PMC1 - PMC6 (Performance Monitor Counters 1 - 6), which count events
  - MMCR0, MMCR1, MMCR2, and MMCRA (Monitor Mode Control Registers 0, 1, 2, and A), which control the Performance Monitor facility
  - SIAR, SDAR, and SIER (Sampled Instruction Address Register, Sampled Data Address Register, and Sampled Instruction Event Register), which contain the address of the "sampled instruction" and of the "sampled data," and additional information about the "sampled instruction" (see Section 9.4.8 - Section 9.4.10).
- The Branch History Rolling Buffer (BHRB), which is a buffer that contains the target addresses of most recent branch instructions for which the branch was taken.
- the performance monitor interrupt, which can be caused by monitored conditions and events.

Many aspects of the operation of the Performance Monitor are summarized by the following hierarchy, which is described starting at the lowest level.

- A "counter negative condition" exists when the value in a PMC is negative (i.e., when bit 0 of the PMC is 1). A "Time Base transition event" occurs when a selected bit of the Time Base changes from 0 to 1 (the bit is selected by a field in MMCR0). The term "condition or event" is used as an abbreviation for "counter negative condition or Time Base transition event". A condition or event can be caused implicitly by the hardware (e.g., incrementing a PMC) or explicitly by software (***mtspr***).
- A condition or event is enabled if the corresponding "Enable" bit (i.e. PMC1CE, PMCjCE, or TBEE) in MMCR0 is 1. The occurrence of an enabled condition or event can have side effects within the Performance Monitor, such as causing the PMCs to cease counting.
- An enabled condition or event causes a Performance Monitor alert if Performance Monitor alerts are enabled by the corresponding "Enable" bit in MMCR0. A single Performance Monitor alert may reflect multiple enabled conditions and events.
- When a Performance Monitor alert occurs, $MMCR0_{PMAO}$ is set to 1 and the writing of BHRB entries, if in process, is suspended.

  When the contents of $MMCR0_{PMAO}$ change from 0 to 1, a Perfrormance Monitor exception will come into existence within a reasonable period of time. When the contents of $MMCR0_{PMAO}$ change from 1 to 0, the existing Performance Monitor exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.
- A Performance Monitor exception causes one of the following.
  - If MSR_{EE} = 1 and $MMCR0_{EBE}$ = 0, an interrupt occurs.
  - If MSR_{PR} = 1, $MMCR0_{EBE}$ = 1, a Performance Monitor event-based exception occurs if $BESCR_{PME}$=1, provided $FSCR_{EBB}$ and $HFSCR_{EBB}$ are enabled. When a Performance Monitor event-based exception occurs,

an event-based branch is generated if BESCR$_{GE}$=1.

Programming Note

The Performance Monitor can be effectively disabled (i.e., put into a state in which Performance Monitor SPRs are not altered and Performance Monitor exceptions do not occur) by setting MMCR0 to 0x0000_0000_8000_0000.

9.3 Probe No-op Instruction

A probe no-op is an ***and*** *0,0,0* instruction. This form of ***and*** is reserved for use exclusively by the Performance Monitor.

Programming Note

Software can insert *probe no-op* instructions at various points in a program and configure the performance monitor such that the instruction is eligible for sampling. See Section 9.4.7.

9.4 Performance Monitor Facility Registers

The Performance Monitor registers count events, control the operation of the Performance Monitor, and provide associated information.

The elapsed time between the execution of an instruction and the time at which events due to that instruction have been reflected in Performance Monitor registers is not defined. No means are provided by which software can ensure that all events due to preceding instructions have been reflected in Performance Monitor registers. Similarly, if the events being monitored may be caused by operations that are performed out-of-order, no means are provided by which software can prevent such events due to subsequent instructions from being reflected in Performance Monitor registers. Thus the contents obtained by reading a Performance Monitor register may not be precise: it may fail to reflect some events due to instructions that precede the ***mfspr*** and may reflect some events due to instructions that follow the ***mfspr***. This lack of precision applies regardless of whether the state of the thread is such that the register is subject to change by the hardware at the time the ***mfspr*** is executed. Similarly, if an ***mtspr*** instruction is executed that changes the contents of the Time Base, the change is not guaranteed to have taken effect with respect to causing Time Base transition events until after a subsequent context synchronizing instruction has been executed.

If an ***mtspr*** instruction is executed that changes the value of a Performance Monitor register other than SIAR, SDAR, and SIER, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 12. “Synchronization Requirements for Context Alterations” on page 1009).

Programming Note

Depending on the events being monitored, the contents of Performance Monitor registers may be affected by aspects of the runtime environment (e.g., cache contents) that are not directly attributable to the programs being monitored.

9.4.1 Performance Monitor SPR Numbers

The Performance Monitor registers have two sets of SPR numbers, one set that is non-privileged and another set that is privileged.

For the purpose of explanation elsewhere in the architecture, the non-privileged registers are divided into two groups as defined below.

- A: The non-privileged read/write Performance Monitor registers (i.e. the PMCs, MMCR0, MMCR2, and MMCRA at SPR numbers 771-776, 779, 769, and 770, respectively)
- B: The non-privileged read-only Performance Monitor registers (i.e. SIER, SIAR, SDAR, and MMCR1 at SPR numbers 768, 780, 781, and 782, respectively).

The SPRs in group B are treated as not implemented registers for write (***mtspr***) operations. See the ***mtspr*** instruction description in Section 4.4.4 for additional information.

When the PCR makes a register in either group A or B unavailable in problem state, that SPR is not included in group A or B.

9.4.2 Performance Monitor Counters

The six Performance Monitor Counters, PMC1 through PMC6, are 32-bit registers that count events.

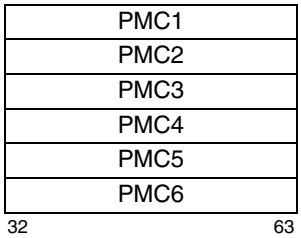


Figure 64. Performance Monitor Counter registers

PMC1 - PMC4 are referred to as “programmable” counters since the events that can be counted can be

specified by the program. The codes that identify the events that are counted are selected by specifying the appropriate code in “PMCn Selector” fields in MMCR1. Some events may include operations that are performed out-of-order.

PMC5 and PMC6 are not programmable. PMC5 counts instructions completed and PMC6 counts cycles. The PMCC field in MMCR0 control whether or not PMCs 5-6 are under the control of various bits in MMCR0 and MMCR2. When PMCs 5-6 are not under the control of these bits, they do not cause performance monitor events.

Normally each PMC is incremented each hardware cycle by the number of times the corresponding event occurred in that cycle. Other modes of incrementing may also be provided.

Programming Note

PMC5 and PMC6 are defined to facilitate calculating basic performance metrics such as cycles per instruction (CPI).

Programming Note

Software can use a PMC to “pace” the collection of Performance Monitor data. For example, if it is desired to collect event counts every n cycles, software can specify that a particular PMC count cycles and set that PMC to 0x8000_0000 - n. The events of interest would be counted in other PMCs. The counter negative condition that will occur after n cycles can, with the appropriate setting of MMCR bits, cause counter values to become frozen, cause a Performance Monitor exception to occur, etc.

9.4.3 Threshold Event Counter

The threshold event counter and associated controls are in MMCRA (see Section 9.4.7). When the Performance Monitor is enabled ($MMCR0_{PMAE}$=1), this counter begins incrementing from value 0 upon each occurrence of the event specified in the Threshold Event (TEV) field after recognizing the event specified by the Threshold Start (TS) field. The counter stops incrementing when the event specified in the Threshold End (TE) field occurs. The counter subsequently freezes until the event specified in the TS is again recognized, at which point it increments as explained above. Incrementing stops when a Performance Monitor Alert occurs. After the Performance Monitor alert occurs, the contents of SIAR are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1.

Programming Note

Because hardware can modify the contents of the threshold event counter when random sampling is enabled ($MMCRA_{SE}$=1) and $MMCR0_{PMAE}$=1 at any time, any value written to the threshold event counter under this condition may be immediately overwritten by hardware.

The threshold counter value is represented as a base-4 floating-point number. The mantissa of the floating-point number is contained in $MMCRA_{TECM}$, and the exponent contained in $MMCRA_{TECX}$. For a given counter mantissa, m, and exponent, e, the number represented is as follows:

$$N = 4^e * m$$

This counter format allows the counter to represent a range of 0 through approximately 2 M counts with many fewer bits than would be required by a binary counter.

Programming Note

Software can obtain the number N from contents of the threshold counter by shifting the mantissa left twice times the value contained in the exponent .

The value in the counter is the exact number of events that occur for values from 0 through the maximum mantissa value (127), within 4 events of the exact value for values from 128 - 508 (or 127*4), within 16 events of the exact value for values from 512 - 2032 (or $127*4^2$), and so on. This represents an event count accuracy of approximately 3%, which is expected to be sufficient for most situations in which a count of events between a start and end event is required.

Programming Note

When using the threshold counter, software typically specifies a “threshold counter exceeded n” event in MMCR1. This enables a PMC to count the number of times the counter exceeded a specified threshold value during the time Performance Monitor alerts were enabled.

9.4.4 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 64-bit register as shown below.

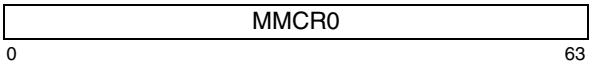


Figure 65. Monitor Mode Control Register 0

Some bits of MMCR0 are altered by the hardware when various events occur, and some bits may be altered by software.

The following notation is used in the definitions below. When $MMCR0_{PMCC}$ = 11, "PMCs" refers to PMCs 1 - 4 and "PMCj" or PMCjCE refers to "PMCj or PMCjCE, respectively, where j = 2 - 4; otherwise, "PMCs" refers to PMCs 1 - 6 and "PMCj" or PMCjCE refers to "PMCj or PMCjCE, respectively, where j = 2 - 6.

When the $MMCR0_{PMCC}$ is set to 10 or 11, providing problem state programs read/write access, only FC, PMAE, PMAO, can be accessed. All other bits are not changed when ***mtspr*** is executed in problem state, and all other bits return 0s when ***mfspr*** is executed.

Programming Note

When PMCC=10 or 11, problem state programs have access to MMCR0 in order to enable event-based branch routines to reset the FC bit after it has been set to 1 as a result of an enabled condition or event (FCECE=1.) During event processing, the event-based branch handler would write the desired initial values to the PMCs and reset the FC bit to 0. PMAO and PMAE can also be set to their appropriate values during the same write operation before returning.

If additional event-based branches are defined in the future, it may be desirable to set PMAE and PMAO using the BESCRS since additional bits in BESCRS will also need to be written.

Bit(s) Description

0:31 Reserved

32 ***Freeze Counters*** (FC)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented.

The hardware sets this bit to 1 when an enabled condition or event occurs and $MMCR0_{FCECE}$=1.

33 ***Freeze Counters and BHRB in Privileged State*** (FCS)

0 The PMCs are incremented (if permitted by other MMCR bits), and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
1 The PMCs are not incremented and entries are not written into the BHRB if $MSR_{HV\ PR}$=0b00.

34 ***Conditionally Freeze Counters and BHRB in Problem State*** (FCP)

If the value of bit 51 (FCPC) is 0, this field has the following meaning.

0 The PMCs are incremented (if permitted by other MMCR bits) and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
1 The PMCs are not incremented and entries are not written into the BHRB if MSR_{PR}=1.

If the value of bit 51 (FCPC) is 1, this field has the following meaning.

0 The PMCs are not incremented and entries are not written into the BHRB if $MSR_{HV\ PR}$=0b01.
1 The PMCs are not incremented and BHRB entries are not written if $MSR_{HV\ PR}$=0b11.

Programming Note

In order to freeze counters in problem state regardless of MSR_{HV}, $MMCR0_{FCPC}$ must be set to 0 and $MMCR0_{FCP}$ must be set to 1.

35 ***Freeze Counters while Mark = 1*** (FCM1)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PMM}=1.

36 ***Freeze Counters while Mark = 0*** (FCM0)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PMM}=0.

37 ***Performance Monitor Alert Enable*** (PMAE)

0 Performance Monitor alerts are disabled and BHRB entries are not written.
1 Performance Monitor alerts are enabled and BHRB entries are written (if enabled by other bits) until a Performance Monitor alert occurs, at which time:
- $MMCR0_{PMAE}$ is set to 0
- $MMCR0_{PMAO}$ is set to 1

Programming Note

Software can set this bit and $MMCR0_{PMAO}$ to 0 to prevent Performance Monitor exceptions.

Software can set this bit to 1 and then poll the bit to determine whether an enabled condition or event has occurred. This is especially useful for software that runs with MSR_{EE}=0.

In earlier versions of the architecture that lacked the concept of Performance Monitor alerts, this bit was called Performance Monitor Exception Enable (PMXE).

38 ***Freeze Counters on Enabled Condition or Event*** (FCECE)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when $MMCR0_{TRIGGER}$=0, at which time:
- $MMCR0_{FC}$ is set to 1

If the enabled condition or event occurs when $MMCR0_{TRIGGER}$=1, the FCECE bit is treated as if it were 0.

39:40 ***Time Base Selector*** (TBSEL)

This field selects the Time Base bit that can cause a Time Base transition event (the event occurs when the selected bit changes from 0 to 1).

00 Time Base bit 63 is selected.
01 Time Base bit 55 is selected.
10 Time Base bit 51 is selected.
11 Time Base bit 47 is selected.

Programming Note

Time Base transition events can be used to collect information about activity, as revealed by event counts in PMCs and by addresses in SIAR and SDAR, at periodic intervals.

In multi-threaded systems in which the Time Base registers are synchronized among the threads, Time Base transition events can be used to correlate the Performance Monitor data obtained by the several threads. For this use, software must specify the same TBSEL value for all the threads in the system.

Because the frequency of the Time Base is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.

41 ***Time Base Event Enable*** (TBEE)

0 Time Base transition events are disabled.
1 Time Base transition events are enabled.

Programming Note

When PMC3 is configured to count the occurrence of Time Base transition events, the events are counted regardless of the value of $MMCR0_{TBEE}$. (See Section 9.4.5.) The occurrence of a Time Base transition causes a Performance Monitor alert only if $MMCR0_{TBEE}$=1.

42 **BHRB Access (BHRBA)**
This field controls whether the BHRB instructions are available in problem state. If an attempt is made to execute a BHRB instruction is made in problem state when the BHRB instructions are not available, a Facility Unavailable interrupt will occur.

0 ***mfbhrb*** and ***clrbhrb*** are not available in problem state.
1 ***mfbhrb*** and ***clrbhrb*** are available in problem state unless they have been made unavailable by some other register.

43 **Performance Monitor Event-Based Branch Enable (EBE)**
This field controls whether Performance Monitor event-based branches are enabled.

When Performance Monitor event-based branches are disabled, no Performance Monitor event-based branches occur regardless of the state of $BESCR_{PME}$.
0 Performance Monitor event-based branches are disabled.
1 Performance Monitor event-based branches are enabled.

Programming Note

In order to enable a problem state applications to use the Event-Based Branch facility for Performance Monitor events, privileged software initializes MMCR1 to specify the events to be counted, and sets MMCR2, and MMCRA to specify additional sampling controls. MMCR0 should be initialized with PMCC set to 10 or 11 (to give problem state access to various Performance Monitor registers), PMAE and PMAO set to 0s (disabling Performance Monitor alerts), and EBE set to 1 (enabling event-based branches to occur). If the Event-Based Branch facility has not been enabled in the FSCR and HFSCR, it must be enabled in these registers as well.

The above operations by the operating system enable the application to control Performance Monitor event-based branching by means of $BESCR_{PME}$ (to enable or disable Performance Monitor event-based branching) and $MMCR0_{PMAE}$ (to enable or disable Performance Monitor alerts). The requirement to set both $BESCR_{PME}$ and $MMCR0_{PMAE}$ is not expected to impact application performance since $MMCR0_{PMAE}$ as well as other MMCR0 bits that need to be set can be done in a single ***mtspr*** instruction.

44:45 **PMC Control (PMCC)**

This field controls whether or not PMCs 5 - 6 are included in the Performance Monitor, and the accessibility of groups A and B (see Section 9.4.1) of non-privileged SPRs in problem state as described below.

I

Programming Note

The PMCC field does not affect the behavior of the privileged Performance Monitor Registers (SPRs 784-792, 795-798); accesses to these SPRs in problem state result in Privileged Instruction Type Program interrupts.

The PMCC field also does not affect the behavior of write operations to Group B; write operations to SPRs in Group B are treated as not supported regardless of privilege state. See the ***mtspr*** instruction description in Section 4.4.4 for additional information on accessing SPRs that are not supported.

Programming Note

When the PCR makes SPRs unavailable in problem state, they are treated as not implemented, and they are not included in groups A or B regardless of the value of PMCC. Thus when the PCR indicates a version of the architecure prior to V 2.07 (i.e. $PCR_{V2.06}$=1), the PMCC field does not affect the newly-defined SPRs MMCR2 or SIER; these SPRs are treated as unimplemented registers. Accesses to them in problem state result in Hypervisor Emulation Assistance interrupts regardless of the value of PMCC, and Facility Unavailable interrupts do not occur. See Section 2.6 for additional information.

00 PMCs 5 - 6 are included in the Performance Monitor.
Group A is read-only in problem state. If an attempt is made to write to an SPR in group A in problem state, a Hypervisor Emulation Assistance interrupt will occur.

01 PMCs 5 - 6 are included in the Performance Monitor.
Group A is not allowed to be read or written in problem state, and group B is not allowed to be read. If an attempt is made to read or write to an SPR in group A, or to read from an SPR in group B in problem state, a Facility Unavailable interrupt will occur.

10 PMCs 5 - 6 are included in the Performance Monitor.
Group A is allowed to be read and written in problem state, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state. If an attempt is made to read MMCR1 in problem state, a Facility Unavailable interrupt will occur.

11 PMCs 5 - 6 are not included in the Performance Monitor. See Section 9.4.2 for details.
Group A except for PMCs 5-6 (SPRs 775,776) is allowed to be read and written, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state.
If an attempt is made to read or write to PMCs 5-6 (SPRs 775,776), or to read from MMCR1 in problem state, a Facility Unavailable interrupt will occur.

When an SPR is made available by the PMCC field, it is available only if it has not been made unavailable by some other register.

Programming Note

In order to give problem-state programs the same level of access to the Performance Monitor registers as was specified in Power ISA V 2.06, PMCC must be set to 0b'00' (restricting access to read only) and the PCR should indicate Version 2.06 (restricting access to the set of Performance Monitor SPRs and SPR bits that were defined in V 2.06).

When PMCC=00 and a write operation to a Performance Monitor register in group A or B is attempted in problem state, a Hypervisor Emulation Assistance interrupt occurs in order to maintain compatibility with V 2.06. For other values of PMCC, write or read operations to group A and read operations from group B that are not allowed result in Facility Unavailable interrupts. Facility Unavailable interrupts provide the operating system with more information about the type of disallowed access that was performed than the Hypervisor Emulation Assistance interrupt provides. See Section 6.2.10 for additional information.

46 ***Freeze Counters in Transactional State*** (FCTS)

0 PMCs are incremented (if permitted by other MMCR bits).

1 PMCs are not incremented when Transactional Memory is in transactional state.

47 ***Freeze Counters in Non-Transactional State*** (FCNTS)

0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when Transactional Memory is in non-transactional state.

48 ***PMC1 Condition Enable*** (PMC1CE)

This bit controls whether counter negative conditions due to a negative value in PMC1 are enabled.

0 Counter negative conditions for PMC1 are disabled.
1 Counter negative conditions for PMC1 are enabled.

49 ***PMCj Condition Enable*** (PMCjCE)

This bit controls whether counter negative conditions due to a negative value in any PMCj (i.e., in any PMC except PMC1) are enabled.

0 Counter negative conditions for all PMCjs except those enabled by PMCnCE, are disabled.
1 Counter negative conditions for all PMCjs are enabled.

50 ***Trigger*** (TRIGGER)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 PMC1 is incremented (if permitted by other MMCR bits). The PMCjs are not incremented until PMC1 is negative or an enabled condition or event occurs, at which time:
- the PMCjs resume incrementing (if permitted by other MMCR bits)
- $MMCR0_{TRIGGER}$ is set to 0

See the description of the FCECE bit, above, regarding the interaction between TRIGGER and FCECE.

Programming Note

Uses of TRIGGER include the following.

- Resume counting in the PMCjs when PMC1 becomes negative, without causing a Performance Monitor interrupt. Then freeze all PMCs (and optionally cause a Performance Monitor interrupt) when a PMCj becomes negative. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time a PMCj becomes negative. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=0
  - PMCjCE=1
  - TBEE=0
  - FCECE=1
  - PMAE=1 (if a Performance Monitor interrupt is desired)
- Resume counting in the PMCjs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=1
  - TBEE=0
  - FCECE=0
  - PMAE=1

51 ***Freeze Counters and BHRB in Problem State Condition*** (FCPC)

This bit controls the operation of bit 34 (FCP). See the definition of bit 34 for details.

Programming Note

In order to enable the FCP bit to freeze counters in problem state regardless of MSR_{HV}, $MMCR0_{FCPC}$ must be set to 0.

52:54 ***Performance Monitor Alert Qualifier*** (PMAOQ)

These bits provide additional implementation-dependent information about the cause of the Performance Monitor Alert. These bits must be set to 0s when no additional information is available.

55 ***Control Counters 5 and 6 With Run Latch*** (CC56RUN)

When $MMCR0_{PMCC}$ = b'11', the setting of this bit has no effect; otherwise it is defined as follows.

0 PMCs 5 and 6 are incremented if $CTRL_{RUN}$=1 if permitted by other MMCR bits.
1 PMCs 5 and 6 are incremented, regardless of the value of $CTRL_{RUN}$, if permitted by other MMCR bits.

56 ***Performance Monitor Alert Occurred*** (PMAO)

0 A Performance Monitor alert has not occurred since the last time software set this bit to 0.
1 A Performance Monitor alert has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor alert occurs. This bit can be set to 0 only by the ***mtspr*** instruction.

Programming Note

Software can set this bit to 1 and set PMAE to 0 to simulate the occurrence of a Performance Monitor alert.

Software should set this bit to 0 after handling the Performance Monitor alert.

57 ***Freeze Counters in Suspended State*** (FCSS)

0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when Transactional Memory is in suspended state.

58 ***Freeze Counters 1-4*** (FC1-4)

0 PMC1 - PMC4 are incremented (if permitted by other MMCR bits).
1 PMC1 - PMC4 are not incremented.

59 ***Freeze Counters 5-6*** (FC5-6)

0 PMC5 - PMC6 are incremented (if permitted by other MMCR bits).
1 PMC5 - PMC6 are not incremented.

60:61 Reserved

62 ***Freeze Counters 1-4 in Wait State*** (FC14WAIT)

0 PMCs 1-4 are incremented (if permitted by other MMCR bits).
1 PMCs 1-4, except for PMCs counting events that are not controlled by this bit, are not incremented if $CTRL_{RUN}$=0.

63 ***Freeze Counters and BHRB in Hypervisor State*** (FCH)

0 The PMCs are incremented (if permitted by other MMCR bits) and BHRB entries are written (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
1 The PMCs are not incremented and BHRB entries are not written if $MSR_{HV\ PR}$=0b10.

9.4.5 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register as shown below.

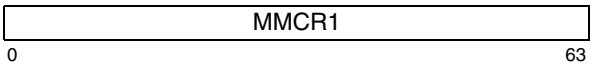


Figure 66. Monitor Mode Control Register 1

The bit definitions of MMCR1 are as follows. Implementation dependent MMCR1 bits that are not supported are treated as reserved.

In the following descriptions, events due to *randomly sampled* instructions only occur only if random sampling is enabled ($MMCRA_{SE}$=1); all other events occur whenever the event specification is met regardless of the value of $MMCR0_{SE}$.

Various events defined below refer to "threshold A" through "threshold H". The table below specifies the number of threshold counter events corresponding to each of these thresholds.

Threshold	Events
A	16
B	32
C	64
D	128
E	256
F	512
G	1024
H	2048

Table 3: Event Counts for Thesholds A-H

Bit(s)	Description
0:31	Problem state access (SPR 782) Reserved Privileged access (SPR 782 or 798) Implementation-dependent

32:39 ***PMC1 Selector*** (PMC1SEL)
The value of PMC1SEL specifies the event to be counted by PMC 1 as defined below.
All values in the range of C0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01-BF	Implementation dependent
C0-DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

- E0 The thread has dispatched a randomly sampled instruction. (RIS)
- E2 The thread has completed a randomly sampled *Branch* instruction for which the branch was taken.(RIS, RBS)
- E4 The thread has failed to locate a randomly sampled instruction in the primary cache. (RIS)
- E6 The threshold event counter has exceeded the number of events corresponding to threshold A (see Table 3). (RIS, RLS, RBS)
- E8 The threshold event counter has exceeded the number of events corresponding to threshold E (see Table 3). (RIS, RLS, RBS)
- EA The thread filled a block in a cache containing data accessed by a randomly sampled *Load* instruction. (RIS, RLS)
- EC The threshold event counter has reached its maximum value when random sampling is enabled. (RIS, RLS, RBS)

- F0 A cycle has occurred. This event is not controlled by $MMCR0_{FC14WAIT}$.
- F2 A cycle has occurred in which the thread completed one or more instructions.
- F4 The thread has completed a *Floating Point, Vector Floating Point,* or *VSX Floating Point* instruction other than a load or store operation up to the point at which it has reported all exceptions that it will cause.
- F6 The thread has failed to locate an ERAT entry during instruction address translation.
- F8 A cycle has occurred in which all previously initiated instructions have completed and no instructions are available for initiation.
- FA A cycle has occurred during which the RUN bit of the control register for one or more threads was set to 1.

40:47 ***PMC2 Selector*** (PMC2SEL)
The value of PMC2SEL specifies the event to be counted by PMC 2 as defined below.
All values in the range of C0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01-BF	Implementation dependent
C0-DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

- E0 The thread located the data for a randomly sampled *Load* instruction from storage that did not reside in any cache. (RIS, RLS)
- E2 The thread has failed to locate the data for a randomly sampled *Load* instruction in the primary data cache. (RIS, RLS)
- E4 The thread filled a block in the primary cache containing data accessed by a randomly sampled *Load* instruction from a location other than the secondary or tertiary cache. (RIS, RLS)
- E6 The threshold event counter has exceeded the number of events corresponding to threshold B (see Table 3). (RIS, RLS, RBS)
- E6 The threshold event counter has exceeded the number of events corresponding to threshold F (see Table 3). (RIS, RLS, RBS)

- F0 The thread has completed a *Store* instruction to the point at which it has caused all the exceptions that it will cause.
- F2 The thread has dispatched an instruction.
- F4 A cycle has occurred during which the RUN bit of the thread's CTRL register was set to 1.
- F6 The thread has failed to locate an ERAT entry during data address translation, and a new ERAT entry corresponding to the data address has been written.
- F8 An external interrupt for the thread has occurred.
- FA The thread has completed a *Branch* instruction for which the branch was taken.
- FC The thread has failed to locate an instruction in the primary cache.
- FE The thread has filled a block in the primary cache containing data accessed by a *Load* instruction from a location other than the secondary cache.

48:55 ***PMC3Selector*** (PMC3SEL)
The value of PMC3SEL specifies the event to be counted by PMC 3 as defined below.
All values in the range of C0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01-BF	Implementation dependent
C0-DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

- E2 The thread has completed a randomly sampled *Store* instruction up to the point at which it has reported all exceptions that it will cause. (RIS)
- E4 The thread has incorrectly predicted the outcome of a randomly sampled *Branch* instruction. (RIS, RBS)
- E6 The thread has failed to locate a data ERAT entry for a randomly sampled instruction. (RIS)
- E8 The threshold event counter has exceeded the number of events corresponding to threshold C (see Table 3). (RIS, RLS, RBS)
- EA The threshold event counter has exceeded the number of events corresponding to threshold G (see Table 3). (RIS, RLS, RBS)

- F0 The thread has attempted to store data in the primary cache but the block containing the effective address did not exist.
- F2 The thread has dispatched an instruction.
- F4 The thread has completed an instruction during a time when the RUN bit of the CTRL register for all enabled threads on the multi-threaded processor is set to 1.
- F6 The thread has filled a block in the primary cache with data accessed by a *Load* instruction.
- F8 A Time Base transition event has occurred for the thread. This event is counted regardless of whether or not Time Base transition events are enabled by $MMCR0_{TBEE}$.
- FA The thread has loaded an instruction from a higher level cache than the tertiary cache.
- FC A thread was unable to translate an effective data address using the TLB.
- FE The thread has filled a block in the primary cache containing data accessed by a *Load* instruction with data from a location other than the secondary or tertiary cache.

56:63 ***PMC4 Selector*** (PMC4SEL)
The value of PMC4SEL specifies the event to be counted by PMC 4 as defined below.
All values in the range of C0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01-BF	Implementation dependent
C0-DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

- E0 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
- E4 A thread was unable to translate an effective data address using the TLB. (RIS)
- E6 The thread has loaded a randomly sampled instruction from a location at a higher level than the tertiary cache. (RIS)
- E8 The thread filled a block in the primary cache containing data accessed by a randomly sampled *Load* instruction from a location other than the secondary cache. (RIS, RLS)
- EA The threshold event counter has exceeded the number of events corresponding to threshold D (see Table 3). (RIS, RLS, RBS)
- EC The threshold event counter has exceeded the number of events corresponding to threshold H (see Table 3). (RIS, RLS, RBS)

- F0 The thread has attempted to load data from the primary cache but the block containing the effective address did not exist.
- F2 A cycle has occurred in which the thread has dispatched one or more instructions.
- F4 A cycle has occurred in which the PURR was incremented when the RUN bit of the CTRL register was set to 1.
- F6 The thread has incorrectly predicted the outcome of a branch instruction.
- F8 The thread has discarded fetched instructions.
- FA The thread has completed an instruction when the RUN bit of the CTRL register is set to 1.
- FC A thread was unable to translate an effective instruction address using the TLB and has stored an entry corresponding to the instructions effective address in the TLB.
- FE The thread has loaded data for a *Load* instruction from storage that did not reside in any cache.

Compatibility Note

In versions of the architecture that precede Version 2.02 the PMC Selector Fields were six bits long, and were split between MMCR0 and MMCR1. PMC1-8 were all programmable.

If more programmable PMCs are implemented in the future, additional MMCRs may be defined to cover the additional selectors.

9.4.6 Monitor Mode Control Register 2

Monitor Mode Control Register 2 (MMCR2) is a 64-bit register that contains 9-bit control fields for controlling the operation of PMC1 - PMC6 as shown below.

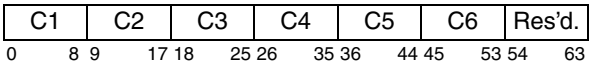


Figure 67. Monitor Mode Control Register 2

When $MMCR0_{PMCC}$ = b'11', fields C1 - C4 control the operation of PMC1 - PMC4, respectively and fields C5 and C6 are meaningless; otherwise, fields C1 - C6 control the operation of PMC1 - PMC6, respectively. The bit definitions of each Cn field are as follows, where n = 1,...6.

When the $MMCR0_{PMCC}$ is set to 10 or 11, providing problem state programs read/write access, only the FCnP bits can be accessed. All other bits are not changed when ***mtspr*** is executed in problem state, and all other bits return 0s when ***mfspr*** is executed.

Bit **Description**

0 ***Freeze Counter n in Privileged State*** (FCnS)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b00.

1 ***Freeze Counter n in Problem State if MSR_{HV}=0*** (FCnP)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b01.

Programming Note

Problem state programs need access to this field in order to enable them to individually enable counters when analyzing sections of code. All the other fields will typically be initialized by the operating system.

2 ***Freeze Counter n in Problem State if MSR_{HV}=1*** (FCnP1)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b11.

3 ***Freeze Counter n while Mark = 1*** (FCnM1)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if MSR_{PMM}=1.

4 ***Freeze Counter n while Mark = 0*** (FCnM0)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if MSR_{PMM}=0.

5 ***Freeze Counter n in Wait State*** (FCnWAIT)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $CTRL_{RUN}$=0. Software is expected to set $CTRL_{RUN}$=0 when it is in a "wait state", i.e, when there is no process ready to run.

6 ***Freeze Counter n in Hypervisor State*** (FCnH)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b10.

Bits 54:63 of MMCR2 are implementation dependent.

9.4.7 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register as defined below.

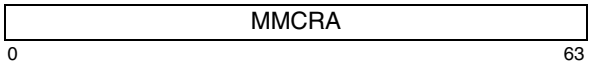


Figure 68. Monitor Mode Control Register A

MMCRA gives privileged programs the ability to control the sampling process and threshold events.

$MMCR0_{PMCC}$ controls problem-state access to this register. When $MMCR0_{PMCC}$ is set to 00 providing

read-only access, all fields can be read. When $MMCR0_{PMCC}$ is set to 10 or 11, the Threshold Event Counter Exponent (TECX) and Threshold Event Counter Mantissa (TECM) fields are read-only, and all other fields return 0s when ***mfspr*** is executed; all fields are not changed when ***mtspr*** is executed in problem state.

The bit definitions of MMCRA are as follows.

Bit(s) Description

0:31 Problem state access (SPR 770)
Reserved

Privileged access (SPR 770 or 786)
Implementation-dependent

32:33 ***BHRB Instruction Filtering Mode*** (IFM)
This field controls the filter criterion used by the hardware when recording *Branch* instructions in the BHRB. See Section 9.5.

00 No filtering
01 Do not record any *Branch* instructions unless the LK field is set to 1.
10 Do not record *I-Form* instructions. For *B-Form* and *XL-Form* instructions for which the BO field indicates "Branch always," do not record the instruction if it is *B-Form* and do not record the instruction address but record only the branch target address if it is XL-Form.
11 Filter and enter BHRB entries as for mode 10, but for *B-Form* and *XL-Form* instructions for which BO_0=1 or for which the "a" bit in the BO field is set to 1, do not record the instruction if it is *B-Form* and do not record the instruction address but record only the branch target address if it is XL-Form.

Programming Note

The filters provided by the 10 and 11 values of IFM field can be restated in terms of the operation performed as follows:

10 Do not record the instruction address of any unconditional *Branch* instruction; record only the target address of XL-form unconditional *Branch* instructions.

11 Filter as for encoding 10, but for conditional *Branch* instructions that provide a hint or that do not depend on the value of CR_{BI}, do not record the instruction if it is *B-Form* and record only the target address if it is *XL-Form*.

34:36 ***Threshold Event Counter Exponent*** (TECX)
This field species the exponent of the threshold event counter value. See Section 9.4.3 for additional information. The maximum exponent supported must be at least 5.

37 Reserved

38:44 ***Threshold Event Counter Mantissa***(TECM)

This field species the mantissa of the threshold event counter value. See Section 9.4.3 for additional information.

Programming Note

Problem state programs need to read the TECX and TECM fields during processing of an event-based branch that occurs due to a "threshold counter exceeded n" event. All the other fields will typically be initialized by the operating system.

45:47 ***Threshold Event Counter Event***(TECE)

This field specifies that is counted by the Threshold Event Counter. The values and meanings are specified as follows.

Value Event

000 Disable Counting
001 A cycle has occurred
010 An instruction has completed.
011 Reserved

All other values are implementation-dependent.

48:51 ***Threshold Start Event*** (TS)

This field specifies the event that causes the threshold event counter to start counting cycles. The events only occur if $MMCRA_{SE}$=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is en effect. (The sampling mode is specified in $MMCRA_{SM}$.)
0000Reserved.
0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)
0010 The thread has dispatched a randomly sampled instruction. (RIS)
0011 A randomly sampled instruction has been sent to a facility (e.g. *Branch, Fixed Point*, etc.) (RIS, RLS, RBS)
0100 The thread has completed a randomly sampled instruction up to the point at which it has caused all the exceptions that it will cause. (RIS, RLS, RBS)
0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
0110 The thread has failed to locate data for a randomly sampled *Load* instruction in the primary data cache. (RIS, RLS)

0111 The thread has filled a block in the primary cache containing data accessed by a randomly sampled *Load* instruction. (RIS, RLS)

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
1000 - 1111 - Reserved

Privileged access (SPR 770 or 786)
1000 - 1111 - Implementation-dependent

52:55 ***Threshold End Event*** (TE)

This field specifies the event that causes the threshold event counter to stop counting cycles. The events only occur if $MMCRA_{SE}$=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is en effect. (The sampling mode is specified in $MMCRA_{SM}$.)

0000 Reserved
0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)
0010 The thread has dispatched a randomly sampled instruction. (RIS)
0011 A randomly sampled instruction has been sent to a facility (e.g. *Branch, Fixed Point*, etc.) (RIS)
0100 The thread has completed a randomly sampled instruction up to the point at which it has caused all the exceptions that it will cause. (RIS, RLS, RBS)
0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
0110 The thread has failed to locate data for a randomly sampled *Load* instruction in the primary data cache (RIS, RLS)
0111 The thread has filled a block in the primary cache containing data accessed by a randomly sampled *Load* instruction. (RIS, RLS)

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
1000 - 1111 - Reserved

Privileged access (SPR 770 or 786)
1000 - 1111 - Implementation-dependent

56 Reserved

57:59 ***Eligibility for Random Sampling*** (ES)

When random sampling is enabled (SE=1) and the SM field indicates random instruction sampling (RIS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 All instructions
001 All Load and Store instructions
010 All probe no-op instructions
011 Reserved.

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

When random sampling is enabled (SE=1) and the SM field indicates random load/store facility sampling (RLS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 Instructions for which the thread has attempted to load data from the cache but the block containing the effective address did not exist.
001 Reserved
010 Reserved
011 Reserved

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

When random sampling is enabled (ES=1) and the SM field indicates random Branch Facility sampling (RBS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 Instructions for which the thread has incorrectly predicted the outcome of a *Branch* instruction.
001 Instructions for which the thread has incorrectly predicted the outcome of a Branch instruction because the Condition Register was different from the predicted value.
010 Instructions for which the thread has incorrectly predicted the target address of a *Branch* instruction.
011 All *Branch* instructions for which the branch was taken.

The following values have different definitions for problem state than other privilege states.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

60 Reserved

61:62 ***Random Sampling Mode*** (SM)

00 **Random Instruction Sampling** (RIS) - Instructions that meet the criterion specified in the ES field for random instruction sampling are eligible to be sampled.
01 **Random Load/Store Facility Sampling** (RLS) - Instructions that meet the criterion specified in the ES field for random Load/Store Facility sampling are eligible for sampling.
10 **Random Branch Facility Sampling** (RBS) Instructions that meet the criterion specified in the ES field for random Branch Facility sampling are eligible for sampling.
11 Reserved

63 ***Random Sample Enable*** (SE)

0 Continuous sampling is enabled.
1 Random sampling is enabled.

See Section 9.4.8 for information about sampling.

9.4.8 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register.

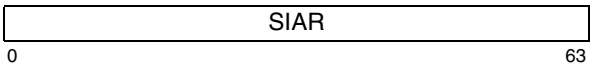


Figure 69. Sampled Instruction Address Register

When a Performance Monitor alert occurs because of an event that occurred due to a randomly sampled instruction, the SIAR contains the effective address of the instruction if $SIER_{SIARV}$ = 1 and is invalid if $SIER_{SIARV}$ = 0.

When a Performance Monitor alert occurs because of an event other than an event that occurred due to a randomly sampled instruction, the SIAR contains the effective address of an instruction that was being executed, possibly out of order, at or around the time that the Performance Monitor alert occurred.

The contents of SIAR may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIAR are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SIAR are undefined until the next Performance Monitor alert occurs.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, the value of $MSR_{HV\ PR}$ that was in effect when the sampled instruction was being executed is reported in SIER.

9.4.9 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register.

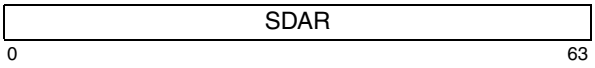


Figure 70. Sampled Data Address Register

When a Performance Monitor alert occurs because of an event that occurred due to a randomly sampled instruction, the SDAR contains the effective address of the data accessed by the instruction if $SIER_{SDARV}$ = 1 and is invalid if $SIER_{SDARV}$ = 0.

When a Performance Monitor alert occurs because of an event other than an event that occurred due to a randomly sampled instruction, the SDAR contains the effective address of the storage operand of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred. This storage operand may or may not be the storage operand (if any) associated with the instruction whose address is in the SIAR.

The contents of SDAR may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SDAR are undefined until the next Performance Monitor alert occurs.

Programming Note

If the Performance Monitor alert causes a Performance Monitor interrupt, MMCRA indicates whether the sampled data is the storage operand of the sampled instruction.

9.4.10 Sampled Instruction Event Register

The Sampled Instruction Event Register (SIER) is a 64-bit register.

When random sampling is enabled and a Performance Monitor Alert occurs because of an event caused by execution of a randomly sampled instruction, this register contains information about the sampled instruction

(i.e. the instruction whose effective address is contained in the SIAR) when a Performance Monitor alert occurred. All fields are valid unless otherwise indicated.

Programming Note

A Performance Monitor Alert is caused by execution of a randomly sampled instruction if any of the following apply when the Performance Monitor Alert occurs:

- random sampling was enabled and a counter negative condition exists in a PMC that was counting events based on randomly sampled instructions
- a threshold start event was specified in MMCRA and the threshold event counter has reached its maximum value.

When random sampling is disabled or when a Performance Monitor Alert occurs because of an event that was not caused by execution of a randomly sampled instruction, then the contents of this register are meaningless.

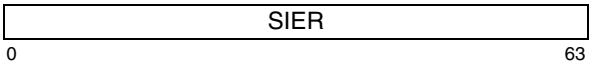


Figure 71. Sampled Instruction Event Register

The contents of SIER may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIER are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SIER are undefined until the next Performance Monitor alert occurs.

The bit definitions of the SIER are as follows.

0:37 These bits have different definitions for problem state than for other privilege states.

Problem state access (SPR 770)
Reserved

Privileged access (SPR 770 or 786)
Implementation dependent

38:40 These bits have different definitions for problem state than for other privilege states.

Problem state access (SPR 770)
Reserved

Privileged access (SPR 770 or 786)

38 ***Sampled MSR_{PR}*** (SAMPPR)
Value of MSR_{PR} when the Performance Monitor Alert occurred.

39 ***Sampled MSR_{HV}*** (SAMPHV)
Value of MSR_{HV} when the Performance Monitor Alert occurred.

40 Reserved

41 ***SIAR Valid*** (SIARV)

Set to 1 when the contents of the SIAR are valid (i.e. they contain the effective address of the sampled instruction); otherwise set to 0.

42 ***SDAR Valid*** (SDARV)

Set to 1 when the contents of the SDAR are valid (i.e. they contain the effective address of the data accessed by the sampled instruction); otherwise set to 0.

Programming Note

Other SIER fields may be valid even though both SIARV and SDARV are set to 0.

43 **T*hreshold Exceeded*** (TE)

Set to 1 by the hardware if the contents of threshold event counter exceeded the maximum value when the Performance Monitor Alert occurred; otherwise set to 0 by hardware.

44 **Slew Down**

Set to 1 by the hardware if the processor clock was lower than nominal when the performance monitor alert occurred; otherwise set to 0 by hardware.

45 **Slew Up**

Set to 1 by the hardware if the processor clock was higher than nominal when the performance monitor alert occurred; otherwise set to 0 by hardware.

46:48 ***Sampled Instruction Type*** (SITYPE)
This field indicates the sampled instruction type. The values and their meanings are as follows.

000 The hardware is unable to indicate the sampled instruction type
001 *Load* Instruction
010 *Store* instruction
011 *Branch* Instruction
100 *Floating Point* Instruction other than a *Load* or *Store* instruction
101 *Fixed Point* Instruction other than a *Load* or *Store* instruction
110 *Condition Register* or *System Call* Instruction
111 Reserved

49:51 ***Sampled Instruction Cache Information*** (SICACHE)

This field provides cache-related information about the instruction.

000 The hardware is unable to indicate any cache information for the sampled instuction.

001 The thread located the instruction in the primary cache.
010 The thread located the instruction in the secondary cache.
011 The thread located the instruction in the tertiary cache
100 The thread failed to locate the instruction in the primary, secondary, or tertiary cache
101 Reserved.
110 Reserved
111 Reserved

52 ***Sampled Instruction Taken Branch*** (SITAKBR)

Set to 1 if the SITYPE field indicates a *Branch* instruction and the branch was taken; otherwise set to 0.

53 ***Sampled Instruction Mispredicted Branch*** (SIMISPED)

Set to 1 if the SITYPE field indicates a *Branch* instruction and the thread incorrectly predicted the outcome of the instruction; otherwise set to 0.

54:55 ***Sampled Branch Instruction Misprediction Information*** (SIMISPREDICT)

If SIMISPRED=1, this field indicates how the thread incorrectly predicted the outcome of a *Branch* instruction; otherwise this field is set to 0s.
00 The instruction was not a mispredicted *Branch* instruction.
01 The thread incorrectly predicted outcome of the instruction because the value of the Condition Register was different from the predicted value.
10 The thread mispredicted the target address of the instruction.
011 Reserved

56 ***Sampled Instruction Data ERAT Miss (SIDERAT)***

When the SITYPE field indicates a *Load* or *Store* instruction, this field is set to 1 if the thread has failed to locate an ERAT entry during data address translation for the sampled instruction and otherwise is set to 0.

When the SITYPE field does not indicate a *Load* or *Store* instruction, this field is not valid.

57:59 ***Sampled Instruction Data Address Translation Information*** (SIDAXLATE)

This field contains information about data address translation for the sampled instruction. The values and their meanings are as follows.
000 The instruction did not access data.
001 The thread was able to translate the effective data address using the TLB.
010 A PTEG entry required for data translation for the instruction was loaded from the secondary cache.
011 A PTEG entry required for data translation for the instruction was loaded from the tertiary cache.
100 A PTEG entry required for data translation for the instruction was loaded from storage that did not reside in any cache.
101 A PTEG entry required for data translation for the instruction was loaded from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110 A PTEG entry required for data translation for the instruction was loaded from a cache on a different chip from the thread.
111 Reserved

60:62 ***Sampled Instruction Storage Access Information*** (SISAI)

This field contains information about storage accesses made by the sampled instruction. The values and their meanings are as follows.
000 The instruction did not access data.
001 The thread was able to access the data referenced by the instruction from the primary cache.
010 The thread was able to access the data referenced by the instruction from the secondary cache.
011 The thread was able to access the data referenced by the instruction from the tertiary cache.
100 The thread was able to was able to access the data referenced by the instruction from storage that did not reside in any cache.
101 The thread was able to access the data referenced by the instruction from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110 The thread was able to access the data referenced by the instruction from a cache on a different chip from the thread.
111 The instruction was a Store for which the data was placed into a location other than the primary cache.

63 ***Sampled Instruction Completed*** (SICMPL)

Set to 1 if the sampled instruction has completed; otherwise set to 0.

9.5 Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is described in Book I but only at the level required by application programmers. Additional aspects of the BHRB are described here.

In order to enable problem state programs to use the BHRB, $MMCR0_{BHRBA}$ must be set to 1 to enable execution of ***mfbhrb*** and ***clrbhrb*** instructions in problem state. Additionally, $MMCR0_{EBE}$ must also be set to 1 to enable Performance Monitor event-based branches, and $MMCR0_{PMCC}$ must be set to 10 or 11 to allow problem state programs to access the necessary Performance Monitor registers. (See Section 9.4.4.)

The BHRB is written by the hardware if and only if performance monitor alerts are enabled by setting $MMCR0_{PMAE}$ to 1. After $MMCR0_{PMAE}$ has been set to 1 and a Performance Monitor alert occurs, $MMCR0_{PMAE}$ is set to 0 and the BHRB is not altered by hardware until software sets $MMCR0_{PMAE}$ to 1 again.

When $MMCR0_{PMAE}$=1, ***mfbhrbe*** instructions return 0s to the target register.

BHRB Entries

When the BHRB is written by hardware, only those *Branch* instructions that meet the filtering criterion indicated by $MMCRA_{PFM\ IFM}$ and for which the branch was taken are included.

In addition to the mode of BHRB operation described in Book I, an implementation is allowed to provide an implementation-dependent mode of the operation in which the entries that are written are implementation dependent. If such a mode is provided, the means by which the BHRB is put into this mode must be a hypervisor resource.

Programming Note

A potential combined use of the Trace and Performance Monitor facilities is to trace the control flow of a program and simultaneously count events for that program.

9.6 Interaction With Other Facilities

9.6.1 Trace Facility

If the Trace facility includes setting SIAR and SDAR (see Section 6.5.15, “Trace Interrupt [Category: Trace]”), and tracing is active (MSR_{SE}=1 or MSR_{BE}=1), the contents of SIAR and SDAR as used by the Performance Monitor facility are undefined and may change even when $MMCR0_{PMAE}$=0.

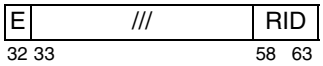
Chapter 10. External Control [Category: External Control]

The External Control facility permits a program to communicate with a special-purpose device. The facility consists of a Special Purpose Register, called EAR, and two instructions, called *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***).

This facility must provide a means of synchronizing the devices with the hardware to prevent the use of an address by the device when the translation that produced that address is being invalidated.

10.1 External Access Register

This 32-bit Special Purpose Register controls access to the External Control facility and, for external control operations that are permitted, identifies the target device.



Bit(s)	Name	Description
32	E	Enable bit
58:63	RID	Resource ID

All other fields are reserved.

Figure 72. External Access Register

The External Access Register (EAR) is a hypervisor resource; see Chapter 2.

The high-order bits of the RID field that correspond to bits of the Resource ID beyond the width of the Resource ID supported by the implementation are treated as reserved bits.

Programming Note

The hypervisor can use the EAR to control which programs are allowed to execute *External Access* instructions, when they are allowed to do so, and which devices they are allowed to communicate with using these instructions.

10.2 External Access Instructions

The *External Access* instructions, *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***), are described in Book II. Additional information about them is given below.

If attempt is made to execute either of these instructions when EAR_E=0, a Data Storage interrupt occurs with bit 43 of the DSISR set to 1.

The instructions are supported whenever MSR_{DR}=1. If either instruction is executed when MSR_{DR}=0 (real addressing mode), the results are boundedly undefined.

Chapter 11. Processor Control [Category: Server.Processor Control]

11.1 Overview

The Processor Control facility provides a mechanism for the hypervisor to send messages to other threads that are on the same multi-threaded processor. Privileged programs are able to send messages to other threads on the same multi-threaded processor; if the processor is configured into sub-processors, privileged programs can only send messages to other threads on the same sub-processor.

11.2 Programming Model

Both hypervisor-level and privileged-level messages can be sent. Hypervisor-level messages are sent using the ***msgsnd*** instruction and cause hypervisor-level exceptions when accepted. Privileged-level messages are sent using the ***msgsndp*** instruction and cause privileged-level exceptions when accepted. For both instructions, the message type is specified in a general purpose register.

11.2.1 Message Type

The message type is specified by the contents of bits 32:36 in the RB operand of the ***msgsnd*** or ***msgsndp*** instruction as follows.

Message type for *msgsnd*

$RB_{32:36}$	Description
5	***Directed Hypervisor Doorbell Interrupt*** (DH_DBELL) A Directed Hypervisor Doorbell exception is generated on a thread only after it has filtered and determined that it should accept the message, and the thread is on the same multi-threaded processor as the thread executing the ***msgsnd*** instruction.

All other values of $RB_{32:36}$ are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.

Message type for *msgsndp*

$RB_{32:36}$	Description
5	***Directed Privileged Doorbell Interrupt*** (DP_DBELL) A Directed Privileged Doorbell exception is generated on a thread only after it has filtered and determined that it should accept the message, and the following are satisfied: – for processors not partitioned into sub-processors, the thread is on the same multi-threaded processor as the thread executing the ***msgsndp*** instruction, – for processors partitioned into sub-processors, the thread is on the same sub-processor as the thread executing the ***msgsndp*** instruction.

All other values of $RB_{32:36}$ are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.

11.2.2 Doorbell Message Payload and Filtering

The message payload is specified by the contents of bits 37:63 in the RB operand of the ***msgsnd*** or ***msgsndp*** instruction.

Bit	Description
37	***Reserved***
38:56	***Reserved***
57:63	***TIR Tag*** (TIRTAG) For ***msgsndp*** instructions, the recipient of the message compares this field during the filtering process with its privileged thread number. For ***msgsnd*** instructions, the recipient of the message compares this field during the filtering process with its hypervisor thread number.

Programming Note

If ***msgsndp*** is executed with TIRTAG set to a value greater than the highest privileged thread number on the sub-processor (or on the processor if sub-processors are not supported), then this instruction behaves as a no-op because the filtering process (see below) prevents the receiving threads from accepting it. Similarly, if ***msgsnd*** is executed with TIRTAG set to a value greater than the highest hypervisor thread number on the processor, then the instruction also behaves as a no-op.

Filtering

The examination of the message payload for the purpose of determining if the message is to be accepted is referred to as *filtering*.

If a Directed Hypervisor Doorbell message is received by a thread, the message is accepted and the corresponding exception is generated only if the TIRTAG field of the message payload is equal to the hypervisor thread number of the recipient.

If a Directed Privileged Doorbell message is received by a thread, the message is accepted and the corresponding exception is generated only if the TIRTAG field of the message payload is equal to the privileged thread number of the recipient.

If the message is to be accepted, the exception specified by the type field is generated, otherwise the message is ignored. When the exception is generated, the corresponding interrupt occurs when no higher priority exception exists and the interrupt is enabled (MSR_{EE}=1 for the Directed Privileged Doorbell interrupt and MSR_{EE}=1 or MSR_{HV}=0 for the Directed Hypervisor Doorbell interrupt).

A Directed Privileged Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a ***mtspr*** or ***msgclrp*** instruction.

A Directed Hypervisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a ***mtspr*** or ***msgclr*** instruction.

If a doorbell exception is present and the corresponding interrupt is pended because MSR_{EE}=0, additional doorbell exceptions are ignored until the exception is cleared.

11.3 Processor Control Registers

11.3.1 Directed Privileged Doorbell Exception State

The layout of the Directed Privileged Doorbell Exception State (DPDES) register is shown in below.

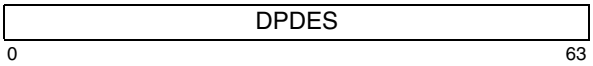


Figure 73. Directed Privileged Doorbell Exception State Register

The DPDES register is a 64-bit register. For t < T, where T is the number of threads on the sub-processor (or on the multi-threaded processor if sub-processors are not supported), bit 63-t corresponds to the thread with privileged thread number t.

When the contents of $DPDES_{63-t}$ change from 0 to 1, a Directed Privileged Doorbell exception will come into existence within a reasonable period of time. When the contents of $DPDES_{63}$-t change from 1 to 0, the existing Directed Privileged Doorbell exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of $DPDES_{63-t}$ is the result a ***msgsndp*** or ***msgclrp*** instruction or of modification of the DPDES register caused by execution of an ***mtspr*** instruction.

Bits 0:63-T of the DPDES are reserved.

11.3.2 Directed Hypervisor Doorbell Exception State

The layout of the Directed Hypervisor Doorbell Exception State (DPDES) register is shown in below.

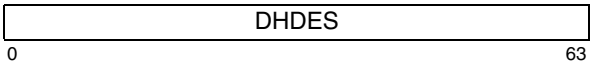


Figure 74. Directed Hypervisor Doorbell Exception State Register

The DHDES register is a 64-bit register. For t < T, where T is the number of threads on the multi-threaded processor, bit 63-t corresponds to the thread with hypervisor thread number t.

When the contents of $DHDES_{63-t}$ change from 0 to 1, a Directed Hypervisor Doorbell exception will come into existence within a reasonable period of time. When the contents of $DHDES_{63}$-t change from 1 to 0, the existing Directed Hypervisor Doorbell exception, if any, will

cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of $DHDES_{63-t}$ is the result a ***msgsnd*** or ***msgclr*** instruction or of modification of the DHDES register caused by execution of an ***mtspr*** instruction.

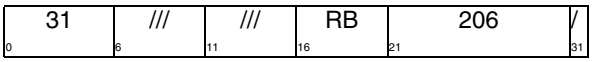
Bits 0:63-T of the DHDES are reserved.

11.4 Processor Control Instructions

msgsnd, msgsndp, msgclr, and ***msgclrp*** instructions are provided for sending and clearing messages. ***msgsndp*** and ***msgclrp*** are privileged instructions, ***msgsnd*** and ***msgclr*** are hypervisor privileged instructions.

Message Send *X-form*

msgsnd RB




```
msgtype ← GPR(RB)32:36
payload ← GPR(RB)37:63
t ← RB57:63
IF(msgtype = 0x05) and
   t ≤ maximum hypervisor thread number
       on processor
  then
   DHDES63-t ← 1
   send_msg(msgtype, payload)
```

msgsnd sends a message to other threads.

Let msgtype be $(RB)_{32:36}$, let message payload be $(RB_{37:63})$, and let t be the hypervisor thread number indicated in $RB_{57:63}$. If msgtype = 0x05 and t is less than or equal to the maximum hypervisor thread number, then send the Directed Hypervisor Doorbell message to thread t on the same multithreaded processor (or sub-processor if sub-processors are supported), and set $DPDES_{63-t}$ to 1.

The actions taken on receipt of a message are defined in Section 11.2.2.

This instruction is hypervisor privileged.

Special Registers Altered:

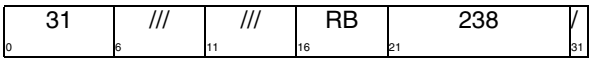
DHDES

Programming Note

If ***msgsnd*** is used to notify the receiver that updates have been made to storage, a ***sync*** should be placed between the stores and the ***msgsnd***. See Section 5.9.2.

Message Clear *X-form*

msgclr RB




```
msgtype ← (RB)32:36
IF(msgtype = 0x05)
  then
   t ← hypervisor thread number of executing thread
   DHDES63-t ← 0
```

msgclr clears a message of *msgtype* previously accepted by the thread executing the ***msgclr***.

Let msgtype be $(RB)_{32:36}$, and let t be the hypervisor thread number of the thread executing the ***msgclr***. If msgtype = 0x05, then clear any exception that exists for this message type by setting $DHDES_{63-t}$ to 0; otherwise do not modify DHDES or clear any exceptions for this message.

Execution of ***msgclr*** ensures that the formerly pending exception will not result in an interrupt when the corresponding interrupt class is reenabled.

The types of messages that can be cleared are defined in Section 11.2.1.

This instruction is hypervisor privileged.

Special Registers Altered:

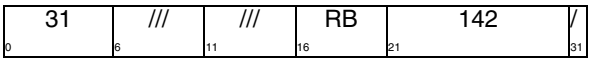
DHDES

Programming Note

msgclr is typically issued only when MSR_{EE}=0. If ***msgclr*** is executed when MSR_{EE}=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

Message Send Privileged *X-form*

msgsndp RB




```
msgtype ← GPR(RB)32:36
payload ← GPR(RB)37:63
t ← RB57:63
IF(msgtype = 0x05) and
   t ≤ maximum privileged thread number
       on processor or sub-processor
  then
   DPDES63-t ← 1
   send_msg(msgtype, payload)
```

msgsndp sends a message to other threads.

Let msgtype be $(RB)_{32:36}$, let message payload be $(RB_{37:63})$, and let t be the privileged thread number indicated in $RB_{57:63}$. If msgtype = 0x05 and t is less than or equal to the maximum privileged thread number on the multi-threaded processor (or on the sub-processor if sub-processors are supported), then send the Directed Privileged Doorbell message to thread t on the same multithreaded processor (or sub-processor if sub-processors are supported), and set $DPDES_{63-t}$ to 1.

The actions taken on receipt of a message are defined in Section 11.2.2.

This instruction is privileged.

Special Registers Altered:

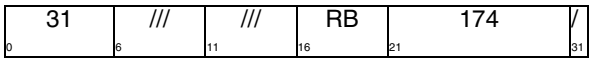
DPDES

Programming Note

If ***msgsndp*** is used to notify the receiver that updates have been made to storage, a ***sync*** should be placed between the stores and the ***msgsnd***. See Section 5.9.2.

Message Clear Privileged *X-form*

msgclrp RB




```
msgtype ← (RB)32:36
IF(msgtype = 0x05)
  then
   t ← privileged thread number of executing thread
   DPDES63-t ← 0
```

msgclrp clears a message of *msgtype* previously accepted by the thread executing the ***msgclrp***.

Let msgtype be $(RB)_{32:36}$, and let t be the thread number of the thread executing the ***msgclrp***. If msgtype = 0x05, then clear any exception that exists for this message type by setting $DPDES_{63-t}$ to 0; otherwise do not modify DPDES or clear any exceptions for this message.

Execution of ***msgclrp*** ensures that the formerly pending exception will not result in an interrupt when the corresponding interrupt class is reenabled.

The types of messages that can be cleared are defined in Section 11.2.1.

This instruction is privileged.

Special Registers Altered:

DPDES

Programming Note

msgclrp is typically issued only when MSR_{EE}=0. If ***msgclrp*** is executed when MSR_{EE}=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

Chapter 12. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers, the contents of SLB entries, or the contents of other system resources that control the context in which a program executes can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed and data accesses are performed. For example, changing MSR_{IR} from 0 to 1 has the side effect of enabling translation of instruction addresses. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed or data accesses are performed, is called a *context-altering instruction*. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 4 (for data access) and Table 5 (for instruction fetch and execution).

The notation "CSI" in the tables means any context synchronizing instruction (e.g., ***sc***, ***isync***, or ***rfid***). A context synchronizing interrupt (i.e., any interrupt except non-recoverable System Reset or non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like "the synchronizing instruction", below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, "the synchronizing instruction before (after) the context-altering instruction" should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Programming Note

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as the ***rfid*** that returns from an interrupt handler, provide the required synchronization.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing or when altering the MSR in most cases (see the tables). No software synchronization is required before most of the other alterations shown in Table 5, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the hardware must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a single-threaded environment.

Instruction or Event	Required Before	Required After	Notes
event-based branch and ***rfebb***	none	none	21
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
sc	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	
mtmsrd (TM)	none	none	
mtmsrd (TS)	none	none	
mtmsrd (SLE)	none	none	
mtmsr[***d***] (PR)	none	none	
mtmsr[***d***] (DR)	none	none	
mtsr[***in***]	CSI	CSI	
mtspr (SDR1)	***ptesync***	CSI	3,4
mtspr (AMR)	CSI	CSI	15
mtspr (EAR)	CSI	CSI	
mtspr (RMOR)	CSI	CSI	13, 19
mtspr (HRMOR)	CSI	CSI	13,19
mtspr (LPCR)	CSI	CSI	13
mtspr (DAWRn)	--	CSI	
mtspr (DAWRXn)	--	CSI	
slbie	CSI	CSI	
slbia	CSI	CSI	
slbmte	CSI	CSI	11
tlbie	CSI	CSI	5,7
tlbiel	CSI	***ptesync***	5
tlbia	CSI	CSI	5
Store(PTE)	none	{***ptesync***, CSI}	6,7
transaction failure and all TM instructions except ***tcheck***	none	none	21

Table 4: Synchronization requirements for data access

Instruction or Event	Required Before	Required After	Notes
event-based branch and ***rfebb***	none	none	21
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
sc	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	8
mtmsrd (TM)	none	none	
mtmsrd (TS)	none	none	
mtmsr[***d***] (EE)	none	none	1
mtmsr[***d***] (PR)	none	none	9
mtmsr[***d***] (FP)	none	none	
mtmsr[***d***](FE0,FE1)	none	none	
mtmsr[***d***] (SE, BE)	none	none	
mtmsr[***d***] (IR)	none	none	9
mtmsr[***d***] (RI)	none	none	
mtsr[***in***]	none	CSI	9
mtspr (CIABR)	CSI	CSI	
mtspr (DEC)	none	none	10
mtspr (SDR1)	***ptesync***	CSI	3,4,19
mtspr (CTRL)	none	none	
mtspr (HDEC)	none	none	10
mtspr (IAMR)	none	CSI	
mtspr (RMOR)	none	CSI	14,19
mtspr (HRMOR)	none	CSI	9,13,19
mtspr (LPCR)	none	CSI	13, 14
mtspr (LPIDR)	CSI	CSI	7,12,16,19
mtspr (PCR)	none	CSI	19
mtspr (TEXASR)	none	none	
mtspr (TFHAR)	none	none	
mtspr (DPDES)	none	CSI	19
mtspr (DHDES)	none	CSI	19
mtspr (BESCR)	none	CSI	18,20
mtspr (FSCR)	none	CSI	
mtspr (HFSCR)	none	CSI	
mtspr (Perf. Mon.)	none	CSI	17,20
slbie	none	CSI	
slbia	none	CSI	
slbmte	none	CSI	9,11
tlbie	none	CSI	5,7
tlbiel	none	CSI	5
tlbia	none	CSI	5
Store(PTE)	none	{***ptesync***, CSI}	6,7,9
transaction failure and all TM instructions except ***tcheck***	none	none	21

Table 5: Synchronization requirements for instruction fetch and/or execution

Notes:

1. The effect of changing the EE bit is immediate, even if the ***mtmsr***[***d***] instruction is not context synchronizing (i.e., even if L=1).
   - If an ***mtmsr***[***d***] instruction sets the EE bit to 0, neither an External interrupt a Decrementer interrupt nor a Performance Monitor interrupt occurs after the ***mtmsr***[***d***] is executed.
   - If an ***mtmsr***[***d***] instruction changes the EE bit from 0 to 1 when an External, Decrementer, Performance Monitor or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] is executed, and before the next instruction is executed in the program that set EE to 1.
   - If a hypervisor executes the ***mtmsr***[***d***] instruction that sets the EE bit to 0, a Hypervisor Decrementer interrupt does not occur after ***mtmsr***[***d***] is executed as long as the thread remains in hypervisor state.
   - If the hypervisor executes an ***mtmsr***[***d***] instruction that changes the EE bit from 0 to 1 when a Hypervisor Decrementer or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] instruction is executed, and before the next instruction is executed, provided HDICE is 1.
2. Synchronization requirements for this instruction are implementation-dependent.
3. SDR1 must not be altered when MSR_{DR}=1 or MSR_{IR}=1; if it is, the results are undefined.
4. A ***ptesync*** instruction is required before the ***mtspr*** instruction because (a) SDR1 identifies the Page Table and thereby the location of Reference and Change bits, and (b) on some implementations, use of SDR1 to update Reference and Change bits may be independent of translating the virtual address. (For example, an implementation might identify the PTE in which to update the Reference and Change bits in terms of its offset in the Page Table, instead of its real address, and then add the Page Table address from SDR1 to the offset to determine the real address at which to update the bits.) To ensure that Reference and Change bits are updated in the correct Page Table, SDR1 must not be altered until all Reference and Change bit updates associated with address translations that were performed, by the thread executing the ***mtspr*** instruction, before the ***mtspr*** instruction is executed have been performed with respect to that thread. A ***ptesync*** instruction guarantees this synchronization of Reference and Change bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.
5. For data accesses, the context synchronizing instruction before the ***tlbie***, ***tlbiel***, or ***tlbia*** instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the ***tlbie***, ***tlbiel***, or ***tlbia*** instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the TLB entry(s) being invalidated.

   (If it is necessary to order storage accesses associated with preceding instructions, or Reference and Change bit updates associated with preceding address translations, with respect to subsequent data accesses, a ***ptesync*** instruction must also be used, either before or after the ***tlbie***, ***tlbiel***, or ***tlbia*** instruction. These effects of the ***ptesync*** instruction are described in the last paragraph of Note 8.)
6. The notation "{***ptesync***,CSI}" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the *Store* instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the *Store* instruction are not performed again after the store has been performed (see Section 5.5). These properties ensure that all address translations associated with instructions preceding the *Store* instruction will be performed using the old contents of the PTE.

   The ***ptesync*** instruction after the *Store* instruction ensures that all searches of the Page Table that are performed after the ***ptesync*** instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the ***ptesync*** instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).

   The ***ptesync*** instruction also ensures that all storage accesses associated with instructions preceding the ***ptesync*** instruction, and all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ***pte-***

sync instruction are performed with respect to that thread or mechanism.

7. There are additional software synchronization requirements for this instruction in multi-threaded environments (e.g., it may be necessary to invalidate one or more TLB entries on all threads in the system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect).

   Section 5.10 gives examples of using ***tlbie***, *Store*, and related instructions to maintain the Page Table, in both multi-threaded environments and environments consisting of only a single-threaded processor.

   Programming Note

   In a multi-threaded system, if software locking is used to help ensure that the requirements described in Section 5.10 are satisfied, the ***lwsync*** instruction near the end of the lock acquisition sequence (see Section B.2.1.1 of Book II) may naturally provide the context synchronization that is required before the alteration.

8. The alteration must not cause an implicit branch in effective address space. Thus, when changing MSR_{SF} from 1 to 0, the ***mtmsrd*** instruction must have an effective address that is less than 2^{32} - 4. Furthermore, when changing MSR_{SF} from 0 to 1, the ***mtmsrd*** instruction must not be at effective address 2^{32} - 4 (see Section 5.3.2 on page 889).

9. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

   Programming Note

   If it is desired to set MSR_{IR} to 1 early in an operating system interrupt handler, advantage can sometimes be taken of the fact that $EA_{0:3}$ are ignored when forming the real address when address translation is disabled and MSR_{HV} = 0. For example, if address translation resources are set such that effective address 0xn000_0000_0000_0000 maps to real address 0x000_0000_0000_0000 when address translation is enabled, where n is an arbitrary 4-bit value, the following code sequence, in real page 0, can be used early in the interrupt handler.

```
       la rx,target
       li ry,0xn000
       sldiry,ry,48
       or rx,rx,ry  # set high-order
                       nibble of target
                           addr to 0xn
       mtctrrx
       bcctr   # branch to targ

targ:  mfmsrrx
       orirx,rx,0x0020
       mtmsrdrx# set MSR[IR] to 1
```

   The ***mtmsrd*** does not cause an implicit branch in real address space because the real address of the next sequential instruction is independent of MSR_{IR}. Using ***mtmsrd***, rather than ***rfid*** (or similar context synchronizing instruction that alters the control flow), may yield better performance on some implementations.

   (Variations on the technique are possible. For example, the target instruction of the ***bcctr*** can be in arbitrary real page P, where P is a 48-bit value, provided that effective address 0xn || P || 0x000 maps to real address P || 0x000 when address translation is enabled.)

10. The elapsed time between the contents of the Decrementer or Hypervisor Decrementer becoming negative and the signaling of the corresponding exception is not defined.

11. If an ***slbmte*** instruction alters the mapping, or associated attributes, of a currently mapped ESID, the ***slbmte*** must be preceded by an ***slbie*** (or ***slbia***) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the ***slbie*** and the ***slbmte***, regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the ***slbmte***.

    No ***slbie*** (or ***slbia***) is needed if the ***slbmte*** instruction replaces a valid SLB entry with a mapping of a

different ESID (e.g., to satisfy an SLB miss). However, the ***slbie*** is needed later if and when the translation that was contained in the replaced SLB entry is to be invalidated.

12. The context synchronizing instruction before the ***mtspr*** instruction ensures that the LPIDR is not altered out-of-order. (Out-of-order alteration of the LPIDR could permit the requirements described in Section 5.10.1 to be violated. For the same reason, such a context synchronizing instruction may be needed even if the new LPID value is equal to the old LPID value.)

    See also Chapter 2. "Logical Partitioning (LPAR) and Thread Control" on page 843 regarding moving a thread from one partition to another.

13. When the RMOR or HRMOR is modified, or the VC, VRMASD, or RMLS fields of the LPCR are modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of these registers or fields (i.e., the contents immediately before the modification). The ***slbia*** instruction can be used to invalidate all such implementation-specific lookaside information.

14. A context synchronizing instruction or event that is executed or occurs when $LPCR_{MER}$ = 1 does not necessarily ensure that the exception effects of $LPCR_{MER}$ are consistent with the contents of $LPCR_{MER}$. See Section 2.2.

15. This line applies regardless of which SPR number (13 or 29) is used for the AMR.

16. LPIDR must not be altered when MSR_{DR}=1 or MSR_{IR}=1; if it is, the results are undefined.

17. This line applies to the following Performance Monitor SPRs: PMC1-6, MMCR0, MMCR1, MMCR2, and MMCRA.

18. This line applies to all SPR numbers that access the BESCR (800-803, 806).

19. There are additional software synchronization requirements when an ***mtspr*** instruction modifies this SPR in a multi-threaded environment. See Section 2.8.

20. As an alternative to a CSI, the execution of an ***rfebb*** instruction or the occurrence of an event-based branch is sufficient to provide the necessary synchronization.

21. These instructions and events, with the exception of nested ***tbegin.*** and ***tend.***, TM instructions that except or are described to be treated as noops, *Conditional Abort* instructions that do not abort, and events and ***rfebb*** instructions for which the event did not take place in Transactional state, will change MSR_{TS}. No explicit synchronization is required.

Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book III.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Move To/From Special Purpose Register Mnemonics

This section defines extended mnemonics for the ***mtspr*** and ***mfspr*** instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, and for the *Move From Time Base* instruction defined in Book II.

The ***mtspr*** and ***mfspr*** instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the *Move From Time Base* instruction, which specifies the portion of the Time Base as a numeric operand.

Note: ***mftb*** serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

Programming Note

The extended mnemonics in Table 6 for SPRs associated with the Performance Monitor facility are based on the definitions in Chapter 9..

Other versions of Performance Monitor facilities used different sets of SPR numbers (all 32-bit PowerPC implementations used a different set, and some early Power ISA implementations used yet a different set).

Table 6: Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR[1]	
	Extended	Equivalent to	Extended	Equivalent to
Fixed-Point Exception Register	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Data Stream Control Register	mtdscr Rx	mtspr 17,Rx	mfdscr Rx	mfspr Rx,17
Data Storage Interrupt Status Register	mtdsisr Rx	mtspr 18,Rx	mfdsisr Rx	mfspr Rx,18
Data Address Register	mtdar Rx	mtspr 19,Rx	mfdar Rx	mfspr Rx,19
Decrementer	mtdec Rx	mtspr 22,Rx	mfdec Rx	mfspr Rx,22
Storage Description Register 1	mtsdr1 Rx	mtspr 25,Rx	mfsdr1 Rx	mfspr Rx,25
Save/Restore Register 0	mtsrr0 Rx	mtspr 26,Rx	mfsrr0 Rx	mfspr Rx,26
Save/Restore Register 1	mtsrr1 Rx	mtspr 27,Rx	mfsrr1 Rx	mfspr Rx,27
Come-From Address Register	mtcfar Rx	mtspr 28,Rx	mfcfar Rx	mfspr Rx,28
AMR	mtamr Rx	mtspr 29,Rx	mfamr Rx	mfspr Rx,29
IC	mtic Rx	mtspr 848, Rx	mfic Rx	mfspr Rx, 848
CTRL	mtctrl Rx	mtspr 152,Rx	mfctrl Rx	mfspr Rx,136
UAMOR	mtuamor Rx	mtspr 157,Rx	mfuamor Rx	mfspr Rx,157
Special Purpose Registers G0 through G3	mtsprg n,Rx	mtspr 272+n,Rx	mfsprg Rx,n	mfspr Rx,272+n
Time Base [Lower]	mttbl Rx	mtspr 284,Rx	mftb Rx	mftb Rx,268[1] mfspr Rx,268
Time Base Upper	mttbu Rx	mtspr 285,Rx	mftbu Rx	mftb Rx,269[1] mfspr Rx,269
Time Base Upper 40	mttbu40 Rx	mtspr 286,Rx	-	-
VTB	mtvtb Rx	mtspr 849, Rx	mfvtb Rx	mfspr Rx, 849
Processor Version Register	-	-	mfpvr Rx	mfspr Rx,287
HMER	mthmer Rx	mtspr 336,Rx	mfhmer Rx	mfspr Rx,336
HMEER	mthmeer Rx	mtspr 337,Rx	mfhmeer Rx	mfspr Rx,337
AMOR	mtamor Rx	mtspr 349,Rx	mfamor Rx	mfspr Rx,349
MMCRA	mtmmcra Rx	mtspr 786,Rx	mfmmcra Rx	mfspr Rx,770
PMC1	mtpmc1 Rx	mtspr 787,Rx	mfpmc1 Rx	mfspr Rx,771
PMC2	mtpmc2 Rx	mtspr 788,Rx	mfpmc2 Rx	mfspr Rx,772
PMC3	mtpmc3 Rx	mtspr 789,Rx	mfpmc3 Rx	mfspr Rx,773
PMC4	mtpmc4 Rx	mtspr 790,Rx	mfpmc4 Rx	mfspr Rx,774
PMC5	mtpmc5 Rx	mtspr 791,Rx	mfpmc5 Rx	mfspr Rx,775
PMC6	mtpmc6 Rx	mtspr 792,Rx	mfpmc6 Rx	mfspr Rx,776
MMCR0	mtmmcr0 Rx	mtspr 795,Rx	mfmmcr0 Rx	mfspr Rx,779
MMCR1	mtmmcr1 Rx	mtspr 798,Rx	mfmmcr1 Rx	mfspr Rx,782
PPR	mtppr Rx	mtspr 896, Rx	mfppr Rx	mfspr Rx, 896
PPR32	mtppr32 Rx	mtspr 898, Rx	mfppr32 Rx	mfspr Rx, 898
Processor Identification Register	-	-	mfpir Rx	mfspr Rx,1023

[1] The ***mftb*** instruction is Category: Phased-Out. Assemblers targeting Version 2.03 or later of the architecture should generate an ***mfspr*** instruction for the mftb and mftbu extended mnemonics; see the corresponding Assembler Note in the ***mftb*** instruction description (see Section 6.2.1 of Book II).

Appendix B. Interpretation of the DSISR as Set by an Alignment Interrupt

For most causes of Alignment interrupt, the interrupt handler will emulate the interrupting instruction. To do this, it needs the following characteristics of the interrupting instruction:

Load or store
Length (halfword, word, doubleword)
String, multiple, or elementary
Fixed-point or floating-point
Update or non-update
Byte reverse or not
Is it ***dcbz***?

The Power ISA optionally provides this information by setting bits in the DSISR that identify the interrupting instruction type. It is not necessary for the interrupt handler to load the interrupting instruction from storage. The mapping is unique except for a few exceptions that are discussed below. The near-uniqueness depends on the fact that many instructions, such as the fixed- and floating-point arithmetic instructions and the one-byte loads and stores, cannot cause an Alignment interrupt.

See Section 6.5.8 for a description of how the opcode and extended opcode are mapped to a DSISR value for an X-, D-, or DS-form instruction that causes an Alignment interrupt.

The table on the next page shows the inverse mapping: how the DSISR bits identify the interrupting instruction. The following notes are cited in the table.

1. For cases in which multiple instructions can give the same value for the DSISR bits (44:45 47:53) that are derived from the opcode, the Alignment interrupt handler should load the instruction from storage, using the effective address in SRR0, and treat the instruction appropriately (e.g., emulate the instruction, treat the case as a programming error, etc.). For example, if ***lbarx*** or ***lwarx*** causes an Alignment interrupt, it should not be emulated, but instead should be treated as a programming error.
2. These are distinguished by DSISR bits 44:45, which are not shown in the table.

The interrupt handler has no need to distinguish between an X-form instruction and the corresponding D- or DS-form instruction if one exists, and vice versa. Therefore two such instructions may yield the same DSISR value (all 32 bits). For example, ***stw*** and ***stwx*** may both yield either the DSISR value shown in the following table for ***stw***, or that shown for ***stwx***.

If DSISR 47:53 is:	then it is either X-form opcode:	or D/DS/DQ/DQE-form opcode:	so the instruction is:
00 0 0000	00000xxx00	x00000	lwarx,lwz, reserved(1)
00 0 0001	00010xxx00	x00010	ldarx
00 0 0010	00100xxx00	x00100	stw
00 0 0011	00110xxx00	x00110	-
00 0 0100	01000xxx00	x01000	lhz, lqarx
00 0 0101	01010xxx00	x01010	lha, lxvdsx(1)
00 0 0110	01100xxx00	x01100	sth
00 0 0111	01110xxx00	x01110	lmw
00 0 1000	10000xxx00	x10000	lfs
00 0 1001	10010xxx00	x10010	lfd, lxsdx(1)
00 0 1010	10100xxx00	x10100	stbrx, stfs(1)
00 0 1011	10110xxx00	x10110	stfd, stxsdx(1)
00 0 1100	11000xxx00	x11000	lq
00 0 1101	11010xxx00	x11010	ld, ldu, lwa (2)
00 0 1110	11100xxx00	x11100	stxvw4x
00 0 1111	11110xxx00	x11110	std, stdu, stq (2)
00 1 0000	00001xxx00	x00001	lbarx, lwzu (1)
00 1 0001	00011xxx00	x00011	lharx
00 1 0010	00101xxx00	x00101	stwu
00 1 0011	00111xxx00	x00111	-
00 1 0100	01001xxx00	x01001	lhzu
00 1 0101	01011xxx00	x01011	lhau
00 1 0110	01101xxx00	x01101	sthu
00 1 0111	01111xxx00	x01111	stmw
00 1 1000	10001xxx00	x10001	lfsu
00 1 1001	10011xxx00	x10011	lfdu
00 1 1010	10101xxx00	x10101	stfsu
00 1 1011	10111xxx00	x10111	stfdu
00 1 1100	11001xxx00	x11001	lfdp
00 1 1110	11101xxx00	x11101	stfdp
01 0 0000	00000xxx01		ldx
01 0 0001	00010xxx01		-
01 0 0010	00100xxx01		stdx
01 0 0011	00110xxx01		-
01 0 0100	01000xxx01		-
01 0 0101	01010xxx01		lwax
01 0 0110	01100xxx01		-
01 0 0111	01110xxx01		-
01 0 1000	10000xxx01		lswx
01 0 1001	10010xxx01		lswi
01 0 1010	10100xxx01		stswx
01 0 1011	10110xxx01		stswi
01 0 1100	11000xxx01		-
01 0 1101	11010xxx01		-
01 0 1110	11100xxx01		-
01 0 1111	11110xxx01		-
01 1 0000	00001xxx01		ldux
01 1 0001	00011xxx01		-
01 1 0010	00101xxx01		stdux
01 1 0011	00111xxx01		-
01 1 0100	01001xxx01		-
01 1 0101	01011xxx01		lwaux
01 1 0110	01101xxx01		-
01 1 0111	01111xxx01		-
01 1 1000	10001xxx01		-
01 1 1001	10011xxx01		-
01 1 1010	10101xxx01		-
01 1 1011	10111xxx01		-
01 1 1100	11001xxx01		-
01 1 1101	11011xxx01		-
01 1 1110	11101xxx01		-
01 1 1111	11111xxx01		-

If DSISR 47:53 is:	then it is either X-form opcode:	or D/DS/DQ/DQE-form opcode:	so the instruction is:
10 0 0000	00000xxx10		-
10 0 0001	00010xxx10		-
10 0 0010	00100xxx10		stwcx.
10 0 0011	00110xxx10		stdcx.
10 0 0100	01000xxx10		-
10 0 0101	01010xxx10		-
10 0 0110	01100xxx10		-
10 0 0111	01110xxx10		-
10 0 1000	10000xxx10		lwbrx
10 0 1001	10010xxx10		-
10 0 1010	10100xxx10		stwbrx
10 0 1011	10110xxx10		sthcx.
10 0 1100	11000xxx10		lhbrx
10 0 1101	11010xxx10		-
10 0 1110	11100xxx10		sthbrx
10 0 1111	11110xxx10		-
10 1 0000	00001xxx10		-
10 1 0001	00011xxx10		-
10 1 0010	00101xxx10		stqcx.
10 1 0011	00111xxx10		-
10 1 0100	01001xxx10		eciwx
10 1 0101	01011xxx10		-
10 1 0110	01101xxx10		ecowx
10 1 0111	01111xxx10		-
10 1 1000	10001xxx10		-
10 1 1001	10011xxx10		-
10 1 1010	10101xxx10		stbcx.
10 1 1011	10111xxx10		-
10 1 1100	11001xxx10		-
10 1 1101	11011xxx10		-
10 1 1110	11101xxx10		-
10 1 1111	11111xxx10		dcbz
11 0 0000	00000xxx11		lwzx
11 0 0001	00010xxx11		-
11 0 0010	00100xxx11		stwx
11 0 0011	00110xxx11		-
11 0 0100	01000xxx11		lhzx
11 0 0101	01010xxx11		lhax
11 0 0110	01100xxx11		sthx
11 0 0111	01110xxx11		-
11 0 1000	10000xxx11		lfsx
11 0 1001	10010xxx11		lfdx
11 0 1010	10100xxx11		stfsx
11 0 1011	10110xxx11		stfdx
11 0 1100	11000xxx11		lfdpx
11 0 1101	11010xxx11		lfiwax
11 1 1101	11010xxx11		lfiwzx
11 0 1110	11100xxx11		stfdpx
11 0 1111	11110xxx11		stfiwx
11 1 0000	00001xxx11		lwzux
11 1 0001	00011xxx11		-
11 1 0010	00101xxx11		stwux
11 1 0011	00111xxx11		-
11 1 0100	01001xxx11		lhzux
11 1 0101	01011xxx11		lhaux
11 1 0110	01101xxx11		sthux
11 1 0111	01111xxx11		-
11 1 1000	10001xxx11		lfsux
11 1 1001	10011xxx11		lfdux
11 1 1010	10101xxx11		stfsux
11 1 1011	10111xxx11		stfdux
11 1 1100	11001xxx11		-
11 1 1101	11011xxx11		-
11 1 1110	11101xxx11		-

If DSISR 47:53 is:	then it is either X-form opcode:	or D/DS/DQ/DQE-form opcode:	so the instruction is:
11 1 1111	11111xxx11		-

Book III-E:

Power ISA Operating Environment Architecture - Embedded Environment [Category: Embedded]

Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I describes computation modes, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the Power ISA Operating Environment Architecture.

1.2 32-Bit Implementations

Though the specifications in this document assume a 64-bit implementation, 32-bit implementations are permitted as described in Appendix C, “Guidelines for 64-bit Implementations in 32-bit Mode and 32-bit Implementations” on page 1249.

1.3 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system auxiliary processor enabled exception error handler” substitute “Auxiliary Processor Enabled Exception type Program interrupt”,
- For “system data storage error handler” substitute “Data Storage interrupt” or Data TLB Error interrupt” as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Illegal Instruction exception type Program interrupt” or “Unimplemented Operation exception type Program interrupt”, as appropriate.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt” or “Instruction TLB Error”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction exception type Program interrupt”.
- For “system service program” substitute “System Call interrupt”.
- For “system trap handler” substitute “Trap type Program interrupt”.

1.3.1 Definitions and Notation

The definitions and notation given in Books I and II are augmented by the following.

- Threaded processor, single-threaded processor, thread

  A threaded processor implements one or more “threads”, where a thread corresponds to the Book I/II concept of “processor”. That is, the definition of “thread” is the same as the Book I definition of “processor”, and “processor” as used in Books I and II can be thought of as either a single-threaded processor or as one thread of a multi-threaded processor. The only unqualified uses of “processor” in Book III are in resource names (e.g. Processor Identification Register); such uses should be regarded as meaning “threaded processor”. The threads of a multi-threaded processor typically share certain resources, such as the hardware components that execute certain kinds of instructions (e.g., Fixed-Point instructions), certain caches, the address translation mechanism, and certain hypervisor resources.

- Thread enabled, thread disabled

  A thread can be enabled or disabled. When enabled, the thread can prefetch and execute instructions; when disabled, prefetched instructions are discarded, and the thread cannot prefetch or execute instructions.

- Performed

  An explicit modification, by a thread T1, of a shared SPR (using ***mtspr***) or an entry in a shared TLB (using ***tlbwe***), is performed with respect to thread T2 when a read of the shared SPR or TLB

entry (using ***mfspr*** or ***tlbre*** respectively) by T2 will return the result of the modification (or of a subsequent modification). For T2, the effects of such a modification having been performed with respect to T2 are the same as if the ***mtspr*** or ***tlbwe*** were in T2's instruction stream at the point at which the modification was performed with respect to T2. T1 and T2 may be any threads that share the SPR or TLB with one another, and may be the same thread.

- real page

  A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size may range from 1KB to 1TB [MAV=1.0] or to 2TB [MAV=2.0].

- [MAV=x.x]

  Instructions and facilities are considered part of all MMU architecture versions unless otherwise marked. If a facility or section is marked with a specific MMU Architecture version x.x, that facility or all material in that section and its subsections are considered part of the specific MMU architecture version.

- "must"

  If the Embedded.Hypervisor category is not supported and privileged software violates a rule that is stated using the word "must", the results are undefined. If the Embedded.Hypervisor category is supported, the following applies.
  - If hypervisor software violates a rule that is stated using the word "must" (e.g., "this field must be set to 0"), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage using a TLB entry with a TGS value of 0, the results are undefined, and may include altering resources belonging to other partitions, causing the system to "hang", etc.
  - If supervisor software violates the requirements for storage control bit values or their alteration, the result of accessing the associated storage is undefined.
  - In other cases of violation of a rule that is stated using the word "must", the results are boundedly undefined unless otherwise stated.

Programming Note

Contrary to the general principle of partition isolation, the result of accessing storage associated with violations of the requirements for storage control bit values and their alteration is specified as "undefined". This is the only case where a guest operating system can cause "undefined" results. Embedded architecture does this as a hardware simplification because of the limited amount of code involved with storage control bits and because operating systems used in the Embedded environment can be tested and then controlled.

- context of a program

  The state (e.g., privilege and relocation) in which the program executes. The context is controlled by the contents of certain System Registers, such as the MSR, of certain lookaside buffers, such as the TLB, and of other resources.

- exception

  An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.

- interrupt

  The act of changing the machine state in response to an exception, as described in Chapter 7. "Interrupts and Exceptions" on page 1145.

- trap interrupt

  An interrupt that results from execution of a *Trap* instruction.

- Additional exceptions to the sequential execution model, beyond those described in the section entitled "Instruction Fetching" in Book I, are the following.
  - A reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)
  - A context-altering instruction is executed (Chapter 12. "Synchronization Requirements for Context Alterations" on page 1235). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.

- hardware

Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.

- hypervisor privileged (or hypervisor-privileged)

  If category E.HV is implemented, this term describes an instruction, register, or facility that is available only when the thread is in hypervisor state. Otherwise, this term describes an instruction, register, or facility that is available only when the thread is in supervisor state.

- privileged state and supervisor state

  Used interchangeably to refer to a state in which privileged facilities are available.

- guest state

  A state used to run software under control of a hypervisor program in which hypervisor-privileged facilities are not available.

- guest supervisor state

  A state which is in both the guest state and the supervisor state.

- problem state and user mode

  Used interchangeably to refer to a state in which privileged facilities are not available.

- volatile
  Bits in a register or array (e.g., TLB) are considered volatile if they may change even if not explicitly modified by software.

- directed

  In a hypervised system, the attribute of an interrupt that execution occurs in the guest supervisor or hypervisor state as described in Section 2.3.1, “Directed Interrupts”.

- /, //, ///, ... denotes a field that is reserved in an instruction, in a register, or in an architected storage table.

- ?, ??, ???, ... denotes a field that is implementation-dependent in an instruction, in a register, or in an architected storage table.

1.3.2 Reserved Fields

Some fields of certain architected registers may be written to automatically by the hardware, e.g., Reserved bits in System Registers. When the hardware writes to such a register, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) specifically being updated are modified.
- Contents of reserved fields are either preserved or written as zero.

The reader should be aware that reading and writing of some of these registers (e.g., the MSR) can occur as a side effect of processing an interrupt and of returning from an interrupt, as well as when requested explicitly by the appropriate instruction (e.g., ***mtmsr*** instruction).

1.4 General Systems Overview

The hardware contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Most implementations also contain data and instruction caches. Instructions fall into the following classes:

- instructions executed in the Branch Facility
- instructions executed in the Fixed-Point Facility
- instructions executed in the Floating-Point Facility
- instructions executed in the Vector Facility
- instructions executed in an Auxiliary Processor
- other instructions

Almost all instructions executed in the Branch Facility, Fixed-Point Facility, Floating-Point Facility, and Vector Facility are nonprivileged and are described in Book I. Book I may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions executed in an Auxiliary Processor are implementation-dependent. Instructions related to the supervisor mode, control of hardware resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

1.5 Exceptions

The following augments the exceptions defined in Book I that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSR_{FP}=0 (Floating-Point Unavailable interrupt)
- execution of an instruction that causes a debug event (Debug interrupt).
- the execution of an auxiliary processor instruction when the auxiliary processor is unavailable (Auxiliary Processor Unavailable interrupt)
- the execution of a Vector, SPE, or Embedded Floating-Point instruction when MSR_{SPV}=0 (SPE/Embedded Floating-Point/Vector Unavailable interrupt)

1.6 Synchronization

The synchronization described in this section refers to the state of the thread that is performing the synchronization.

1.6.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. The context synchronizing operations include the ***dnh*** instruction, the ***isync*** instruction, the *System Linkage* instructions, and most interrupts (see Section 7.1). Also, the combination of disabling and enabling a thread is context-synchronizing for the thread being enabled (See Section 3).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of ***dnh***, ***isync*** does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.
3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated.
4. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 7.9, “Exception Priorities” on page 1190).
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 6.5, “Performing Operations Out-of-Order”.)

   The operation ensures that all explicit modifications of shared SPRs (***mtspr***) and shared TLBs (***tlbwe***), caused by instructions that precede the operation, have been performed with respect to all other threads that share the SPR or TLB.

Programming Note

The context established by a context synchronizing instruction includes modifications to certain resources that were performed with respect to the context synchronizing thread before the operation was initiated. The resources in this case include shared SPRs that contain program context such as LPIDR as well as TLBs shared by other threads.

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.6.2.

Unlike the *Synchronize* instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for ***isync*** (and ***sync***; see Section 1.6.2) because all other execution synchronizing operations also alter context.

1.6.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.6.1). ***sync*** is treated like ***isync*** with respect to item 2. The execution synchronizing instructions are ***sync***, ***mtmsr*** and all context synchronizing instructions.

Programming Note

Unlike a context synchronizing operation, an execution synchronizing instruction does not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

Chapter 2. Logical Partitioning [Category: Embedded.Hypervisor]

2.1 Overview

The Embedded.Hypervisor category permits threads and portions of real storage to be assigned to logical collections called *partitions*, such that a program executing in one partition cannot interfere with any program executing in a different partition. This isolation can be provided for both problem state and privileged state programs, by using a layer of trusted software, called a *hypervisor* program (or simply a "hypervisor"), and the resources provided by this facility to manage system resources. The collection of software that runs in a given partition and its associated resources is called a *guest*. The guest normally includes an operating system (or other system software) running in privileged state and its associated processes running in the problem state under the management of the hypervisor. The thread is in the *guest state* when a guest is executing and is in the *hypervisor state* when the hypervisor is executing. The thread is executing in the guest state when MSR_{GS}=1.

The number of partitions supported is implementation-dependent.

A thread is assigned to one partition at any given time. A thread can be assigned to any given partition without consideration of the physical configuration of the system (e.g. shared registers, caches, organization of the storage hierarchy), except that threads that share certain hypervisor resources may need to be assigned to the same partition. Additionally, certain resources may be utilized by the guest at the discretion of the hypervisor. Such usage may cause interference between partitions and the hypervisor should allocate those resources accordingly. The primary registers and facilities used to control Logical Partitioning are listed below and described in the following subsections. Other facilities associated with Logical Partitioning are described within the appropriate sections within this Book.

An instruction that is *hypervisor privileged* must be execute in the hypervisor state ($MSR_{GS\ PR}$ = 0b00). If an attempt is made to execute a hypervisor-privileged instruction in the guest supervisor state ($MSR_{GS\ PR}$ = 0b10), an Embedded Hypervisor Privilege exception occurs. A register that is *hypervisor privileged* may only be accessed in the hypervisor state ($MSR_{GS\ PR}$ = 0b00). If a hypervisor-privileged register is accessed in the guest supervisor state ($MSR_{GS\ PR}$ = 0b10), an Embedded Hypervisor Privilege exception occurs.

When $MSR_{GS\ PR}$ = 0b01 or $MSR_{GS\ PR}$ = 0b11, the thread is in problem (user) state. The resources (instructions and registers) that are available are generally the same when MSR_{PR} = 0b1 regardless of the state of MSR_{GS}, however when $MSR_{GS\ PR}$ = 0b11 some interrupts are directed to the guest supervisor state. When $MSR_{GS\ PR}$ = 0b01 interrupts are always directed to the hypervisor (see Section 2.3.1, "Directed Interrupts").

Category Embedded.Hypervisor changes the operating system programming model to allow for easier virtualization, while retaining a default backward compatible mode in which an operating system written for hardware not implementing this category will still operate as before without using the Logical Partitioning facilities.

Category Embedded.Hypervisor requires that Category: Embedded.Processor Control is also supported.

2.2 Registers

Registers specific to Logical Partitioning and hypervisor control are defined in this section. Other registers which are hypervisor privileged or have hypervisor-only fields appear and are described in other sections in this Book.

2.2.1 Register Mapping

To facilitate better performance for operating systems executing in the guest supervisor state, some Special Purpose Register (SPR) accesses are redirected to analogous guest-state SPRs. An SPR is said to be *mapped* if this redirection takes place when executing in guest supervisor state. These guest-state SPRs separate performance critical state of the hypervisor and the operating system executing in guest supervisor

state. The mapping of these register accesses allows the same programming model to be used for an operating system running in the guest supervisor state or in the hypervisor state.

For example, when a ***mtspr*** SRR0,r5 instruction is executed in guest supervisor state, the access to SRR0 is mapped to GSRR0. This produces the same operation as executing ***mtspr*** GSRR0,r5

Programming Note

Since accesses to the mapped SPRs are automatically mapped to the appropriate guest-accessible SPR, guest supervisor software should use the original SPRs for accessing these registers (i.e. SRR0, not GSRR0). This facilitates using the same code in hypervisor or guest state.

SPR accesses that are mapped in guest supervisor state are listed in Table 1.

SPR Accessed	SPR Mapped to	Type of Access
DEC	GDEC	***mtspr***, ***mfspr***
DECAR	GDECAR	***mtspr***
TCR	GTCR	***mtspr***, ***mfspr***
TSR	GTSR	***mtspr***, ***mfspr***
SRR0	GSRR0	***mtspr***, ***mfspr***
SRR1	GSRR1	***mtspr***, ***mfspr***
EPR	GEPR	***mfspr***
ESR	GESR	***mtspr***, ***mfspr***
DEAR	GDEAR	***mtspr***, ***mfspr***
PIR	GPIR	***mfspr***
SPRG0	GSPRG0	***mtspr***, ***mfspr***
SPRG1	GSPRG1	***mtspr***, ***mfspr***
SPRG2	GSPRG2	***mtspr***, ***mfspr***
SPRG3	GSPRG3	***mtspr***, ***mfspr***
[1] If an implementation permits problem state read access to SPRG3, the problem state read access is remapped to GSPRG3.		

Table 1: Mapped SPRs

2.2.2 Logical Partition Identification Register (LPIDR)

The Logical Partition Identification Register (LPIDR) contains the Logical Partition ID (LPID) currently in use for the thread. The format of the LPIDR is shown in Figure 1 below.

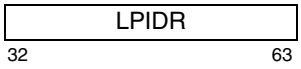


Figure 1. Logical Partition Identification Register

The LPIDR is part of the virtual address. During address translation, its content is compared to the TLPID field in the TLB entry to determine a matching TLB entry.

The LPIDR is hypervisor privileged.

The 12 least significant bits of LPIDR contain the LPID value. All 12 bits do not need to be implemented. Unimplemented bits should read as zero. The number of implemented bits is reported in $MMUCFG_{LPIDSIZE}$

2.3 Interrupts and Exceptions

2.3.1 Directed Interrupts

Category Embedded.Hypervisor introduces new interrupt semantics. Interrupts are *directed* to either the guest state or the hypervisor state. The state to which interrupts are directed determines which SPRs are used to form the vector address, which save/restore registers are used to capture the thread state at the time of the interrupt, and which registers are used to post exception status.

- If IVORs [Category: Embedded.Phased-Out] are supported, interrupts directed to the guest state use the Guest Interrupt Vector Prefix Register (GIVPR) to determine the high-order 48 bits of the vector address and use Guest Interrupt Vector Registers (GIVORs) to provide the low-order 16 bits (of which the last 4 bits are 0).
- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, interrupts directed to the guest state use the Guest Interrupt Vector Prefix Register (GIVPR) to determine the high-order 52 bits of the vector address and use the 12-bit exception vector offsets (described in Section 7.2.15) to provide the low-order 12 bits (of which the last 5 bits are 0).
- If IVORs [Category: Embedded.Phased-Out] are supported, interrupts directed to the Embedded hypervisor state use the IVPR for the upper 48 bits of the address and the IVORs for the lower 16 bits of the address.
- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, interrupts directed to the Embedded hypervisor state use one of the following for the interrupt vector address.
  - If the Machine Check Interrupt Vector Prefix Register (see Section 7.2.18.4) is supported and the interrupt is a Machine Check, MCIVPR provides the high-order 52 bits of the vector address and the 12-bit exception vector offsets (described in Section 7.2.15) provides the low-order 12 bits (of which the last 5 bits are 0).
  - Otherwise, IVPR provides the high-order 52 bits of the vector address and the 12-bit exception vector offsets (described in Section

7.2.15) provides the low-order 12 bits (of which the last 5 bits are 0).

Interrupts that are directed to the guest state use GSRR0/GSRR1 registers to save the context at interrupt time. Interrupts directed to the embedded hypervisor state use SRR0/SRR1, with the exception of Guest Processor Doorbell interrupts which use GSRR0/GSRR1.

All interrupts are directed to the hypervisor except when the processor is already in guest state (MSR_{GS}=1) and:

- The interrupt is a system call and the LEVEL field is 0.
- The interrupt is a Data TLB Error, Instruction TLB Error, Data Storage Interrupt, Instruction Storage Interrupt, or External Input Interrupt and the corresponding bit in the EPCR to direct these interrupts to guest state is a 1 and the interrupt is not caused by either a Virtualization Fault or a TLB Ineligible Exception.

2.3.2 Hypervisor Service Interrupts

Two interrupts exist as mechanisms for the hypervisor to provide services to the guest.

The Embedded Hypervisor Privilege Interrupt occurs when guest supervisor state attempts execution of a hypervisor-privileged instruction or attempts to access a hypervisor-privileged resource. This can be used by the hypervisor to provide virtualization services for the guest. The Embedded Hypervisor Privilege Interrupt is described in Section 7.6.31, “Embedded Hypervisor Privilege Interrupt [Category: Embedded.Hypervisor]”. An Embedded Hypervisor Privilege Interrupt will also occur if a ***ehpriv*** instruction is executed regardless of the thread state.

The Embedded Hypervisor System Call Interrupt occurs when an ***sc*** instruction is executed and LEV=1. The ***sc*** instruction is described in Section 4.3.1, “System Linkage Instructions”.

2.4 Instruction Mapping

When executing in the guest supervisor state ($MSR_{GS\ PR}$ = 0b10), execution of an ***rfi*** instruction is mapped to ***rfgi*** and the ***rfgi*** instruction is executed in place of the ***rfi***. The mapping of these instructions allows the same programming model to be used for an operating system running in the guest supervisor state or in the hypervisor state.

Chapter 3. Thread Control [Category: Embedded Multi-Threading]

3.1 Overview

The Thread Control facility permits the hypervisor to control and monitor the execution, priority, and other aspects of threads.

3.2 Thread Identification Register (TIR)

The layout of the TIR is shown in below.

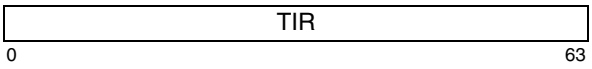


Figure 2. Thread Identification Register

The TIR is a 64-bit read-only register that can be used to distinguish the thread from other threads on a multi-threaded processor. Threads are numbered sequentially, with valid values ranging from 0 to t-1, where t is the number of threads implemented. A thread for which TIR = n is referred to as "thread n."

The TIR is hypervisor privileged.

3.3 Thread Enable Register (TEN)

The layout of the TEN is shown in below.

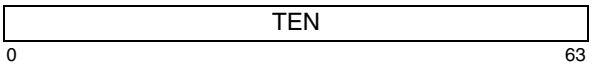


Figure 3. Thread Enable Register

The TEN is a 64-bit register. For t < T, where T is the number of threads supported by the implementation, bit 63-t corresponds to thread t. When TEN_{63-t} is 0, thread t is disabled. When TEN_{63-t} is 1, thread t is enabled.

Software is permitted to write any value to bits 0:63-T; a subsequent reading of these bits always returns 0.

The TEN can be accessed using two SPR numbers.

- When SPR 438 (Thread Enable Set, or TENS) is written, threads for which the corresponding bit in TENS is 1 are enabled; threads for which the corresponding bit in TENS is 0 are unaffected.
- When SPR 439 (Thread Enable Clear, or TENC) is written, threads for which the corresponding bit in TENC is 1 are disabled; threads for which the corresponding bit in TENC is 0 are unaffected.

When each SPR is read, the current value of the TEN is returned.

The TEN is hypervisor privileged.

Architecture Note

32-bit implementations are limited to 32 threads. In 64-bit mode, the high order 32 bits of TEN are not modified when written. See Section 1.5.2, "Modes [Category: Embedded]", in Book I.

Programming Note

Software can determine the number of threads supported by the implementation by setting each progressively higher-order bit to 1, and testing whether a subsequent read returns a 1. Because this operation enables the thread, software should ensure that an acceptable instruction sequence is located at the thread's starting effective address. (See Section 8.3, "Thread State after Reset".)

3.4 Thread Enable Status Register (TENSR)

The layout of the TENSR is shown in below.

Figure 4. Thread Enable Status Register

The TENSR is a 64-bit read-only register. Bit 63-t of the TENSR corresponds to thread t. The contents of the TENSR are equal to the contents of the TEN, except that when TEN_{63-t} changes from 1 to 0, $TENSR_{63-t}$ does not change from 1 to 0 until thread t is disabled.

The TENSR is hypervisor privileged.

Architecture Note

32-bit implementations are limited to 32 threads.

3.5 Disabling and Enabling Threads

The combination of disabling and enabling a thread is context-synchronizing for the thread being enabled. Steps 1-3 of context synchronization (see Section 1.6.1) occur as a result of the thread being disabled, and updates to SPRs and shared TLBs caused by preceding instructions executed by the thread occur. When all updates to these shared SPRs and shared TLBs have been performed with respect to all other threads on a multi-threaded processor, the TENSR bit corresponding to the disabled thread is set to 0.

Asynchronous interrupts that occur after the thread is disabled are pended until the thread is enabled.

When a thread is enabled by setting the TEN bit corresponding to the thread to 1, the thread begins execution at the next instruction to be executed when it was disabled or at the effective address specified by the INIA [Category: Embedded Multi-threading.Thread Management] if the INIA corresponding to the thread was written while the thread was disabled.

Programming Note

The architecture provides no method to make a thread's updates to shared storage visible to other threads before it is disabled. Similarly, the architecture provides no method to make updates to shared storage made while a thread is disabled visible to a thread when it is subsequently enabled.

Programming Note

When thread T1 disables other threads, Tn, it sets the TEN bits corresponding to Tn to 0s. In order to ensure that all updates to shared SPRs and shared TLBs caused by instructions being performed by threads Tn have been performed with respect to all threads on a multi-threaded processor, thread T1 reads the TENSR until all the bits corresponding to the disabled threads, Tn, are 0s.

3.6 Sharing of Multi-Threaded Processor Resources

The PVR and TEN must be shared among all threads of a multi-threaded processor. Various other resources are allowed to be shared among threads. Programs that modify shared resources must be aware of such sharing, and must allow for the fact that changes to these resources may affect more than one thread.

Resources that may be shared are grouped into the following five groups of related resources. If any of the resources in a group are shared among threads, all of the resources in the group must be shared.

- ATB, ATBL, ATBU [Category: ATB]
- IVORs [Category: Phased-Out]
- IVPR
- TB, TBL, TBU
- MMUCFG, MMUCSR0, TLB, TLBnCFG, TLBnCFG2, TLBnEPT, [Category: Embedded.Hypervisor.LRAT]: LRAT, LRATCFG, LRATCFG2

If the implementation requires all threads to be in the same partition, the following additional groups of resources may be shared. If any of the resources in a group are shared among threads, all of the resources in the group must be shared.

- DAC1, DAC2, IAC1, IAC2, IAC3
- EHCSR [Category: Embedded.Hypervisor]
- GIVORs [Category: Phased-Out]
- GIVPR [Category: Embedded.Hypervisor]
- LPIDR [Category: Embedded.Hypervisor]

Certain implementation-dependent registers, instruction and Data Caches, and implementation-dependent look-aside information may also be shared.

The set of resources that is shared is implementation-dependent.

Programming Note

When software executing in thread T1 writes a new value in an SPR (***mtspr***) that is shared with other threads, or explicitly writes to an entry in a shared TLB (***tlbwe***), either of the following sequences of operations can be performed in order to ensure that the write operation has been performed with respect to other threads.

Sequence 1

- Disable all other threads (see Section 3.5)
- Write to the shared SPR (***mtspr***) or to the shared TLB (***tlbwe***)
- Perform a context synchronizing operation
- Enable the previously-disabled threads

In the above sequence, the context synchronizing operation ensures that the write operation has been performed with respect to all other threads that share the SPR or TLB; the enabling of other threads ensures that subsequent instructions of the enabled threads use the new SPR or TLB value since enabling a thread is a context synchronizing operation.

Sequence 2

- All threads are put in hypervisor state and begin polling a storage flag
- The thread updating the SPR or TLB does the following:
  - Writes to the SPR (***mtspr***) or the TLB (***tlbwe***)
  - Sets a storage flag indicating the write operation was done
  - Performs a context synchronizing operation
- When other threads see the updated storage flag, they perform context synchronizing operations.

In the above sequence, the context synchronizing operation by the thread that writes to the SPR or TLB ensures that the write operation has been performed with respect to all other threads that share the SPR or TLB; the context synchronizing operation by the other threads ensure that subsequent instructions for these threads use the updated value.

3.7 Thread Management Facility [Category: Embedded Multi-threading.Thread Management]

The thread management facility enables software to control features related to threads. The capabilities provided allow software, for a disabled thread, to specify the address of the instruction to be executed when the thread is enabled. Other implementation-dependent capabilities may also be provided.

3.7.1 Initialize Next Instruction Address Registers

The Initialize Next Instruction Address (INIAn, where n = 0..63) registers are 64-bit write-only registers that can be used to specify the effective address of the instruction to be executed when a currently-disabled thread is enabled. INIAn corresponds to thread n.

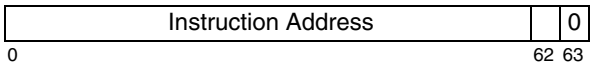


Figure 5. Initialize Next Instruction Address Register

Bit 63 is always 0. Bit 62 is part of the Instruction Address if Category: VLE is supported; otherwise bit 62 is always 0.

When the INIA is written in 32-bit mode, bits 0:31 are set to 0s.

The initial value of all INIAs is x0xFFFF_FFFF_FFFF_FFFC.

3.7.2 Thread Management Instructions

Move To Thread Management Register *XFX-form*

mttmr TMR,RS

31	RS	tmr	494	/
0	6	11	21	31

```
n ← tmr5:9 || tmr0:4
TMR(n) ← (RS)
```

The TMR field denotes a Thread Management Register, encoded as shown in the table below. The contents of register RS are placed into the designated Thread Management Register.

decimal	TMR[1]	Register Name
	$tmr_{5:9}$ $tmr_{0:4}$	
320	01010 00000	$INIA_0$
...	.....	...
383	10111 11111	$INIA_{63}$

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.

Figure 6. Thread Management Register Numbers

All values of the TMR field not shown in Figure 6 are implementation-specific.

An implementation only provides INIA registers corresponding to its implemented threads. Execution of this instruction specifying a TMR number that is not defined for the implementation causes an Illegal Instruction type Program interrupt if $MSR_{GS\ PR}$=0b00.

This instruction is hypervisor privileged.

Special Registers Altered:

See above

Chapter 4. Branch Facility

4.1 Branch Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Facility that are not covered in Book I.

4.2 Branch Facility Registers

4.2.1 Machine State Register

The MSR (MSR) is a 32-bit register. MSR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). This register defines the state of the thread. The MSR can also be modified by the ***mtmsr***, ***rfi***, ***rfci***, ***rfdi*** [Category: Embedded.Enhanced Debug], ***rfmci, rfgi*** [Category: Embedded.Hypervisor], ***wrtee*** and ***wrteei*** instructions and interrupts. It can be read by the ***mfmsr*** instruction.

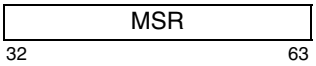


Figure 7. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Bit **Description**

32 ***Computation Mode*** (CM)

0 The thread runs in 32-bit mode.
1 The thread runs in 64-bit mode.

33 Reserved

34 Implementation-dependent

35 ***Guest State*** (GS)

[Category: Embedded.Hypervisor]
0 The thread is in hypervisor state if MSR_{PR} = 0.
1 The thread is in guest state.

MSR_{GS} cannot be changed unless thread is in the hypervisor state.

Virtualized Implementation Note

In a virtualized implementation, MSR_{GS} will be 1.

36 Implementation-dependent

37 ***User Cache Locking Enable*** (UCLE)
[Category: Embedded Cache Locking.User Mode]

0 *Cache Locking* instructions are privileged.
1 *Cache Locking* instructions can be executed in user mode (MSR_{PR}=1).

If Category: Embedded Cache Locking.User Mode is not supported, this bit is treated as reserved.

38 ***SP/Embedded Floating-Point/Vector Available*** (SPV)

[Category: Signal Processing]:
0 The thread cannot execute any SP instructions except for the ***brinc*** instruction.
1 The thread can execute all SP instructions.

[Category: Vector]:
0 The thread cannot execute any Vector instruction.
1 The thread can execute Vector instructions.

39 Reserved

40 ***VSX Available*** (VSX)

0 The thread cannot execute any VSX instructions, including VSX loads, stores, and moves.
1 The thread can execute VSX instructions.

Programming Note

An application binary interface defined to support Category: Vector-Scalar operations should also specify a requirement that `MSR.FP` and `MSR.VEC` be set to 1 whenever `MSR.VSX` is set to 1.

41:45 Reserved

46 ***Critical Enable*** (CE)

0 Critical Input, Watchdog Timer, Guest Processor Doorbell Critical <E.HV>, and Processor Doorbell Critical interrupts are disabled.
1 Critical Input, Watchdog Timer, Guest Processor Doorbell Critical <E.HV>, and Processor Doorbell Critical interrupts are enabled.

[Category: Embedded.Hypervisor]
Critical level interrupts with the exception of Guest Processor Doorbell Critical are enabled regardless of the state of MSR_{CE} when MSR_{GS} = 1.

47 Reserved

48 ***External Enable*** (EE)

0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell <E.HV> and Embedded Performance Monitor <E.PM> interrupts are disabled.
1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell <E.HV>, and Embedded Performance Monitor <E.PM> interrupts are enabled.

[Category: Embedded.Hypervisor]
When an interrupt that is maskable by MSR_{EE} is directed to the hypervisor state, the interrupt is enabled if MSR_{EE}=1 or MSR_{GS}=1 except for Guest Processor Doorbell which is enabled if MSR_{EE}=1 and MSR_{GS}=1. When an interrupt that is maskable by MSR_{EE} is directed to the guest supervisor state, the interrupt is enabled if MSR_{EE}=1 and MSR_{GS}=1. Also, see the EXTGS bit in Section 4.2.3.

49 ***Problem State*** (PR)

0 The thread is in privileged state (supervisor state).
1 The thread is in problem state (user mode).

MSR_{PR} also affects storage access control, as described in Section 6.7.6

50 ***Floating-Point Available*** (FP)
[Category: Floating-Point]

0 The thread cannot execute any floating-point instructions, including floating-point loads, stores and moves.
1 The thread can execute floating-point instructions.

51 ***Machine Check Enable*** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

[Category: Embedded.Hypervisor]
Machine Check interrupts with the exception of Guest Processor Doorbell Machine Check are enabled regardless of the state of MSR_{ME} when MSR_{GS} = 1.

52 ***Floating-Point Exception Mode 0*** (FE0)
[Category: Floating-Point]

(See below)

53 Implementation-dependent

54 ***Debug Interrupt Enable*** (DE)

0 Debug interrupts are disabled
1 Debug interrupts are enabled if $DBCR0_{IDM}$=1

Virtualized Implementation Note

In a virtualized implementation, when MSR_{DE}=1, the registers SPRG9, DSRR0, and DSRR1 are volatile.

55 ***Floating-Point Exception Mode*** 1 (FE1)
[Category: Floating-Point]

(See below)

56 Reserved

57 Reserved

58 ***Instruction Address Space*** (IS)

0 The thread directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry).
1 The thread directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).

59 ***Data Address Space*** (DS)

0 The thread directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry).
1 The thread directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).

60 Implementation-dependent

61 ***Performance Monitor Mark*** (PMM)
[Category: Embedded.Performance Monitor]

0 Disable statistics gathering on marked processes.
1 Enable statistics gathering on marked processes

See Appendix D for additional information.

62 Reserved

63 Reserved

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

FE0	FE1	Mode
0	0	Ignore Exceptions
0	1	Imprecise Nonrecoverable
1	0	Imprecise Recoverable
1	1	Precise

See Section 8.3 for the initial state of the MSR.

[Category:Embedded.Hypervisor]
Some bits in the MSR can only be changed when the thread is in hypervisor state or the MSRP register has been configured to allow changes in the guest supervisor state. See Section 4.2.2, “Machine State Register Protect Register (MSRP)”.

Programming Note

A Machine State Register bit that is reserved may be altered by ***rfi***/***rfci/rfmci/rfdi*** [Category:Embedded.Enhanced Debug]/***rfgi*** [Category:Embedded.Hypervisor].

4.2.2 Machine State Register Protect Register (MSRP)

The Machine State Register Protect Register (MSRP) controls whether certain bits in the Machine State Register (MSR) can be modified in guest supervisor state. In addition, the MSRP impacts the behavior of cache locking and performance monitor instructions in guest state, as described below. The format of the MSRP is shown in Figure 8 below.

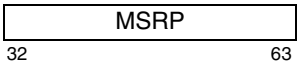


Figure 8. Machine State Register Protect Register

The MSRP is used to prevent guest supervisor state program from modifying the UCLE, DE, or PMM bits in the MSR. The MSRP bits UCLEP, DEP, and PMMP control whether the guest can change the corresponding MSR bits UCLE, DE, and PMM, respectively. When the MSRP bit associated with a corresponding MSR bit is 0, any operation in guest privileged state is allowed to modify that MSR bit, whether from an instruction that modifies the MSR, or from an interrupt which is taken in the guest supervisor state. When the MSRP bit associated with a corresponding MSR bit is 1 no operation in guest privileged state is allowed to modify that MSR bit (i.e., it remains unchanged), whether from an instruction that modifies the MSR, or from an interrupt from the guest state which is taken in the guest supervisor state.

These bits are interpreted as follows:

Bit **Definition**

32:36 Reserved

37 ***User Cache Lock Enable Protect*** (UCLEP)
[Category: ECL]

0 MSR_{UCLE} can be modified in guest supervisor state.
1 MSR_{UCLE} cannot be modified in guest supervisor state and guest state cache locking using ***dcbtls***, ***dcbtstls***, ***dcblc***, ***dcblq.***, ***icbtls***, ***icblq.***, and ***icblc*** is affected as descrbed later in this section.".

38:53 Reserved

54 ***Debug Enable Protect*** (DEP)

0 MSR_{DE} can be modified in guest supervisor state.
1 MSR_{DE} cannot be modified in guest supervisor state.

Virtualized Implementation Note

It is the responsibility of the hypervisor to ensure that $DBCR0_{EDM}$ is consistent with usage of DEP.

55:60 Reserved

61 ***Performance Monitor Mark Protect*** (PMMP)
[Category: E.PM]

0 MSR_{PMM} can be modified in guest supervisor state.
1 MSR_{PMM} cannot be modified in guest supervisor state and guest state accesses to Performance Monitor Registers using ***mfpmr*** and ***mtpmr*** are affected as described later in this section.

62:63 Reserved

The MSRP is hypervisor privileged.

A context synchronizing operation must be performed following a change to MSRP to ensure that its changes are visible in the current context.

The behavior of cache locking instructions (***dcbtls***, ***dcbtstls***, ***dcblc***, ***dcblq.***, ***icbtls***, ***icblq.***, ***icblc***) in guest privileged state is dependent on the setting of $MSRP_{UCLEP}$. When $MSRP_{UCLEP}$ = 0, cache locking instructions are permitted to execute normally in the guest privileged state. When $MSRP_{UCLEP}$ = 1, cache locking instructions are not permitted to execute in the guest privileged state and cause an Embedded Hypervisor Privilege exception. [Category: ECL]

The behavior of Performance Monitor instructions (***mtpmr***, ***mfpmr***) is dependent on the setting of $MSRP_{PMMP}$. When $MSRP_{PMMP}$ = 0, Performance Monitor instructions are permitted to execute normally in the guest state. When $MSRP_{PMMP}$ = 1, Performance Monitor instructions are not permitted to execute normally in the guest state. Execution of a ***mfpmr*** instruction which specifies a user Performance Monitor register produces a value of 0 in the destination GPR. In the guest supervisor state (MSR_{PR} = 0 and MSR_{GS} = 1), execution of any ***mtpmr*** instruction or execution of a ***mfpmr*** instruction which specifies a privileged Performance Monitor Register produces an Embedded Hypervisor Privilege exception. [Category: E.PM]

Programming Note

Setting the MSRP to 0 at initialization allows guest state access to $MSR_{UCLE,DE,PMM}$ and the associated cache locking and performance monitor facilities.

4.2.3 Embedded Processor Control Register (EPCR)

The Embedded Processor Control Register (EPCR) provides general controls for both privileged and hypervisor privileged facilities. The format of the EPCR is shown in Figure 1 below.

Figure 9. Embedded Processor Control Register

These bits are interpreted as follows:

Bit **Definition**

32 ***External Input Interrupt Directed to Guest State*** (EXTGS)
[Category: Embedded.Hypervisor]
Controls whether an External Input Interrupt is taken in the guest supervisor state or the hypervisor state.

0 External Input Interrupts are directed to the hypervisor state. External Input Interrupts pend until MSR_{GS}=1 or MSR_{EE}=1.
1 External Inputs interrupts are directed to the guest supervisor state. External Input interrupts pend until MSR_{GS}=1 and MSR_{EE}=1.

33 ***Data TLB Error Interrupt Directed to Guest State*** (DTLBGS)
[Category: Embedded.Hypervisor]
Controls whether a Data TLB Error Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state.

0 Data TLB Error Interrupts that occur in the guest state are directed to the hypervisor state.
1 Data TLB Error Interrupts that occur in the guest state are directed to the guest supervisor state.

34 ***Instruction TLB Error Interrupt Directed to Guest State*** (ITLBGS)
[Category: Embedded.Hypervisor]
Controls whether an Instruction TLB Error Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state.

0 Instruction TLB Error Interrupts that occur in the guest state are directed to the hypervisor state.
1 Instruction TLB Error Interrupts that occur in the guest state are directed to the guest supervisor state.

35 ***Data Storage Interrupt Directed to Guest State*** (DSIGS)
[Category: Embedded.Hypervisor]
Controls whether a Data Storage Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state, except for an interrupt caused by a TLB Ineligible exception <E.PT>.

0 Data Storage Interrupts that occur in the guest state are directed to the hypervisor state.
1 Data Storage Interrupts that occur in the guest state are directed to the guest supervisor state except that a Data Storage Interrupt due to a TLB Ineligible exception <E.PT> is directed to the hypervisor state, regardless of the existence of other exceptions that cause a Data Storage interrupt.

36 ***Instruction Storage Interrupt Directed to Guest State*** (ISIGS)
[Category: Embedded.Hypervisor]
Controls whether an Instruction Storage Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor

state, except for an interrupt caused by a TLB Ineligible exception <E.PT>.

0 Instruction Storage Interrupts that occur in the guest supervisor state are directed to the hypervisor state.
1 Instruction Storage Interrupts that occur in the guest state are directed to the guest supervisor state.

37 ***Disable Embedded Hypervisor Debug*** (DUVD)
[Category: Embedded.Hypervisor]
Controls whether Debug Events occur in the hypervisor state.

0 Debug events can occur in the hypervisor state.
1 Debug events, except for the Unconditional Debug Event, are suppressed in the hypervisor state. It is implementation-dependent whether the Unconditional Debug Event is suppressed.

38 ***Interrupt Computation Mode*** (ICM)
[Category: 64-bit]
If category E.HV is implemented, this bit controls the computational mode of the thread when an interrupt occurs that is directed to the hypervisor state. At interrupt time, $EPCR_{ICM}$ is copied into MSR_{CM} if the interrupt is directed to the hypervisor state.

If category E.HV is not implemented, then this bit controls the computational mode of the thread when any interrupt occurs. At interrupt time, $EPCR_{ICM}$ is copied into MSR_{CM}.

0 Interrupts will execute in 32-bit mode.
1 Interrupts will execute in 64-bit mode.

39 ***Guest Interrupt Computation Mode*** (GICM)
[Category: Embedded.Hypervisor]
[Corequisite Category: 64-bit]
Controls the computational mode of the thread when an interrupt occurs that is directed to the guest supervisor state. At interrupt time, $EPCR_{GICM}$ is copied into MSR_{CM} if the interrupt is directed to the guest supervisor state

0 Interrupts will execute in 32-bit mode.
1 Interrupts will execute in 64-bit mode.

40 ***Disable Guest TLB Management Instructions*** (DGTMI)
[Category: Embedded.Hypervisor]
Controls whether guest supervisor state can execute any TLB management instructions.

0 ***tlbsrx.*** and ***tlbwe*** (for a Logical to Real Address translation hit) are allowed to execute normally when $MSR_{GS,PR}$ = 0b10.
1 ***tlbsrx.*** and ***tlbwe*** always cause an Embedded Hypervisor Privilege Interrupt when $MSR_{GS,PR}$ = 0b10.

41 ***Disable MAS Interrupt Updates for Hypervisor*** (DMIUH)
[Category: Embedded.Hypervisor]
Controls whether MAS registers are updated by hardware when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.

0 MAS registers are set as described in Table 11 on page 1116 when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.
1 MAS registers updates as described in Table 11 are disabled and MAS registers are left unchanged when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.

42 ***Performance Monitor Interrupt Directed to Guest State*** (PMGS)
[Category: Embedded.Hypervisor]
[Corequisite Category: Embedded.Performance Monitor]
Controls whether a Performance Monitor Interrupt that occurs in the guest state is taken in the guest supervisor state or the hypervisor state.

0 Performance Monitor Interrupts that occur in the guest state are directed to the hypervisor state.
1 Performance Monitor Interrupts that occur in the guest state are directed to the guest supervisor state.

43:63 Reserved

This register is hypervisor privileged.

Engineering Note

EPCR only needs to be implemented if Category: Embedded.Hypervisor or category 64-bit are implemented.

4.3 Branch Facility Instructions

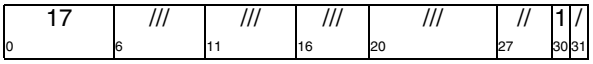
4.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

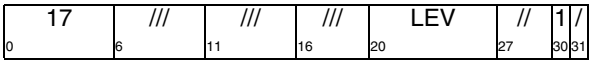
The *System Call* instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

System Call **SC-form**

sc



sc LEV
[Category:Embedded.Hypervisor]




```
if LEV = 0 then

   if MSR_GS = 1 then
       GSRR0 ←_iea CIA + 4
       GSRR1 ← MSR
       if IVORs supported then
          NIA ← GIVPR0:47 || GIVOR848:59 || 0b0000
       else
          NIA ← GIVPR0:51||0x0120
       MSR ← new_value (see below)

   else
       SRR0 ←_iea CIA + 4
       SRR1 ← MSR
       if IVORs supported then
          NIA ← IVPR0:47 || IVOR848:59 || 0b0000
       else
          NIA ← IVPR0:51||0x0120
       MSR ← new_value (see below)

else if LEV = 1 then
   SRR0 ←_iea CIA + 4
   SRR1 ← MSR
   if IVORs supported then
       NIA ← IVPR0:47 || IVOR4048:59 || 0b0000
   else
       NIA ← IVPR0:51||0x300
   MSR ← new_value (see below)
```

If category E.HV is not implemented, the System Call instruction behaves as if MSR_{GS} = 0 and LEV = 0.

If MSR_{GS} = 0 or if LEV = 1, the effective address of the instruction following the *System Call* instruction is placed into SRR0 and the contents of the MSR are copied into SRR1. Otherwise, the effective address of the instruction following the *System Call* instruction is placed into GSRR0 and the contents of the MSR are copied into GSRR1.

If LEV=0, a System Call interrupt is generated. If LEV=1, an Embedded Hypervisor System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 7.6.10 and Section 7.6.30.

If LEV=0 and the thread is in guest state, the interrupt causes the next instruction to be fetched from the effective address based on one of the following.

- $GIVPR_{0:47}$||$GIVOR8_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x120 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

If LEV=0 and the thread is in hypervisor state, the interrupt causes the next instruction to be fetched from the effective address based on one of the following.

- $IVPR_{0:47}$||$IVOR8_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x120 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

If LEV=1, the interrupt causes the next instruction to be fetched from the effective address based on one of the following.

- $IVPR_{0:47}$||$IVOR40_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x300 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

This instruction is context synchronizing.

Special Registers Altered:

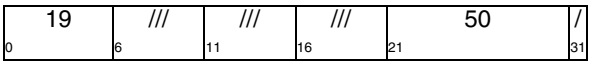
SRR0 GSRR0 SRR1 GSRR1 MSR

Programming Note

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an ***sc*** mnemonic with one operand as the basic form, and an ***sc*** mnemonic with no operand as the extended form. In the extended form, the LEV operand is omitted and assumed to be 0.

Return From Interrupt *XL-form*

rfi




```
MSR ← SRR1
NIA ←iea SRR0 0:61 || 0b00
```

The ***rfi*** instruction is used to return from a base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of SRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $SRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the applicable save/restore register 0 by the interrupt processing mechanism (see Section 7.6 on page 1161) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in SRR0 at the time of the execution of the ***rfi***).

This instruction is privileged and context synchronizing.

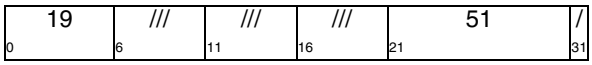
[Category:Embedded.Hypervisor]
When ***rfi*** is executed in guest state, the instruction is mapped to ***rfgi*** and ***rfgi*** is executed instead.

Special Registers Altered:

MSR

Return From Critical Interrupt *XL-form*

rfci




```
MSR ← CSRR1
NIA ←iea CSRR0 0:61 || 0b00
```

The ***rfci*** instruction is used to return from a critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of CSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $CSRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or CSRR0 by the interrupt processing mechanism (see Section 7.6 on page 1161) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in CSRR0 at the time of the execution of the ***rfci***).

This instruction is hypervisor privileged and context synchronizing.

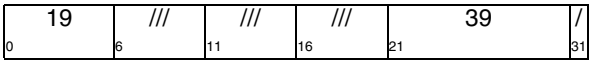
Special Registers Altered:

MSR

Return From Debug Interrupt *X-form*

rfdi

[Category: Embedded.Enhanced Debug]




```
MSR ← DSRR1
NIA ←iea DSRR0 0:61 || 0b00
```

The ***rfdi*** instruction is used to return from a Debug interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of DSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $DSRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, CSRR0, or DSRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in DSRR0 at the time of the execution of the ***rfdi***).

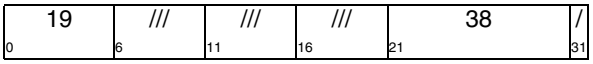
This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

MSR

Return From Machine Check Interrupt *XL-form*

rfmci




```
MSR ← MCSRR1
NIA ←iea MCSRR0 0:61 || 0b00
```

The ***rfmci*** instruction is used to return from a Machine Check class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of MCSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $MCSRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0, CSRR0, MCSRR0, or DSRR0 [Category: Embedded.Enhanced Debug] by the interrupt processing mechanism (see Section 7.6 on page 1161) is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in MCSRR0 at the time of the execution of the ***rfmci***).

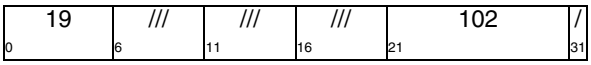
This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

MSR

Return From Guest Interrupt XL-form

rfgi [Category:Embedded.Hypervisor]




```
newmsr ← GSRR1
if MSR_GS = 1 then
     newmsr_GS,WE ← MSR_GS
     prots ← MSRP_UCLEP,DEP,PMMP
     newmsr ← prots & MSR | ~prots & newmsr
MSR ← newmsr
NIA ←_iea GSRR0_0:61 || 0b00
```

The ***rfgi*** instruction is used to return from a guest state base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of Guest Save/Restore Register 1 are placed into the MSR. If the ***rfgi*** is executed in the guest supervisor state ($MSR_{GS\ PR}$ = 0b10), the bit MSR_{GS} is not modified and the bits $MSR_{UCLE\ DE\ PMM}$ are modified only if the associated bits in the Machine State Register Protect (MSRP) Register are set to 0. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $GSRR0_{0:61}$||0b00. (Note: VLE behavior may be different; see Book VLE.) If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the associated save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in GSRR0 at the time of the execution of the ***rfgi***).

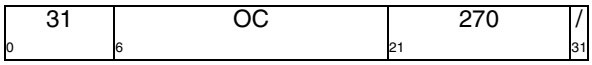
This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Embedded Hypervisor Privilege XL-form

ehpriv OC [Category: Embedded.Hypervisor]



The ***ehpriv*** instruction generates an Embedded Hypervisor Privilege Exception resulting in an Embedded Hypervisor Privilege Interrupt.

The OC field may be used by hypervisor software to provide a facility for emulated virtual instructions.

Special Registers Altered:

None

Programming Note

The ***ehpriv*** instruction is analogous to a guaranteed illegal instruction encoding in that it guarantees that an Embedded Hypervisor Privilege exception is generated. The instruction is useful for programs that need to communicate information to the hypervisor software, particularly as a means for implementing breakpoint operations in a hypervisor managed debugger.

Programming Note

ehpriv serves as both a basic and an extended mnemonic. The Assembler will recognize an ***ehpriv*** mnemonic with one operand as the basic form, and an ***ehpriv*** mnemonic with no operand as the extended form. In the extended form, the OC operand is omitted and assumed to be 0.

Chapter 5. Fixed-Point Facility

5.1 Fixed-Point Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Facility that are not covered in Book I.

5.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 1054) and ***mtspr*** (page 1053) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

5.3 Fixed-Point Facility Registers

5.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the hardware. The contents of the PVR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PVR is privileged; write access is not provided.

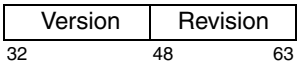


Figure 10. Processor Version Register

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

Version A 16-bit number that identifies the version of the implementation. Different version numbers indicate major differences between implementations, such as which optional facilities and instructions are supported.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between implementations having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA Architecture process. Revision numbers are assigned by an implementation-defined process.

5.3.2 Chip Information Register

The Chip Information Register (CIR) is a 32-bit read-only register that contains a value identifying the manufacturer and other characteristics of the chip on which the processor is implemented. The contents of the CIR can be copied to a GPR by the ***mfspr*** instruction. Read access to the CIR is privileged; write access is not provided.

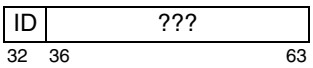


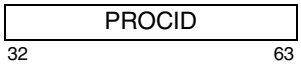
Figure 11. Chip Information Register

Bit	Description
32:35	**Manufacturer ID** (ID) A four-bit field that identifies the manufacturer of the chip.
36:63	Implementation-dependent.

5.3.3 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the thread from other threads in the system. The contents of the PIR can be read using ***mfspr*** and written using ***mtspr***. Read access to the PIR is privileged; write access, if provided, is hypervisor privileged.

[Category:Embedded.Hypervisor]
Read accesses to the PIR in guest supervisor state are mapped to the GPIR.



Bits	Name	Description
32:63	PIR	Thread ID

Figure 12. Processor Identification Register

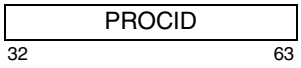
The means by which the PIR is initialized are implementation-dependent.

Programming Note

The PIR can be used to identify the thread globally among all threads in a system that contains multiple threads. This facilitates more efficient usage of the Processor Control facility (see Section 11).

5.3.4 Guest Processor Identification Register [Category:Embedded.Hypervisor]

The Guest Processor Identification Register (GPIR) is a 32-bit register that contains a value that can be used to distinguish the thread from other threads in the system. The contents of the GPIR can be read using ***mfspr*** and written using ***mtspr***. Read access to the GPIR is privileged; write access, if provided, is hypervisor privileged.



Bits	Name	Description
32:63	GPIR	Thread ID

Figure 13. Guest Processor Identification Register

The means by which the GPIR is initialized are implementation-dependent.

Programming Note

mfspr *RT,PIR* should be used to read GPIR in guest supervisor state. See Section 2.2.1, “Register Mapping”.

Engineering Note

Some previous implementations did not allow write access to the PIR. This allows the hypervisor to easily track changes to the GPIR. If write access is allowed to the PIR, write access should be allowed to the GPIR and should only be available to the hypervisor.

5.3.5 Program Priority Register 32-bit

Privileged programs may set a wider range of program priorities in the PRI field of PPR32 than may be set by problem-state programs (see Section 3.1 of Book II). Problem-state programs may only set values in the range of 0b010 to 0b100. Privileged programs may set values in the range of 0b001 to 0b110. Hypervisor software may also set 0b111. If a program attempts to set a value that is not available to it, the PRI field remains unchanged. The values and their corresponding meanings are as follows.

001 very low
010 low
011 medium low
100 medium
101 medium high
110 high
111 very high

5.3.6 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

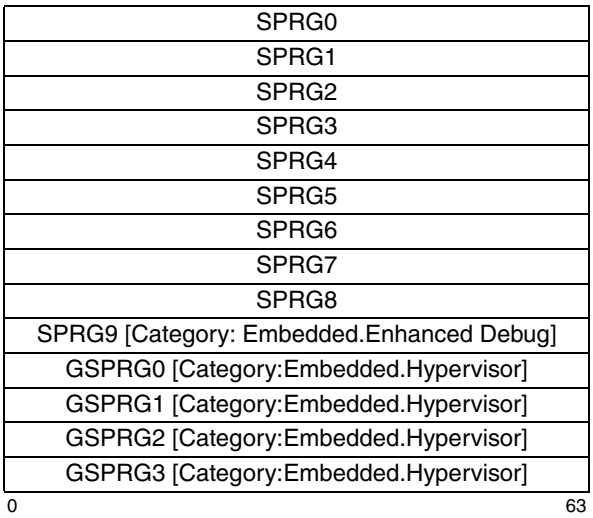


Figure 14. Special Purpose Registers

Programming Note

USPRG0 was made a 32-bit register and renamed to VRSAVE; see Sections 3.2.3 and 5.3.3 of Book I.

SPRG0 through SPRG2

These 64-bit registers can be accessed only in supervisor mode.

[Category:Embedded.Hypervisor]
Access to these registers in guest supervisor state is mapped to GSPRG0 through GSPRG2.

SPRG3

This 64-bit register can be read in supervisor mode and can be written only in supervisor mode. It is implementation-dependent whether or not this register can be read in user mode.

[Category:Embedded.Hypervisor]
Access to this register in guest state is mapped to GSPRG3.

SPRG4 through SPRG7

These 64-bit registers can be written only in supervisor mode. These registers can be read in supervisor and user modes.

SPRG8 through SPRG9

These 64-bit registers can be accessed only in supervisor mode.

Programming Note

The intended use for SPRG9 is for internal debug exception handling.

GSPRG0 through GSPRG2

[Category:Embedded.Hypervisor]
These 64-bit registers can be accessed only in supervisor mode.

GSPRG3

[Category:Embedded.Hypervisor]
This 64-bit register can be read in supervisor mode and can be written only in supervisor mode. If an implementation permits problem state read access to SPRG3, the problem state read access is remapped to GSPRG3.

SPRGi or GSPRGi can be read using ***mfspr*** and written using ***mtspr***.

Programming Note

mfspr *RT,SPRGi* should be used to read GSPRGi in guest state. ***mtspr*** *SPRGi,RS* should be used to write GSPRGi in guest state. See Section 2.2.1, “Register Mapping”.

5.3.7 External Process ID Registers [Category: Embedded.External PID]

The External Process ID Registers provide capabilities for loading and storing General Purpose Registers and performing cache management operations using a supplied context other than the context normally used by the programming model.

Two SPRs describe the context for loading and storing using external contexts. The External Process ID Load Context (EPLC) Register provides the context for *External Process ID Load* instructions, and the External Process ID Store Context (EPSC) Register provides the context for *External Process ID Store* instructions. Each of these registers contains a PR (privilege) bit, an AS (address space) bit, a Process ID, a GS (guest state) bit <E.HV>, and an LPID <E.HV>. Changes to the EPLC or the EPSC Register require that a context synchronizing operation be performed prior to using any *External Process ID* instructions that use these registers.

External Process ID instructions that use the context provided by the EPLC register include ***lbepx***, ***lhepx***, ***lwepx***, ***ldepx***, ***dcbtep***, ***dcbfep***, ***dcbstep***, ***icbiep***, ***lfdepx***, ***evlddepx***, ***lvepx***, and ***lvepxl*** and those that use the context provided by the EPSC register include ***stbepx***, ***sthepx***, ***stwepx***, ***stdepx***, ***dcbzep***, ***stfdepx***, ***evstddepx***, ***stvepx***, ***stvepxl***, and ***dcbtstep***. Instruction definitions appear in Section 5.4.3.

System software configures the EPLC register to reflect the Process ID, AS, PR, GS <E.HV>, and LPID <E.HV> state from the context that it wishes to perform loads from and configures the EPSC register to reflect the Process ID, AS, PR, GS <E.HV>, and LPID <E.HV> state from the context it wishes to perform stores to. Software then issues *External Process ID* instructions to manipulate data as required.

When the an *External Process ID Load* instruction is executed, it uses the context information in the EPLC Register instead of the normal context with respect to address translation and storage access control. $EPLC_{EPR}$ is used in place of MSR_{PR}, $EPLC_{EAS}$ is used in place of MSR_{DS}, $EPLC_{EPID}$ is used in place of PID, $EPLC_{EGS}$. is used in place of MSR_{GS} <E.HV>, and $EPLC_{ELPID}$. is used in place of LPIDR <E.HV>. Similarly, when the an *External Process ID Store* instruction is executed, it uses the context information in the EPSC Register instead of the normal context with respect to address translation and storage access control. $EPSC_{EPR}$ is used in place of MSR_{PR}, $EPSC_{EAS}$ is used in place of MSR_{DS}, $EPSC_{EPID}$ is used in place of PID, $EPSC_{EGS}$. is used in place of MSR_{GS} <E.HV>, and $EPSC_{ELPID}$. is used in place of LPIDR <E.HV>. Translation occurs using the new substituted values.

If the TLB lookup is successful, the storage access control mechanism grants or denies the access using context information from $EPLC_{EPR}$ or $EPSC_{EPR}$ for loads and stores respectively. If access is not granted, a Data Storage interrupt occurs, and the ESR_{EPID} bit is set to 1. If the operation was a *Store*, the ESR_{ST} bit is also set to 1.

5.3.7.1 External Process ID Load Context (EPLC) Register

The EPLC register contains fields to provide the context for *External Process ID Load* instructions.

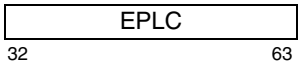


Figure 15. External Process ID Load Context Register

These bits are interpreted as follows:

Bit	Definition
32	***External Load Context PR Bit*** (EPR) Used in place of MSR_{PR} by the storage access control mechanism when an *External Process ID Load* instruction is executed. 0 Supervisor mode 1 User mode
33	***External Load Context AS Bit*** (EAS) Used in place of MSR_{DS} for translation when an *External Process ID Load* instruction is executed, and, if this *Load* instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of MSR_{DS}. 0 Address space 0 1 Address space 1
34	***External Load Context GS Bit*** (EGS) [Category: Embedded.Hypervisor] Used in place of MSR_{GS} for translation when an *External Process ID Load* instruction is executed. 0 Hypervisor state 1 Guest state
35	Reserved
36:47	***External Load Context LPID Value*** (ELPID) [Category:Embedded.Hypervisor] Used in place of LPIDR register for translation

Programming Note

When a ***mtspr*** instruction is executed that targets EPLC, the EGS and ELPID fields are only modified if the thread is in hypervisor state.

when an *External Process ID Load* instruction is executed.

Programming Note

When a ***mtspr*** instruction is executed that targets EPLC, the EGS and ELPID fields are only modified if the thread is in hypervisor state.

48:49 Reserved

50:63 ***External Load Context Process ID Value*** (EPID)
Used in place of the Process ID register value for translation when an external Process ID *Load* instruction is executed, and, if this *Load* instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of PID contents.

5.3.7.2 External Process ID Store Context (EPSC) Register

The EPSC register contains fields to provide the context for *External Process ID Store* instructions. The field encoding is the same as the EPLC Register.

Figure 16. External Process ID Store Context Register

These bits are interpreted as follows:

Bits **Definition**

32 ***External Store Context PR Bit*** (EPR)
Used in place of MSR_{PR} by the storage access control mechanism when an *External Process ID Store* instruction is executed.

0 Supervisor mode
1 User mode

33 ***External Store Context AS Bit*** (EAS)
Used in place of MSR_{DS} for translation when an *External Process ID Store* instruction is executed, and, if this *Store* instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of MSR_{DS}.

0 Address space 0
1 Address space 1

34 ***External Store Context GS Bit*** (EGS)

[Category: Embedded.Hypervisor]
Used in place of MSR_{GS} for translation when an *External Process ID Store* instruction is executed.

0 Hypervisor state
1 Guest state

Programming Note

When a ***mtspr*** instruction is executed that targets EPSC, the EGS and ELPID fields are only modified if the thread is in hypervisor state.

35 Reserved

36:47 ***External Store Context LPID Value*** (ELPID)
[Category:Embedded.Hypervisor]
Used in place of LPIDR register for translation when an *External Process ID Store* instruction is executed.

Programming Note

When a ***mtspr*** instruction is executed that targets EPSC, the EGS and ELPID fields are only modified if the thread is in hypervisor state.

48:49 Reserved

50:63 ***External Store Context Process ID Value*** (EPID)
Used in place of the Process ID register value for translation when an external PID *Store* instruction is executed, and, if this *Store* instruction causes a Data TLB Error interrupt, loaded into MAS registers in place of PID contents.

5.4 Fixed-Point Facility Instructions

5.4.1 Move To/From System Register Instructions

The *Move To Special Purpose Register* and *Move From Special Purpose Register* instructions are described in Book I, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in supervisor mode. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are implementation-dependent. In the descriptions of these instructions given below, the "defined" SPR numbers are the SPR numbers shown in Table 17 and the implementation-specific SPR numbers that are implemented, and similarly for "defined" registers.

Extended mnemonics

Extended mnemonics are provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand; see Appendix B.

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
1	00000 00001	XER	no	no	64	B
3	00000 00011	DSCR	no	no	64	STM
8	00000 01000	LR	no	no	64	B
9	00000 01001	CTR	no	no	64	B
17	00000 10001	DSCR	yes	yes	64	STM
22	00000 10110	DEC	yes[9]	yes[9]	32	B
26	00000 11010	SRR0	yes[9]	yes[9]	64	B
27	00000 11011	SRR1	yes[9]	yes[9]	64	B
48	00001 10000	PID	yes	yes	32	E
53	00001 10101	GDECAR	hypv[3]	no	32	E.HV
54	00001 10110	DECAR	hypv[8]	-	32	E
55	00001 10111	MCIVPR	hypv[8]	hypv[8]	64	E
56	00001 11000	LPER	hypv[8]	hypv[8]	64	E.HV; E.PT
57	00001 11001	LPERU	hypv[8]	hypv[8]	32	E.HV; E.PT
58	00001 11010	CSRR0	hypv[8]	hypv[8]	64	E
59	00001 11011	CSRR1	hypv[8]	hypv[8]	32	E
60	00001 11100	GTSRWR	hypv[3]	no	32	E.HV
61	00001 11101	DEAR	yes[9]	yes[9]	64	E
62	00001 11110	ESR	yes[9]	yes[9]	32	E
63	00001 11111	IVPR	hypv[8]	hypv[8]	64	E
256	01000 00000	VRSAVE	no	no	32	B
259	01000 00011	SPRG3	-	no	64	B
260-263	01000 001xx	SPRG[4-7]	-	no	64	E
268	01000 01100	TB	-	no	64	B
269	01000 01101	TBU	-	no	32[5]	B
272-275	01000 100xx	SPRG[0-3]	yes[9]	yes[9]	64	B
276-279	01000 101xx	SPRG[4-7]	yes	yes	64	E
282	01000 11010	EAR	hypv[4]	hypv[4]	32	EC
284	01000 11100	TBL	hypv[4]	-	32	B
283	01000 11011	CIR	-	hypv[4]	32	E
285	01000 11101	TBU	hypv[4]	-	32	B
286	01000 11110	PIR	hypv[8]	yes[9]	32	E
287	01000 11111	PVR	-	yes	32	B
304	01001 10000	DBSR	hypv[5,8]	hypv[8]	32	E
306	01001 10010	DBSRWR	hypv[3]	-	32	E.HV
307	01001 10011	EPCR	hypv[3]	hypv[3]	32	E.HV,(E;64)
308	01001 10100	DBCR0	hypv[8]	hypv[8]	32	E
309	01001 10101	DBCR1	hypv[8]	hypv[8]	32	E
310	01001 10110	DBCR2	hypv[8]	hypv[8]	32	E

Figure 17. SPR Numbers (Sheet 1 of 4)

decimal	SPR[1]		Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$	$spr_{0:4}$		mtspr	mfspr		
311	01001	10111	MSRP	hypv[3]	hypv[3]	32	E.HV
312	01001	11000	IAC1	hypv[8]	hypv[8]	64	E
313	01001	11001	IAC2	hypv[8]	hypv[8]	64	E
314	01001	11010	IAC3	hypv[8]	hypv[8]	64	E
315	01001	11011	IAC4	hypv[8]	hypv[8]	64	E
316	01001	11100	DAC1	hypv[8]	hypv[8]	64	E
317	01001	11101	DAC2	hypv[8]	hypv[8]	64	E
336	01010	10000	TSR	yes[9]	yes[9]	32	E
338	01010	10010	LPIDR	hypv[3]	hypv[3]	32	E.HV
339	01010	10011	MAS5	hypv[3]	hypv[3]	32	E.HV
340	01010	10100	TCR	yes[9]	yes[9]	32	E
341	01010	10101	MAS8	hypv[3]	hypv[3]	32	E.HV
342	01010	10110	LRATCFG	-	hypv[3]	32	E.HV.LRAT
343	01010	10111	LRATPS	-	hypv[3]	32	E.HV.LRAT
344-347	01010	110xx	TLB[0-3]PS	-	hypv[3]	32	E.HV
348	01010	11100	MAS5||MAS6	hypv[3]	hypv[3]	64	E.HV; 64
349	01010	11101	MAS8||MAS1	hypv[3]	hypv[3]	64	E.HV; 64
350	01010	11110	EPTCFG	hypv[8]	hypv[8]	32	E.PT
368-371	01011	100xx	GSPRG0-3	yes	yes	64	E.HV
372	01011	10100	MAS7||MAS3	yes	yes	64	E; 64
373	01011	10101	MAS0||MAS1	yes	yes	64	E; 64
374	01011	10110	GDEC	yes	yes	32	E.HV
375	01011	10111	GTCR	yes	yes	32	E.HV
376	01011	11000	GTSR	yes	yes	32	E.HV
378	01011	11010	GSRR0	yes	yes	64	E.HV
379	01011	11011	GSRR1	yes	yes	32	E.HV
380	01011	11100	GEPR	yes	yes	32	E.HV;EXP
381	01011	11101	GDEAR	yes	yes	64	E.HV
382	01011	11110	GPIR	hypv[3]	yes	32	E.HV
383	01011	11111	GESR	yes	yes	32	E.HV
400-415	01100	1xxxx	IVOR0-15	hypv[8]	hypv[8]	32	E
432-435	01101	100xx	IVOR38-41	hypv[8]	hypv[8]	32	E.HV
436	01101	10100	IVOR42	hypv[8]	hypv[8]	32	E.HV.LRAT
437	01101	10101	TENSR	-	hypv[8]	64	E.MT
438	01101	10110	TENS	hypv[8]	hypv[8]	64	E.MT
439	01101	10111	TENC	hypv[8]	hypv[8]	64	E.MT
440-441	01101	1100x	GIVOR2-3	hypv[3]	yes	32	E.HV
442	01101	11010	GIVOR4	hypv[3]	yes	32	E.HV
443	01101	11011	GIVOR8	hypv[3]	yes	32	E.HV
444	01101	11100	GIVOR13	hypv[3]	yes	32	E.HV
445	01101	11101	GIVOR14	hypv[3]	yes	32	E.HV
446	01101	11110	TIR	-	hypv[9]	64	E.MT
447	01101	11111	GIVPR	hypv[3]	yes	64	E.HV
474	11010	01110	GIVOR10	hypv[3]	yes	32	E.HV
475	11011	01110	GIVOR11	hypv[3]	yes	32	E.HV
476	11100	01110	GIVOR12	hypv[3]	yes	32	E.HV
512	10000	00000	SPEFSCR	no	no	32	SP
526	10000	01110	ATB/ATBL	-	no	64	ATB
527	10000	01111	ATBU	-	no	32	ATB
528	10000	10000	IVOR32	hypv[8]	hypv[8]	32	SP
529	10000	10001	IVOR33	hypv[8]	hypv[8]	32	SP
530	10000	10010	IVOR34	hypv[8]	hypv[8]	32	SP
531	10000	10011	IVOR35	hypv[8]	hypv[8]	32	E.PM
532	10000	10100	IVOR36	hypv[8]	hypv[8]	32	E.PC
533	10000	10101	IVOR37	hypv[8]	hypv[8]	32	E.PC

Figure 17. SPR Numbers (Sheet 2 of 4)

decimal	SPR[1]		Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$	$spr_{0:4}$		mtspr	mfspr		
570	10001	11010	MCSRR0	hypv[8]	hypv[8]	64	E
571	10001	11011	MCSRR1	hypv[8]	hypv[8]	32	E
572	10001	11100	MCSR	hypv[8]	hypv[8]	64	E
574	10001	11110	DSRR0	yes	yes	64	E.ED
575	10001	11111	DSRR1	yes	yes	32	E.ED
604	10010	11100	SPRG8	hypv[8]	hypv[8]	64	E
605	10010	11101	SPRG9	yes	yes	64	E.ED
624	10011	10000	MAS0	yes	yes	32	E
625	10011	10001	MAS1	yes	yes	32	E
626	10011	10010	MAS2	yes	yes	64	E
627	10011	10011	MAS3	yes	yes	32	E
628	10011	10100	MAS4	yes	yes	32	E
630	10011	10110	MAS6	yes	yes	32	E
631	10011	10111	MAS2U	yes	yes	32	E
688-691	10101	100xx	TLB[0-3]CFG	-	hypv[8]	32	E
702	10101	11110	EPR	-	yes[9]	32	EXP
808	11001	01000	reserved[10]	no	no	na	B
809	11001	01001	reserved[10]	no	no	na	B
810	11001	01010	reserved[10]	no	no	na	B
811	11001	01011	reserved[10]	no	no	na	B
898	11100	00010	PPR32	no	no	32	B
924	11100	11100	DCDBTRL	-[5]	hypv[8]	32	E.CD
925	11100	11101	DCDBTRH	-[5]	hypv[8]	32	E.CD
926	11100	11110	ICDBTRL	-[6]	hypv[8]	32	E.CD
927	11100	11111	ICDBTRH	-[6]	hypv[8]	32	E.CD
944	11101	10000	MAS7	yes	yes	32	E
947	11101	10011	EPLC	yes	yes	32	E.PD
948	11101	10100	EPSC	yes	yes	32	E.PD
979	11110	10011	ICDBDR	-[6]	hypv[8]	32	E.CD
1012	11111	10100	MMUCSR0	hypv[8]	hypv[8]	32	E
1015	11111	10111	MMUCFG	-	hypv[8]	32	E

- This register is not defined for this instruction.

1 Note that the order of the two 5-bit halves of the SPR number is reversed.

2 See Section 1.3.5 of Book I. If multiple categories are listed separated by a semicolon, all the listed categories must be implemented in order for the other columns of the line to apply. A comma separates two alternatives, and takes precedence over a semicolon; e.g., the EPCR (E.HV,E;64) must be implemented if either (a) category E.HV is implemented or (b) the implementation is Embedded and supports the 64-bit category.

3 This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E).

4 If the Embedded.Hypervisor category is supported, this register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E). Otherwise, the register is privileged.

5 The register can be written by the ***dcread*** instruction.

6 The register can be written by the ***icread*** instruction.

7 The register is Category: Phased-in.

8 If the Embedded.Hypervisor category is supported, this register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E). Otherwise, the register is privileged for Embedded.

9 If the Embedded.Hypervisor category is supported, this register is a hypervisor resource and can be accessed by this instruction only in hypervisor state, and guest references to the register are redirected to the corresponding guest register (see Chapter 2 of Book III-E). Otherwise the register is privileged.

10 Accesses to these SPRs are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I.

Figure 17. SPR Numbers (Sheet 3 of 4)

decimal	SPR[1]		Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$	$spr_{0:4}$		mtspr	mfspr		
All SPR numbers that are not shown above and are not implementation-specific are reserved.							

Figure 17. SPR Numbers (Sheet 4 of 4)

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      SPR(n) ← (RS)
    else
      SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

spr_0=1 if and only if writing the register is privileged. Execution of this instruction specifying a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt; if MSR_{PR}=0: boundedly undefined results

If the SPR number is set to a value that is shown in Figure 17 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were reserved.

Special Registers Altered:

See Figure 17

Compiler and Assembler Note

For the ***mtspr*** and ***mfspr*** instructions, the SPR number coded in assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 12. “Synchronization Requirements for Context Alterations” on page 1235.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      RT ← SPR(n)
    else
      RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 17. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs" in Book I. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

spr_0=1 if and only if reading the register is privileged. Execution of this instruction specifying a defined and privileged register when MSR_{PR}=1 causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if spr_0=0: boundedly undefined results
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: boundedly undefined results

If the SPR field contains a value that is shown in Figure 17 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were reserved.

Special Registers Altered:

None

Note

See the Notes that appear with ***mtspr***.

Move To Device Control Register XFX-form

mtdcr DCRN,RS
[Category: Embedded.Device Control]

31	RS	dcr	451	/
0	6	11	21	31

```
DCRN ← dcr5:9 || dcr0:4
DCR(DCRN) ← (RS)
```

Let DCRN denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of register RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of bits 32:63 of (RS) are placed into the Device Control Register.

This instruction is privileged.

Special Registers Altered:

Implementation-dependent.

Move To Device Control Register Indexed X-form

mtdcrx RA,RS
[Category: Embedded.Device Control]

31	RS	RA	///	387	/
0	6	11	16	21	31

```
DCRN ← (RA)
DCR(DCRN) ← (RS)
```

Let the contents of register RA denote a Device Control Register. (The supported Device Control Registers supported are implementation-dependent.)

The contents of register RS are placed into the designated Device Control Register. For 32-bit Device Control Registers, the contents of $RS_{32:63}$ are placed into the Device Control Register.

The specification of Device Control Registers using ***mtdcrx***, ***mtdcrux*** (see Book I), and ***mtdcr*** is implementation-dependent. For example, *mtdcr 105,r2* and *mtdcrux r1,r2* (where register r1 contains the value 105) may not produce identical results on an implementation.

This instruction is privileged.

Special Registers Altered:

Implementation-dependent.

Move From Device Control Register XFX-form

mfdcr RT,DCRN
[Category: Embedded.Device Control]

31	RT	dcr	323	/
0	6	11	21	31

```
DCRN ← dcr5:9 || dcr0:4
RT ← DCR(DCRN)
```

Let DCRN denote a Device Control Register. (The supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into register RT. For 32-bit Device Control Registers, the contents of the Device Control Register are placed into bits 32:63 of RT. Bits 0:31 of RT are set to 0.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent.

Move From Device Control Register Indexed X-form

mfdcrx RT,RA
[Category: Embedded.Device Control]

31	RT	RA	///	259	/
0	6	11	16	21	31

```
DCRN ← (RA)
RT ← DCR(DCRN)
```

Let the contents of register RA denote a Device Control Register (the supported Device Control Registers are implementation-dependent.)

The contents of the designated Device Control Register are placed into register RT. For 32-bit Device Control Registers, the contents of bits 32:63 of the designated Device Control Register are placed into RT. Bits 0:31 of RT are set to 0.

The specification of Device Control Registers using ***mfdcrx*** and ***mfdcrux*** (see Book I) compared to the specification of Device Control Registers using ***mfdcr*** is implementation-dependent. For example, *mfdcr r2,105* and *mfdcrx r2,r1* (where register r1 contains the value 105) may not produce identical results on an implementation or between implementations. Also, accessing privileged Device Control Registers in supervisor mode with ***mfdcrux*** is implementation-dependent.

This instruction is privileged.

Special Registers Altered:
Implementation-dependent.

Move To Machine State Register X-form

mtmsr RS

31	RS	///	///	146	/
0	6	11	16	21	31

```
newmsr ← (RS)32:63
if MSRCM = 0 & newmsrCM = 1 then NIA0:31 ← 0
if MSRGS = 1 then
   newmsrGS WE ← MSRGS WE
   prots0:31 ← 0
   protsUCLEP DEP PMMP ← MSRPUCLEP DEP PMMP
   newmsr ← prots & MSR | ~prots & newmsr
MSR ← newmsr
```

The contents of register $RS_{32:63}$ are placed into the MSR. If the thread is changing from 32-bit mode to 64-bit mode, the next instruction is fetched from $^{32}0||NIA_{32:63}$.

This instruction is privileged and execution synchronizing.

In addition, alterations to the EE or CE bits are effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing an ***mtmsr*** that sets MSR_{EE} to 1 will cause the External interrupt to be taken before the next instruction is executed, if no higher priority exception exists. Likewise, if MSR_{CE}=0 and a Critical Input interrupt is pending, executing an ***mtmsr*** that sets MSR_{CE} to 1 will cause the Critical Input interrupt to be taken before the next instruction is executed if no higher priority exception exists. (See Section 7.6 on page 1161.)

[Category:Embedded.Hypervisor]
GS, WE and bits protected with MSRP are only modified if ***mtmsr*** is executed in hypervisor state.

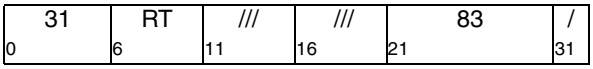
Special Registers Altered:
MSR

Programming Note

For a discussion of software synchronization requirements when altering certain MSR bits please refer to Chapter 12.

Move From Machine State Register X-form

mfmsr RT



RT ← $^{32}0$ || MSR

The contents of the MSR are placed into bits 32:63 of register RT and bits 0:31 of RT are set to 0.

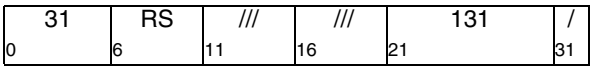
This instruction is privileged.

Special Registers Altered:

None

Write MSR External Enable X-form

wrtee RS



MSR_{EE} ← $(RS)_{48}$

The content of $(RS)_{48}$ is placed into MSR_{EE}.

Alteration of the MSR_{EE} bit is effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing a wrtee instruction that sets MSR_{EE} to 1 will cause the External interrupt to occur before the next instruction is executed, if no higher priority exception exists (Section 7.9, “Exception Priorities” on page 1190).

This instruction is privileged.

Special Registers Altered:

MSR

Write MSR External Enable Immediate X-form

wrteei E

31	///	///	E	///	163	/
0	6	11	16	17	21	31

MSR_{EE} ← E

The value specified in the E field is placed into MSR_{EE}.

Alteration of the MSR_{EE} bit is effective as soon as the instruction completes. Thus if MSR_{EE}=0 and an External interrupt is pending, executing a wrtee instruction that sets MSR_{EE} to 1 will cause the External interrupt to occur before the next instruction is executed, if no higher priority exception exists (Section 7.9, “Exception Priorities” on page 1190).

This instruction is privileged.

Special Registers Altered:

MSR

Programming Note

wrtee and ***wrteei*** are used to provide atomic update of MSR_{EE}. Typical usage is:

```
mfmsr     Rn            #save EE in (Rn)48
wrteei    0             #turn off EE
mfmsr     Rn            #save EE in (Rn)48
wrteei    0             #turn off EE
  :       :             :
  :                     #code with EE disabled
wrtee     Rn    #restore EE without altering
                #other MSR bits that might
                #have changed
```

5.4.2 OR Instruction

or Rx,Rx,Rx can be used to set PPR_{PRI} (see Figure 2 in Section 3.1 of Book II) as shown in Figure 18. PPR_{PRI} remains unchanged if the privilege state of the thread executing the instruction is lower than the privilege indicated in the figure. (The encodings available to problem-state programs, as well as encodings for additional shared resource hints not shown here, are described in Section 3.2 of Book II.)

Rx	$PPR32_{PRI}$	Priority	Privi-leged
31	001	very low	yes
1	010	low	no
6	011	medium low	no
2	100	medium	no
5	101	medium high	yes
3	110	high	yes
7	111	very high	hypv[1]

[1] If the Embedded.Hypervisor category is supported, this value is hypervisor privileged. Otherwise, the value is privileged.

Figure 18. Priority levels for *or Rx,Rx,Rx*

5.4.3 External Process ID Instructions [Category: Embedded.External PID]

External Process ID instructions provide capabilities for loading and storing General Purpose Registers and performing cache management operations using a supplied context other than the context normally used by translation.

The EPLC and EPSC registers provide external contexts for performing loads and stores. The EPLC and the EPSC registers are described in Section 5.3.7.

If an Alignment interrupt, Data Storage interrupt, or a Data TLB Error interrupt, occurs while attempting to execute an *External Process ID* instruction, ESR_{EPID} is set to 1 indicating that the instruction causing the interrupt was an *External Process ID* instruction; any other applicable ESR bits are also set.

Load Byte by External Process ID Indexed *X-form*

lbepx RT,RA,RB

31	RT	RA	RB	95	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 56 0 || MEM(EA,1)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

For ***lbepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPLC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Special Registers Altered:

None

Programming Note

This instruction behaves identically to a ***lbzx*** instruction except for using the EPLC register to provide the translation context.

Load Halfword by External Process ID Indexed *X-form*

lhepx RT,RA,RB

31	RT	RA	RB	287	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48 0 || MEM(EA,2)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

For ***lhepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPLC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Special Registers Altered:

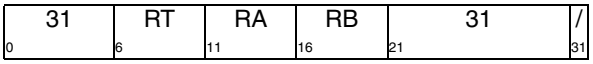
None

Programming Note

This instruction behaves identically to a ***lhzx*** instruction except for using the EPLC register to provide the translation context.

Load Word by External Process ID Indexed *X-form*

lwepx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA,4)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

For ***lwepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPLC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Special Registers Altered:

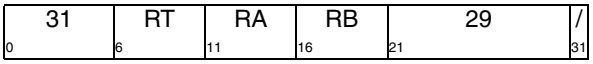
None

Programming Note

This instruction behaves identically to a ***lwzx*** instruction except for using the EPLC register to provide the translation context.

Load Doubleword by External Process ID Indexed *X-form*

ldepx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← MEM(EA,8)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

For ***ldepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPLC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Corequisite Categories:

64-Bit

Special Registers Altered:

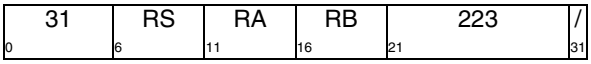
None

Programming Note

This instruction behaves identically to a ***ldx*** instruction except for using the EPLC register to provide the translation context.

Store Byte by External Process ID Indexed *X-form*

stbepx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA,1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+(RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

For ***stbepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPSC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Special Registers Altered:

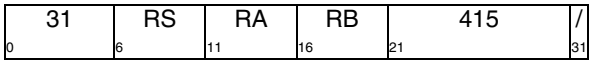
None

Programming Note

This instruction behaves identically to a ***stbx*** instruction except for using the EPSC register to provide the translation context.

Store Halfword by External Process ID Indexed *X-form*

sthepx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA,2) ← (RS)48:63
```

Let the effective address (EA) be the sum (RA|0)+(RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

For ***sthepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPSC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Special Registers Altered:

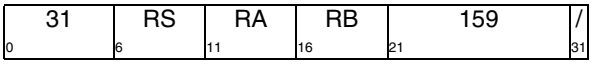
None

Programming Note

This instruction behaves identically to a ***sthx*** instruction except for using the EPSC register to provide the translation context.

Store Word by External Process ID Indexed *X-form*

stwepx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA,4) ← (RS)32:63
```

Let the effective address (EA) be the sum (RAl0)+(RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

For ***stwepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPSC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Special Registers Altered:

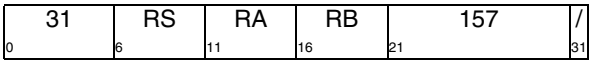
None

Programming Note

This instruction behaves identically to a ***stwx*** instruction except for using the EPSC register to provide the translation context.

Store Doubleword by External Process ID Indexed *X-form*

stdepx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA,8) ← (RS)
```

Let the effective address (EA) be the sum (RAl0)+(RB). (RS) is stored into the doubleword in storage addressed by EA.

For ***stdepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in pace of MSR_{GS} <E.HV>
$EPSC_{ELPID}$ is used in pace of LPIDR <E.HV>

This instruction is privileged.

Corequisite Categories:

64-Bit

Special Registers Altered:

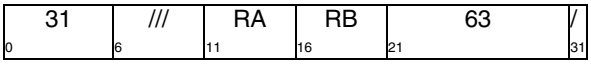
None

Programming Note

This instruction behaves identically to a ***stdx*** instruction except for using the EPSC register to provide the translation context.

Data Cache Block Store by External PID *X-form*

dcbstep RA,RB



Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required, a block containing the byte addressed by EA is in the data cache of any thread, and any locations in the block are considered to be modified there, then those locations are written to main storage. Additional locations in the block may be written to main storage. The block ceases to be considered modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this thread, and any locations in the block are considered to be modified there, those locations are written to main storage. Additional locations in the block may be written to main storage, and the block ceases to be considered modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

The instruction is treated as a *Load* with respect to translation, memory protection, and is treated as a *Write* with respect to debug events.

This instruction is privileged.

For ***dcbstep***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

Special Registers Altered:

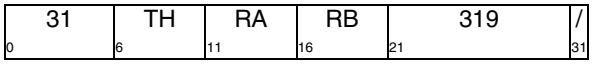
None

Programming Note

This instruction behaves identically to a ***dcbst*** instruction except for using the EPLC register to provide the translation context.

Data Cache Block Touch by External PID *X-form*

dcbtep TH,RA,RB



Let the effective address (EA) be the sum (RA|0)+(RB).

The dcbtep instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA. If the Cache Specification category is supported, the nature of the hint is affected by TH values of 0b00000 to 0b00111. Values associated with the Stream category are ignored. See Section 4.3.2 of Book II for more information.

If the block is in a storage location that is Caching Inhibited or Guarded, the hint is ignored.

The only operation that is "caused" by the ***dcbtep*** instruction is the providing of the hint. The actions (if any) taken in response to the hint are not considered to be "caused by" or "associated with" the ***dcbtep*** instruction (e.g., ***dcbtep*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

The ***dcbtep*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified in the ***dcbt*** instruction in Section 4.3.2 of Book II.

The instruction is treated as a *Load*, except that no interrupt occurs if a protection violation occurs.

The instruction is privileged.

The normal address translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch by External PID* instruction so that it can

be coded with the TH value as the last operand for all categories. .

Extended:	Equivalent to:
dcbtctep RA,RB,TH	dcbtep for TH values of 0b0000 - 0b0111; other TH values are invalid.
dcbtdsep RA,RB,TH	dcbtep for TH values of 0b0000 or 0b1000 - 0b1010; other TH values are invalid.

Programming Note

This instruction behaves identically to a ***dcbt*** instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Data Cache Block Flush by External PID X-form

dcbfep RA,RB,L

31	///	L	RA	RB	127	/
0	6	9	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

L=0

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

L=1 (“dcbf local”) [Category: Embedded.Phased-In]

The L=1 form of the ***dcbfep*** instruction permits a program to limit the scope of the “flush” operation to the data cache of this processor. If the block containing the byte addressed by EA is in the data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 3 (“dcbf local primary”) [Category: Embedded.Phased-In]

The L=3 form of the ***dcbfep*** instruction permits a program to limit the scope of the “flush” operation to the primary data cache of this processor. If the block containing the byte addressed by EA is in the primary data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

For the L operand, the value 2 is reserved. The results of executing a ***dcbfep*** instruction with L=2 are boundedly undefined.

Engineering Note

dcbfep with an L value of 2 should be treated as if the value were 0.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

The instruction is treated as a *Load* with respect to translation, memory protection, and is treated as a *Write* with respect to debug events.

This instruction is privileged.

The normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR_{GS}<E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR<E.HV>

Special Registers Altered:

None

Programming Note

This instruction behaves identically to a ***dcbf*** instruction except for using the EPLC register to provide the translation context.

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Flush by External PID* instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Section B.2 of Book III-E. The extended mnemonics are shown below.

Extended:	Equivalent to:
dcbfep RA,RB	dcbfep RA,RB,0
dcbflep RA,RB	dcbfep RA,RB,1
dcbflpep RA,RB	dcbfep RA,RB,3

Except in the ***dcbfep*** instruction description in this section, references to "***dcbfep***" in Books I-III imply L=0 unless otherwise stated or obvious from context; "***dcbflep***" is used for L=1 and "***dcbflpep***" is used for L=3.

Programming Note

dcbfep serves as both a basic and an extended mnemonic. The Assembler will recognize a ***dcbfep*** mnemonic with three operands as the basic form, and a ***dcbfep*** mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

dcbfep with L=1 can be used to provide a hint that a block in this processor's data cache will not be reused soon.

dcbfep with L=3 can be used to flush a block from the processor's primary data cache but reduce the latency of a subsequent access. For example, the block may be evicted from the primary data cache but a copy retained in a lower level of the cache hierarchy.

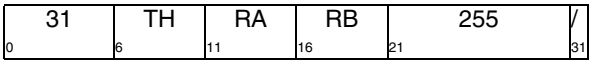
Programs which manage coherence in software must use ***dcbfep*** with L=0.

Engineering Note

dcbfep with L=1 requires caches that are private to this processor to be flushed. Caches shared with other processors need not be affected provided that the requirements of memory coherence are satisfied.

Data Cache Block Touch for Store by External PID *X-form*

dcbtstep TH,RA,RB



Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbtstep*** instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA. If the Cache Specification category is supported, the nature of the hint is affected by TH values of 0b00000 to 0b00111. Values associated with the Stream category are ignored. See Section 4.3.2 of Book II for more information.

If the block is in a storage location that is Caching Inhibited or Guarded, the hint is ignored.

The only operation that is "caused" by the ***dcbtstep*** instruction is the providing of the hint. The actions (if any) taken in response to the hint are not considered to be "caused by" or "associated with" the ***dcbtstep*** instruction (e.g., ***dcbtstep*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

The ***dcbtstep*** instruction may complete before the operation it causes has been performed.

The instruction is treated as a *Store*, except that no interrupt occurs if a protection violation occurs.

The instruction is privileged.

The normal address translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch for Store by External PID* instruction so that it can be coded with the TH value as the last operand for all categories. .

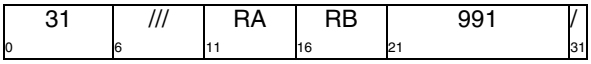
Extended:	**Equivalent to:**
dcbtstctep RA,RB,TH	dcbtstep for TH values of 0b0000 - 0b0111; other TH values are invalid.

Programming Note

This instruction behaves identically to a ***dcbtst*** instruction except for using the EPLC register to provide the translation context, and not supporting the Stream category.

Instruction Cache Block Invalidate by External PID *X-form*

icbiep RA,RB



Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any thread, the block is invalidated in those instruction caches.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of this thread, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

The instruction is treated as a *Load*.

This instruction is privileged.

For ***icbiep***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

Special Registers Altered:

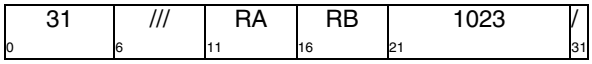
None

Programming Note

This instruction behaves identically to an ***icbi*** instruction except for using the EPLC register to provide the translation context.

Data Cache Block set to Zero by External PID *X-form*

dcbzep RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← block size (bytes)
m ← log2(n)
ea ← EA0:63-m || m0
MEM(ea, n) ← n0x00
```

Let the effective address (EA) be the sum (RA|0)+(RB).

All bytes in the block containing the byte addressed by EA are set to zero.

This instruction is treated as a *Store*.

This instruction is privileged.

The normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPSC_{ELPID}$ is used in place of LPIDR <E.HV>

Special Registers Altered:

None

Programming Note

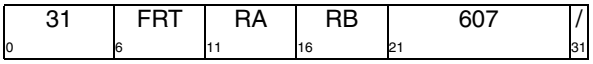
See the Programming Notes for the ***dcbz*** instruction.

Programming Note

This instruction behaves identically to a ***dcbz*** instruction except for using the EPSC register to provide the translation context.

Load Floating-Point Double by External Process ID Indexed *X-form*

lfdepx FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← MEM(EA,8)
```

Let the effective address (EA) be the sum (RAl0)+(RB). The doubleword in storage addressed by EA is loaded into FRT.

For ***lfdepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***lfdepx*** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Corequisite Categories:
Floating-Point

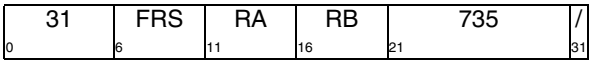
Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***lfdx*** instruction except for using the EPLC register to provide the translation context.

Store Floating-Point Double by External Process ID Indexed *X-form*

stfdepx FRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA,8) ← (FRS)
```

Let the effective address (EA) be the sum (RAl0)+(RB). (FRS) is stored into the doubleword in storage addressed by EA.

For ***stfdepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPSC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***stfdepx*** while MSR_{FP}=0 will cause a Floating-Point Unavailable interrupt.

Corequisite Categories:
Floating-Point

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***stfdx*** instruction except for using the EPSC register to provide the translation context.

Vector Load Doubleword into Doubleword by External Process ID Indexed EVX-form

evlddepx RT,RA,RB

31	RT	RA	RB	799	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← MEM(EA,8)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

For ***evlddepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***evlddepx*** while MSR_{SPV}=0 will cause an SPE Unavailable interrupt.

Corequisite Categories:
Signal Processing Engine

Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***evlddx*** instruction except for using the EPLC register to provide the translation context.

Vector Store Doubleword into Doubleword by External Process ID Indexed EVX-form

evstddepx RS,RA,RB

31	RT	RA	RB	927	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA,8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+(RB). (RS) is stored into the doubleword in storage addressed by EA.

For ***evstddepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPSC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***evstddepx*** while MSR_{SPV}=0 will cause an SPE Unavailable interrupt.

Corequisite Categories:
Signal Processing Engine

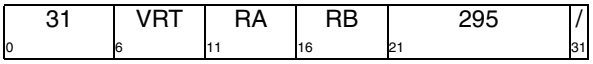
Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***evstddx*** instruction except for using the EPSC register to provide the translation context.

Load Vector by External Process ID Indexed *X-form*

lvepx VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

For ***lvepx***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***lvepx*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

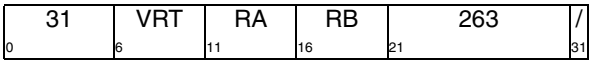
Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***lvx*** instruction except for using the EPLC register to provide the translation context.

Load Vector by External Process ID Indexed LRU *X-form*

lvepxl VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon
( EA )
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

lvepxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

For ***lvepxl***, the normal translation mechanism is not used. The contents of the EPLC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPLC_{EPR}$ is used in place of MSR_{PR}
$EPLC_{EAS}$ is used in place of MSR_{DS}
$EPLC_{EPID}$ is used in place of PID
$EPLC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPLC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***lvepxl*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

Programming Note

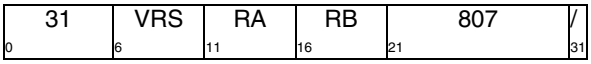
See the Programming Notes for the ***lvxl*** instruction in Section 5.7.2 of Book I.

Programming Note

This instruction behaves identically to a ***lvxl*** instruction except for using the EPLC register to provide the translation context.

Store Vector by External Process ID Indexed *X-form*

stvepx VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

For ***stvepx***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPSC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***stvepx*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

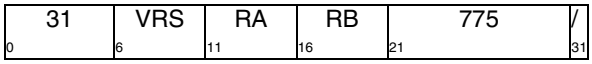
Special Registers Altered:
None

Programming Note

This instruction behaves identically to a ***stvx*** instruction except for using the EPSC register to provide the translation context.

Store Vector by External Process ID Indexed LRU *X-form*

stvepxl VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
mark_as_not_likely_to_be_needed_again_anytime_soon
(EA)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

The ***stvepxl*** instruction provides a hint that the quadword addressed by EA will probably not be needed again by the program in the near future.

For ***stvepxl***, the normal translation mechanism is not used. The contents of the EPSC register are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

$EPSC_{EPR}$ is used in place of MSR_{PR}
$EPSC_{EAS}$ is used in place of MSR_{DS}
$EPSC_{EPID}$ is used in place of PID
$EPSC_{EGS}$ is used in place of MSR[GS] <E.HV>
$EPSC_{ELPID}$ is used in place of LPIDR <E.HV>

This instruction is privileged.

An attempt to execute ***stvepxl*** while MSR_{SPV}=0 will cause a Vector Unavailable interrupt.

Corequisite Categories:
Vector

Special Registers Altered:
None

Programming Note

See the Programming Notes for the ***lvxl*** instruction in Section 5.7.2 of Book I.

Programming Note

This instruction behaves identically to a ***stvxl*** instruction except for using the EPSC register to provide the translation context.

Chapter 6. Storage Control

6.1 Overview

Instruction effective addresses are generated for sequential instruction fetches and for addresses that correspond to a change in program flow (branches, interrupts). Data effective addresses are generated by *Load*, *Store*, and *Cache Management* instructions. *TLB Management* instructions generate effective addresses to determine the presence of or to invalidate a specific TLB entry associated with that address. For a complete discussion of storage addressing and effective address calculation, see Section 1.10 of Book I.

Portions of the context of an effective address are appended to it to form the virtual address. The context is provided by various registers. The virtual address consists of the Logical Partition ID (LPID) <E.HV>, the Guest State <E.HV>, the address space identifier, the process identifier, and the effective address. The virtual address is translated to a real address by a matching "direct" entry in the Translation Lookaside Buffer (TLB) according to procedures described in Section 6.7.3. The Virtual Page Number (VPN) part of the virtual address is compared to the TLB contents to determine a match. The VPN consists of bits of the virtual address with the exception of the low-order effective address bits that correspond to the byte offset within the page. If the Embedded.Page Table category is supported, a virtual address can be translated by the Page Table pointed to by a matching "indirect" TLB entry as described in Section 6.7.4. As a result of a Page Table translation, a direct TLB entry is created, and this direct TLB entry can be used for subsequent translations. All virtual addresses are translated by the Page Table <E.PT> or the TLB, i.e. unlike the Server environment, there is no real mode. The real address that results from the translation is used to access main storage.

The Translation Lookaside Buffer is the hardware resource that also controls protection and storage control attributes. TLB permission bits control user and supervisor read, write and execute capability. If the Embedded.Hypervisor category is supported, the Virtualization Fault bit permits data accesses to pages to be trapped to the hypervisor, which allows the hypervisor to virtualize data accesses to specific pages, e.g. accesses to memory-mapped I/O. Storage control attributes described in Book II are supported by corresponding TLB bits, as are four optional implementation-dependent user-defined storage control attributes. The organization of the TLB (e.g. associativity, number of entries, number of arrays, etc.) is implementation-dependent. MMU configuration and TLB configuration information in various registers describes this implementation-dependent organization.

Software manages translation directly by installing TLB entries, and indirectly by setting up the page tables, which the TLB will cache. *TLB Management* instructions are used by software to read, write, search and invalidate TLB contents. MMU Assist Registers (MAS) are used to transfer data to and from the TLB arrays by *TLB Management* instructions. If the Embedded.Hypervisor category is not supported, *TLB Management* instructions are privileged instructions.

A different MMU Architecture Version (MAV) is used to indicate that different register layouts and functions are provided. The MMU Architecture Version Number is specified by the read-only MMUCFG register. The Embedded.Hypervisor.LRAT, Embedded TLB Write Conditional, and Embedded.Page Table categories are available only in MMU Architecture Version 2.0.

If the Embedded.Hypervisor category is supported and the Embedded.Hypervisor.LRAT category is not supported, most *TLB Management* instructions are hypervisor instructions. TLB entries contain real addresses, and, to maintain isolation between partitions, guest operating systems are not given access to real addresses. In this case most *TLB Management* instructions trap to the hypervisor. The hypervisor can emulate a *TLB Management* instruction by swapping a Real Page Number for corresponding Logical Page Number (LPN) in the MAS registers or vice versa so that the guest OS only sees LPNs.

However, if the Embedded.Hypervisor.LRAT category is supported, hardware can perform the translation of an LPN into a corresponding RPN. In this case, a *TLB Write Entry* (***tlbwe***) instruction can be executed in guest supervisor state. The LPN in the MAS register is translated into a corresponding RPN by a hardware lookup in a Logical to Real Address Translation (LRAT)

array, and the RPN is written to the TLB in place of the LPN when ***tlbwe*** is executed in guest supervisor state.

The Embedded.TLB Write Conditional (E.TWC) category provides a TLB write operation that is conditional on a TLB-reservation where the TLB-reservation is previously established by a ***tlbsrx.*** instruction. The TLB-reservation is cleared by TLB invalidations and TLB writes involving the same virtual page. Thus, without acquiring a software lock, software can use the E.TWC category to write a TLB entry while ensuring that the entry is not a duplicate of an entry created simultaneously by another thread that shares the TLB and is not a stale value for a virtual page that was concurrently invalidated.

Figure 19 gives an overview of address translation if the Embedded.Page Table category is supported. The IND bit in a TLB entry indicates whether the entry is a "direct" entry or "indirect" entry. When a virtual address is translated, the TLB arrays are searched for a matching entry. If there is one and only one matching direct entry, that entry is used to translate the VA. If there is no matching direct TLB entry, but there is one and only one matching indirect entry, the indirect entry is used to access a Page Table Entry (PTE). If the PTE is a valid entry (V bit = 1), the PTE is used to translate the address and a "direct" entry is written to the TLB. If the Embedded.Page Table and Embedded.Hypervisor categories are both supported, the Embedded.Hypervisor.LRAT category is supported. In this case if the TGS bit of the indirect TLB entry is 1, the RPN from the PTE is treated as a Logical Page Number (LPN) and translated by the LRAT into an RPN. If the Embedded.Page Table is supported but the Embedded.Hypervisor category is not supported, supervisor software can create direct and indirect TLB entries and can control the Page Table Entries. If both categories are supported, guest supervisor software can still create direct and indirect TLB entries and control the Page Table Entries if guest execution of *TLB Management* instructions is enabled. However, depending on various factors such as the number of available LRAT entries, performance may be better if guest virtual addresses are translated by a Page Table that is managed by hypervisor software.

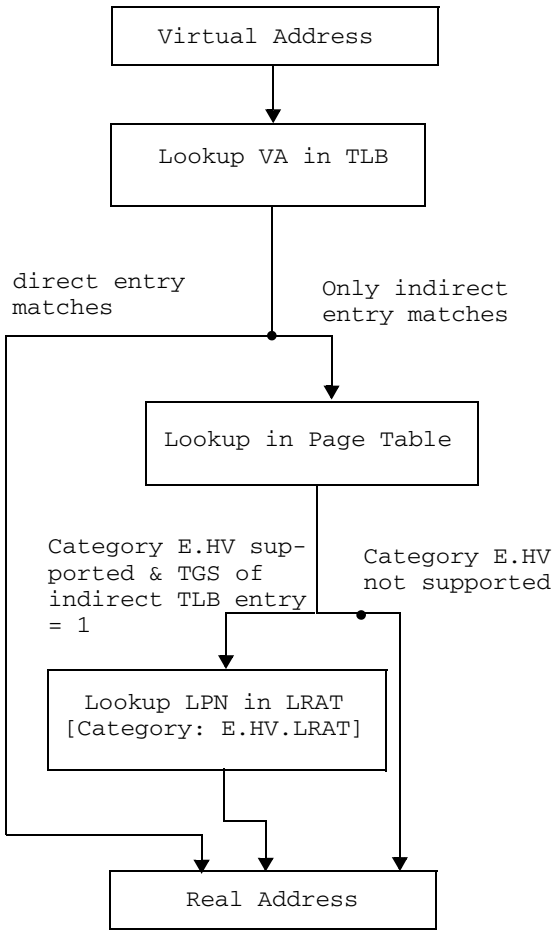


Figure 19. Address translation with page table

Address Size Overview

- Real address space size is 2^m bytes, m≤64; see Note 1.
- In MMU Architecture Version 1.0, real page sizes are 4^p KB where 0≤p≤15 (i.e. 1 KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB, 4GB, 16GB, 64GB, 256GB, 1TB); see Note 2. In MMU Architecture Version 2.0, real page sizes are 2^p KB where 0≤p≤31 (i.e. 1 KB, 2 KB, 4KB, 8 KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB, 8GB, 16GB, 32GB, 64GB, 128GB, 256GB, 512GB, 1TB, 2TB); see Note 2. However, real pages sizes supported by a Page Table are limited to values of p where 2≤p≤15.
- Effective address space size is 2^{64} bytes in 64-bit implementations and 2^{32} bytes in 32-bit implementations.
- The virtual address space size depends on the implementation.

  - Virtual address space size in 64-bit implementations is 2^v bytes, where:
    - $66 \le v \le 79$ if the Embedded.Hypervisor Category is not supported; see Note 3.
    - $68 \le v \le 92$ if the Embedded.Hypervisor Category is supported; see Note 3.
  - Virtual address space size in 32-bit implementations is 2^v bytes, where:
    - $34 \le v \le 47$ if the Embedded.Hypervisor Category is not supported; see Note 3.
    - $36 \le v \le 60$ if the Embedded.Hypervisor Category is supported; see Note 3.
  - The number of LPID <E.HV> bits is $1 \le g \le 12$; see Note 3.
  - There is one GS <E.HV> bit.
  - There is one AS bit.
  - The number of PID bits is $1 \le d \le 14$; see Note 3.
  - For any given real page, the virtual page size is the same as the real page size.
- If the Embedded.Hypervisor.LRAT category is supported, the following applies.
  - The logical page sizes allowed by the architecture are the same as the real page sizes. However, an implementation need not support the same logical and real page sizes.
  - The logical address space size is 2^q bytes, where $q \le 64$; see Note 4.

Notes:

1. The value of m is implementation-dependent (subject to the maximum given above). When used to address storage, the high-order 64-m bits of the "64-bit" real address must be zeros. A maximum of 64 bits of real address can by supported by the TLB. A maximum of 52 bits of real address can be supported by the Page Table <E.PT>.
2. Which of these pages sizes are supported is implementation-dependent. If an implementation supports multiple TLB arrays, the page sizes supported by each array may be different. Supported page sizes are indicated by TLB configuration information (see Sections 6.10.3.3 and 6.10.3.4).
3. The values of v, g, and d are implementation-dependent (subject to the range given above). The value of v is a function of g, d, whether the implementation is 32-bit or 64-bit, and whether the Embedded.Hypervisor category is supported.
4. The value of q is implementation-dependent (subject to the maximum given above). A maximum of 64 bits of logical address can by supported by the LRAT. A maximum of 52 bits of logical address can be supported by the Page Table <E.PT>.

Programming Note

[Category: Embedded.Hypervisor.LRAT]: The logical pages sizes supported by an implementation are typically larger than the real page sizes supported. This implies that memory blocks must be assigned to a partition with larger granularity than the memory blocks that can be managed within a partition.

6.2 Storage Exceptions

A *storage exception* results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Compare Debug Interrupt).

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a Load or Store instruction. See Section 2.1 of Book II and Section 7.7 on page 1186 in this Book.

6.3 Instruction Fetch

For an instruction fetch, MSR_{IS} is appended to the effective address as part of the virtual address. The Address Translation mechanism is described in Section 6.7.2, Section 6.7.3, and, if the Embedded.Page Table category is supported, Section 6.7.4.

6.3.1 Implicit Branch

Explicitly altering certain MSR bits (using ***mtmsr***), or explicitly altering TLB entries, certain System Registers and possibly other implementation-dependent registers, may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an *implicit* branch. For example, an ***mtmsr*** instruction that changes the value of MSR_{CM} may change the real address from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 12. "Synchronization Requirements for Context Alterations" on page 1235. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

6.3.2 Address Wrapping Combined with Changing MSR Bit CM

If the current instruction is at effective address 2^{32}-4 and is an ***mtmsr*** instruction that changes the contents of MSR_{CM}, the effective address of the next sequential instruction is undefined.

Programming Note

In the case described in the preceding paragraph, if an interrupt occurs before the next sequential instruction is executed, the contents of SRR0, CSRR0, or MCSRR0, as appropriate to the interrupt, are undefined if the Embedded.Hypervisor category is not supported or the interrupt is directed to the hypervisor state. If the Embedded.Hypervisor category is supported and the interrupt is directed to the guest state, the contents of GSRR0 are undefined.

6.4 Data Access

For a normal *Load* or *Store* instruction, MSR_{DS} is appended to the effective address as part of the virtual address. The Address Translation mechanism is described in Section 6.7.2, Section 6.7.3, and, if the Embedded.Page Table category is supported, Section 6.7.4. The Embedded.External PID category must be supported. The effective address for an External Process ID Load or Store instruction data access is processed under control of the EPLC or EPSC, respectively. See Section 5.3.7.1 and Section 5.3.7.2.

6.5 Performing Operations Out-of-Order

An operation is said to be performed “in-order” if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed “out-of-order” if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as *Branch*, *Trap*, *System Call*, and *Return From Interrupt* instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, operations are performed out-of-order when resources are available that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, any results of the operation are abandoned (except as described below).

In the remainder of this section, including its subsections, “*Load* instruction” includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be “treated as a *Load*”, and similarly for “*Store* instruction”.

A data access that is performed out-of-order may correspond to an arbitrary *Load* or *Store* instruction (e.g., a *Load* or *Store* instruction that is not in the instruction stream being executed). Similarly, an instruction fetch that is performed out-of-order may be for an arbitrary instruction (e.g., the aligned word at an arbitrary location in instruction storage).

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

- Stores

  Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order).

- Accessing Guarded Storage

  The restrictions for this case are given in Section 6.8.1.1.

The only permitted side effects of performing an operation out-of-order are the following.

- A Machine Check that could be caused by in-order execution may occur out-of-order except that, if category E.HV is supported and the Machine Check is the result of multiple TLB entries that translate the same VA, the Machine Check interrupt must occur in the context in which it was caused. Also, if category E.HV is supported, a Machine Check interrupt resulting from the following situations must be precise.
  - Execution of an *External Process ID* instruction that has an operand that can be translated by multiple TLB entries.
  - Execution of a ***tlbivax*** instruction that isn’t a TLB invalidate all and there are multiple entries in a single thread’s TLB array(s) that match the complete VPN.
  - Execution of a ***tlbilx*** instruction with T=3 and there are multiple entries in the TLB array(s) that match the complete VPN.
  - Execution of a ***tlbsx*** or ***tlbsrx.*** instruction and there are multiple matching TLB entries.
- Non-Guarded storage locations that could be fetched into a cache by in-order fetching or execution of an arbitrary instruction may be fetched out-of-order into that cache.

6.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 6.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist. See Section 7.6.3 on page 1165.

6.7 Storage Control

This section describes the address translation facility, access control, and storage control attributes.

Demand-paged virtual memory is supported, as well as a variety of other management schemes that depend on precise control of effective-to-real address translation and flexible memory protection. Translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

The effective address space is divided into pages. The page represents the granularity of effective address translation, access control, and storage control attributes. In MMU Architecture Version 1.0, up to sixteen page sizes (1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB, 4GB, 16GB, 64GB, 256GB, 1TB) may be simultaneously supported. In MMU Architecture Version 2.0, up to 32 page sizes (1 KB, 2 KB, 4KB, 8 KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB, 8GB, 16GB, 32GB, 64GB, 128GB, 256GB, 512GB, 1TB, 2TB) may be simultaneously supported. In order for an effective to real translation to exist, a valid entry for the page containing the effective address must be in the Translation Lookaside Buffer (TLB). Addresses for which no TLB entry exists cause TLB Miss exceptions.

6.7.1 Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) is the hardware resource that controls translation, protection, and storage control attributes. The organization of the TLB (e.g. associativity, number of entries, number of arrays, etc.) is implementation-dependent. Thus, the software for updating the TLB is also implementation-dependent. However, MMU configuration and TLB configuration information is provided such that software written to handle various TLB organizations could potentially run on multiple MMU implementations. A unified TLB organization (one to four TLB arrays, called TLB0, TLB1, TLB2 and TLB3, where each contains translations for both instructions and data) is assumed in the following description. For details on how to synchronize TLB updates with instruction execution see Section 6.11.4.3 and Chapter 12.

Maintenance of TLB entries is under software control, except that if the Embedded.Page Table category is supported, hardware will write TLB entries for translations performed via the Page Table. System software determines TLB entry replacement strategy and the format and use of any page state information. If a TLB provides Next Victim (NV) information, software can optionally use NV to choose a TLB entry to be replaced. See Section 6.11.4.7. Some implementations allow software to specify that a hardware generated hash and hardware replacement algorithm should be used to select the entry. See Section 6.11.4.7. The TLB entry contains all the information required to identify the page, to specify the translation, to specify access controls, and to specify the storage control attributes.

A TLB entry is written by copying information from MAS registers, using a ***tlbwe*** instruction (see page 1141). A TLB entry is read by copying information to MAS registers, using a ***tlbre*** instruction (see page 1139). Software can also search for specific TLB entries using the ***tlbsx*** instruction (see page 1136) and, if the Embedded.TLB Write Conditional category is supported, ***tlbsrx.*** (see page 1138).

Each TLB entry describes a page. Fields in the TLB entry fall into five categories:

- Page identification fields (information required to identify the page to the hardware translation mechanism).
- Address translation fields
- Access control fields
- Storage control attribute fields
- TLB management field

While the fields in the TLB entry are required, unless they are identified as part of a category that is not supported, no particular TLB entry format is formally specified. The ***tlbre*** and ***tlbwe*** instructions provide the ability to read or write individual entries. Below are shown the field definitions for the TLB entry. Some fields that are used only for indirect TLB entries can be overlaid with fields that are used only for direct TLB entries. Such overlap is implementation-dependent and an example is shown in Figure 20 on page 1081.

.

Page Identification Fields for direct and indirect entries

Name **Description**

EPN **Effective Page Number** (up to 54 bits)

Bits 0:*n*–1 of the EPN field are compared to bits 0:*n*–1 of the effective address (EA) of the storage access (where *n*=64–$\log_2$(*page size* in bytes) and *page size* is specified by the SIZE field of the TLB entry). See Table 2 and Table 3.

Note: Bits X:Y of the EPN field are implemented, where X=0 (64-bit implementation) or X=32 (32-bit implementation), and Y ≤ 53. Y = p - 1 where p = 64–$\log_2$(*smallest page size* in bytes) and *smallest page size* is the smallest page size supported by the implementation as specified by TLB array's TLBnCFG or TLBnPS. The number of bits implemented for EPN is not required to be the same number of bits as are implemented for RPN. Implemented bits that represent offsets within a page are ignored for address comparisons performed for translation, invalidation, and searches and software should set these bits to zero. Unimplemented EPN bits are treated as if they contain 0s.

TS **Translation Address Space**

This bit indicates the address space this TLB entry is associated with. For instruction storage accesses, MSR_{IS} must match the value of TS in the TLB entry for that TLB entry to provide the translation. Likewise, for data storage accesses, MSR_{DS} must match the value of TS in the TLB entry. For the ***tlbsrx.*** <E.TWC> instruction, $MAS1_{TS}$ provides the address space specification that must match the value of TS. For the ***tlbsx*** instruction, $MAS6_{SAS}$ provides the address space specification that must match the value of TS. For the instructions ***tlbilx*** with T=3 and ***tlbivax*** with EA_{61}=0, $MAS6_{SAS}$ provides the address space specification that is compared to the value of TS.

SIZE **Page Size**

For direct TLB entries, the SIZE field specifies the size of the virtual page associated with the TLB entry. For indirect TLB entries, the SIZE field specifies the maximum amount of virtual storage that can be mapped by the page table to which the indirect TLB entry points. The following applies in both cases:

- For MAV = 1.0, 4^{SIZE}KB, where 0 ≤ SIZE ≤ 15. See Table 2. For TLB arrays that contain fixed-size TLB entries, this field is treated as reserved for ***tlbwe*** and ***tlbre*** instructions and is treated as a fixed value for translations. For variable page size TLB arrays, this field must be a value between TLB*n*$CFG_{MINSIZE}$ and TLB*n*$CFG_{MAXSIZE}$.
- For MAV = 2.0, 2^{SIZE}KB, where 0 ≤ SIZE ≤ 31. See Table 3. This field must be one of the page sizes specified by the corresponding TLB*n*PS register.

Implementations may support any one or more of the page sizes described above.

TID **Translation ID** (implementation-dependent size)

Field used to identify a shared page (TID=0) or the owner's process ID of a private page (TID≠0). See Section 6.7.2.

TLPID **Translation Logical Partition ID** <E.HV>

This field identifies a partition. The Translation Logical Partition ID is compared with LPIDR contents during translation. This allows for an efficient change of address space when a transition between partitions occurs. This number of bits in this field is an implementation-dependent number n, where 1≤ n ≤12. See Section 6.7.2.

TGS **Translation Guest State** <E.HV>

This 1-bit field indicates whether this TLB entry is valid for the guest space or for the hypervisor space. The Translation Guest Space field is compared with the MSR_{GS} bit during translation. This allows for an efficient change of address space when a transition from guest state to hypervisor state occurs. See Section 6.7.2.

0 Hypervisor space

1 Guest space

V **Valid**

This bit indicates whether that this TLB entry is valid and may be used for translation. The Valid bit for a given entry can be set or cleared with a ***tlbwe*** instruction; alternatively, the Valid bit for an entry may be cleared by a ***tlbilx*** or ***tlbivax*** instruction or by a MMUCSR0 TLB invalidate all.

IND **Indirect** <E.PT>

This bit distinguishes between an indirect TLB entry that points to a Page Table (IND=1) and a direct TLB entry that can be used directly to translate a virtual address (IND=0). If a TLB array does not support this bit ($TLBnCFG_{IND}$ = 0), the implied IND value is 0. For the ***tlbsx*** instruction, $MAS6_{SIND}$ provides the direct/indirect specification that must match the value of IND. For the instructions ***tlbilx*** with T=3 and ***tlbivax*** with EA_{61}=0, $MAS6_{SIND}$ provides the direct/indirect specification that is compared to the value of IND. See Section 6.7.4.

Page Identification Field for indirect entry

Name Description

SPSIZE Sub-Page Size (IND=1) <E.PT>

SPSIZE is a 5-bit field that specifies the minimum page size that can be specified by each Page Table Entry in the Page Table that is pointed to by the indirect TLB entry. This minimum page size is 2^{SPSIZE} KB and must be at least 4 KB. Thus SPSIZE must be at least 2. Valid values are specified by $EPTCFG_{SPS2\ SPS1\ SPS0}$. See Section 6.7.4

Translation Field

Name Description

RPN **Real Page Number** (up to 54 bits)

For a direct TLB entry, bits 0:*n*–1 of the RPN field are used to replace bits 0:*n*–1 of the effective address to produce the real address for the storage access (where *n*=64–$\log_2$(*page size* in bytes) and *page size* is specified by the SIZE field of the TLB entry). See Section 6.7.3 for a requirement on unused low-order RPN bits (i.e. bits *n*:53) being 0.

For an indirect TLB entry, bits 0:m-1 of the RPN field followed by 64-m 0s are the real address of the page table pointed to the indirect TLB entry, where m = 61 – (SIZE – SPSIZE). RPN bits m:53 must be zero. See Section 6.7.4.

Note: Bits X:Y of the RPN field are implemented, where $X \geq 0$ and $Y \leq 53$. X = 64 - $MMUCFG_{RASIZE}$. Y is the larger of the following applicable values:

- p - 1 where p = 64–$\log_2$(*smallest_page size* in bytes) and *smallest page size* is the smallest page size supported by the implementation as specified by TLB array's TLBnCFG or TLBnPS.
- 52 if the Embedded.Page Table category is supported and a page table size of 2 KB is supported ($EPTCFG_{PSn}$ - $EPTCFG_{SPSn}$ = 8 for some value of n).

The number of bits implemented for EPN is not required to be the same number of bits as are implemented for RPN. Unimplemented RPN bits are treated as if they contain 0s.

Storage Control Bits (see Section 6.8.3 on page 1097)

Name Description

W **Write-Through Required**

This bit indicates whether the page is Write-Through Required. See Section 1.6.1 of Book II.

0 This page is not Write-Through Required storage.

1 This page is Write-Through Required storage.

I **Caching Inhibited**

This bit indicates whether the page is Caching Inhibited. See Section 1.6.2 of Book II.

0 This page is not Caching Inhibited storage.

1 This page is Caching Inhibited storage.

M **Memory Coherence Required**

This bit indicates whether the page is Memory Coherence Required. See Section 1.6.3 of Book II.

0 This page is not Memory Coherence Required storage.

1 This page is Memory Coherence Required storage.

G **Guarded**

This bit indicates whether the page is Guarded. See Section 1.6.4 of Book II and Section 6.8.1.

0 This page is not Guarded storage.

1 This page is Guarded storage.

E **Endian Mode**

This bit indicates whether the page is accessed in Little-Endian or Big-Endian byte order. See Section 1.10.1 of Book I and Section 1.6.5 of Book II.

0 The page is accessed in Big-Endian byte order.

1 The page is accessed in Little-Endian byte order.

U0:U3 **User-Definable Storage Control Attributes** See Section 6.8.2.

Specifies implementation-dependent and system-dependent storage control attributes for the page associated with the TLB entry. The existence of these bits is implementation-dependent.

VLE **Variable Length Encoding** <E.VLE>

This bit specifies whether a page which contains instructions is to be decoded as VLE instructions (see Chapter 1 of Book VLE). See Section 6.8.3 and Chapter 1 of Book VLE.

0 Instructions fetched from the page are decoded and executed as non-VLE instructions.

1 Instructions fetched from the page are decoded and executed as VLE instructions.

ACM **Alternate Coherency Mode**

This bit allows an implementation to employ more than a single coherency method. This allows participation in multiple coherency protocols. If the M attribute (Memory Coherence Required) is not set for a page (M=0), the page has no coherency associated with it and the ACM attribute is ignored. If the M attribute is set to 1 for a page (M=1), the ACM attribute is used to determine the coherence domain (or protocol) used. The coherency method used in Alternate Coherency Mode is implementation-dependent.

Access Control Fields for direct TLB entry

Name Description

UX **User State Execute Enable** (IND=0)

See Section 6.7.6.1.

0 Instruction fetch and execution is not permitted from this page while MSR_{PR}=1 and will cause an Execute Access Control exception type Instruction Storage interrupt.

1 Instruction fetch and execution is permitted from this page while MSR_{PR}=1.

SX **Supervisor State Execute Enable** (IND=0)

See Section 6.7.6.1.

0 Instruction fetch and execution is not permitted from this page while MSR_{PR}=0 and will cause an Execute Access Control exception type Instruction Storage interrupt.

1 Instruction fetch and execution is permitted from this page while MSR_{PR}=0.

UW **User State Write Enable** (IND=0)

See Section 6.7.6.2.

0 *Store* operations, including ***dcba***, ***dcbz***, and ***dcbzep*** are not permitted to this page when MSR_{PR}=1 and will cause a Write Access Control exception. A Write Access Control exception will cause a Data Storage interrupt.

1 *Store* operations, including ***dcba, dcbz***, and ***dcbzep*** are permitted to this page when MSR_{PR}=1.

SW **Supervisor State Write Enable** (IND=0)

See Section 6.7.6.2.

0 *Store* operations, including ***dcba***, ***dcbi***, ***dcbz***, and ***dcbzep*** are not permitted to this page when MSR_{PR}=0. *Store* operations, including ***dcbi, dcbz,*** and ***dcbzep***, will cause a Write Access Control exception. A Write Access Control exception will cause a Data Storage interrupt.

1 *Store* operations, including ***dcba***, ***dcbi***, ***dcbz***, and ***dcbzep***, are permitted to this page when MSR_{PR}=0.

UR **User State Read Enable** (IND=0)

See Section 6.7.6.3.

0 *Load* operations (including load-class *Cache Management* instructions) are not permitted from this page when MSR_{PR}=1 and will cause a Read Access Control exception. A Read Access Control exception will cause a Data Storage interrupt.

1 *Load* operations (including load-class *Cache Management* instructions) are permitted from this page when MSR_{PR}=1.

SR **Supervisor State Read Enable** (IND=0)
See Section 6.7.6.3.
0 *Load* operations (including load-class *Cache Management* instructions) are not permitted from this page when MSR_{PR}=0 and will cause a Read Access Control exception. A Read Access Control exception will cause a Data Storage interrupt.
1 *Load* operations (including load-class *Cache Management* instructions) are permitted from this page when MSR_{PR}=0.

Access Control Field for direct and indirect entries

Name Description

VF **Virtualization Fault** <E.HV;E.PT>
See Section 6.7.6.4
This 1-bit field specifies whether the TLB entry is used by the hypervisor to virtualize data accesses, e.g. accesses to memory-mapped I/O. A translation of the operand address of a *Load*, *Store*, or *Cache Management* instruction that uses a TLB entry with the Virtualization Fault field equal to 1 causes a Virtualization Fault exception type Data Storage interrupt regardless of the settings of the permission bits. The interrupt is always directed to hypervisor state regardless of the setting of $EPCR_{DSIGS}$.
0 A *Load*, *Store*, or *Cache Management* access to this page does not cause a Virtualization Fault exception.
1 A *Load*, *Store*, or *Cache Management* access to this page causes a Virtualization Fault exception.

TLB Management Field

Name Description

IPROT **Invalidation Protection**
A TLB entry with this bit equal to 1 is protected from all TLB invalidation mechanisms except the explicit writing of a 0 to the V bit. See Section 6.11.4.3. IPROT is implemented only for TLB entries in TLB arrays where $TLBnCFG_{IPROT}$ is indicated. If IPROT = 1, the TLB entry is protected from invalidate operations due to any of the following.

- execution of ***tlbivax***
- execution of ***tlbilx***
- ***tlbivax*** invalidations from another thread
- ***tlbilx*** invalidations from another thread when the TLB is shared with that thread
- TLB invalidate all operations

This bit is a hypervisor resource.

Programming Note

Any TLB entry with IPROT = 0 is volatile and may be evicted for the following reasons even though software didn't explicitly remove or invalidate the entry.

- Generous TLB invalidations (***tlbivax*** and ***tlbilx***)
- TLB updates due to Page Table translations <E.PT>
- Hardware replacement algorithm on a ***tlbwe*** instruction if $MMUCFG_{HES}$=1 and $MAS0_{HES}$ =1.

On a virtualized implementation, a TLB entry with IPROT = 0 may be evicted at any time.

TLB entry with IND=0	TLB entry with IND=1
UX	$SPSIZE_0$
SX	$SPSIZE_1$
UW	$SPSIZE_2$
SW	$SPSIZE_3$
UR	$SPSIZE_4$
SR	RPN_{52}

Figure 20. Overlaid TLB Field Example

Virtualized Implementation Note

On virtualized implementations, programmers should weigh the degredation that may be caused by execute-only pages against the need for the security availed by the protection.

6.7.2 Virtual Address Spaces

There are two separate address spaces supported. MSR_{IS} and MSR_{DS} are used to indicate the address space used for instruction and data accesses respectively. MSR_{IS} and MSR_{DS} can be set independently to access address space 0 or address space 1. TLB entries have a corresponding TS bit which is compared either to MSR_{IS} or MSR_{DS} for instruction and data accesses respectively to determine if the TLB entry is a match.

Programming Note

Because MSR_{IS} and MSR_{DS} are set to 0 by the hardware on interrupt, the Operating System software that handles interrupts should be designed to run with AS=0. As a result, Operating System software that wishes to, for example, use one address space for user and the other for supervisor should use AS=0 for supervisor and AS=1 for user.

If the Embedded.Hypervisor category is supported, the above two address spaces exist for each logical parti-

tion and for both the guest and non-guest states within each logical partition. The Logical Partition ID Register identifies the partition and a field in the TLB entry (TLPID) specifies which partition that TLB entry is associated with. The Guest State (GS) bit in the Machine State Register identifies the guest state or non-guest state and a bit in the TLB entry (TGS) specifies which of these states that TLB entry is associated with.

Load, *Store*, *Cache Management*, and *Branch* instructions and next-sequential-instruction fetches produce a 64-bit effective address. A one-bit address space identifier and a process identifier are prepended to the effective address to form the virtual address. If the Embedded.Hypervisor category is supported, this address is also prepended by a Logical Partition ID and Guest State bit. The Logical Partition ID is provided by the contents of LPIDR and the Guest State bit is provided by the MSR_{GS}. For instruction fetches, the address space identifier is provided by MSR_{IS} and the process identifier is provided by the contents of the Process ID Register. For data storage accesses, the address space identifier is provided by the MSR_{DS} and the process identifier is provided by the contents of the Process ID Register.

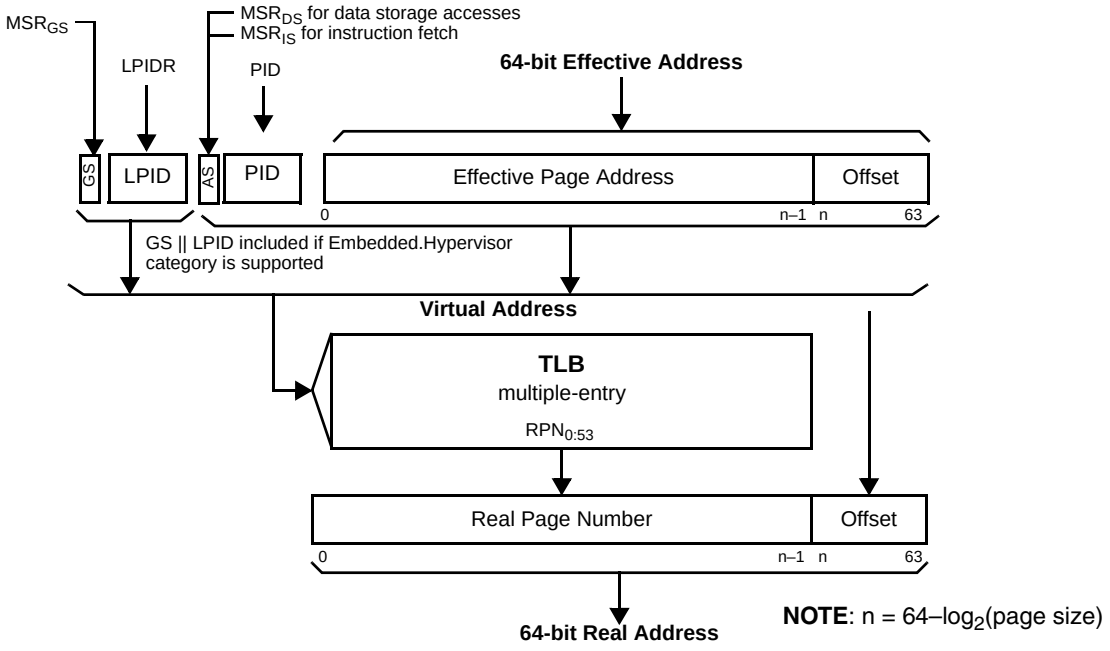


Figure 21. Effective-to-Virtual-to-Real TLB Address Translation Flow

6.7.3 TLB Address Translation

A program references memory by using the effective address computed by the hardware when it executes a *Load*, *Store*, *Cache Management*, or *Branch* instruction, and when it fetches the next instruction. A virtual address is formed from the effective address as described in Section 6.7.2 and the virtual address is translated to a real address according to the procedures described in this section. The storage subsystem uses the real address for the access. All storage access effective addresses are translated to real addresses using the TLB mechanism. See Figure 21.

The virtual address is used to locate the associated entry in the TLB. The address space identifier, the process identifier, and the effective address of the storage access are compared to the Translation Address Space bit (TS), the Translation ID field (TID), and the value in the Effective Page Number field (EPN), respectively, of each TLB entry. If the Embedded.Hypervisor category is supported, the Logical Partition ID and the Guest State bit are also compared to the Translation Logical Partition ID (TLPID) and Translation Guest State (TGS) of each TLB entry. Figure 22 illustrates the criteria for a virtual address to match a specific TLB entry for a direct TLB entry (IND = 0). See Section 6.7.4 for details on Page Table translation using an indirect TLB entry.

The virtual address of a storage access matches a direct TLB entry if the first four following conditions are true, and, additionally, if the Embedded.Page Table category is supported, the fifth condition is true, and, addi-

tionally, if the Embedded.Hypervisor category is supported, the last two conditions are true.

- The Valid bit of the TLB entry is 1.
- The value of the address specifier for the storage access (MSR_{IS} for instruction fetches, MSR_{DS} for data storage accesses) is equal to the value in the TS bit of the TLB entry.
- The value of the process identifier in the PID register is equal to the value in the TID field of the TLB entry or the value of the TID field of the TLB entry is equal to 0.
- The contents of bits 0:*n*–1 of the effective address of the storage access are equal to the value of bits 0:*n*-1 of the EPN field of the TLB entry (where *n*=64-$\log_2$(*page size* in bytes) and *page size* is specified by the value of the SIZE field of the TLB entry). See Table 2 and Table 3.
- One of the following conditions is true.
  - The TLB array supports the IND bit ($TLBnCFG_{IND}$ = 1) and the IND bit of the TLB entry is equal to 0.
  - The TLB array does not support the IND bit ($TLBnCFG_{IND}$ = 0).
- Either the value of the logical partition identifier in LPIDR is equal to the value of the TLPID field of the TLB entry, or the value of the TLPID field of the TLB entry is equal to 0.
- The value of the guest state bit (MSR_{GS}) is equal to the value of the TGS bit of the TLB entry.

Table 2: Page Size and Effective Address to TLB EPN Comparison for MAV = 1.0

SIZE	Page Size (4^{SIZE}KB)	EA to EPN Comparison (bits 0:53–2×SIZE)
0b0000	1 KB	$EPN_{0:53}$ =? $EA_{0:53}$
0b0001	4KB	$EPN_{0:51}$ =? $EA_{0:51}$
0b0010	16KB	$EPN_{0:49}$ =? $EA_{0:49}$
0b0011	64KB	$EPN_{0:47}$ =? $EA_{0:47}$
0b0100	256KB	$EPN_{0:45}$ =? $EA_{0:45}$
0b0101	1MB	$EPN_{0:43}$ =? $EA_{0:43}$
0b0110	4MB	$EPN_{0:41}$ =? $EA_{0:41}$
0b0111	16MB	$EPN_{0:39}$ =? $EA_{0:39}$
0b1000	64MB	$EPN_{0:37}$ =? $EA_{0:37}$
0b1001	256MB	$EPN_{0:35}$ =? $EA_{0:35}$
0b1010	1GB	$EPN_{0:33}$ =? $EA_{0:33}$
0b1011	4GB	$EPN_{0:31}$ =? $EA_{0:31}$
0b1100	16GB	$EPN_{0:29}$ =? $EA_{0:29}$
0b1101	64GB	$EPN_{0:27}$ =? $EA_{0:27}$
0b1110	256GB	$EPN_{0:25}$ =? $EA_{0:25}$
0b1111	1TB	$EPN_{0:23}$ =? $EA_{0:23}$

Table 3: Page Size and Effective Address to TLB EPN Comparison for MAV = 2.0

SIZE	Page Size (2^{SIZE}KB)	EA to EPN Comparison (bits 0:53–SIZE)
0b00000	1KB	$EPN_{0:53}$ =? $EA_{0:53}$
0b00001	2KB	$EPN_{0:52}$ =? $EA_{0:52}$
0b00010	4KB	$EPN_{0:51}$ =? $EA_{0:51}$
0b00011	8KB	$EPN_{0:50}$ =? $EA_{0:50}$
0b00100	16KB	$EPN_{0:49}$ =? $EA_{0:49}$
0b00101	32KB	$EPN_{0:48}$ =? $EA_{0:48}$
0b00110	64KB	$EPN_{0:47}$ =? $EA_{0:47}$
0b00111	128KB	$EPN_{0:46}$ =? $EA_{0:46}$
0b01000	256KB	$EPN_{0:45}$ =? $EA_{0:45}$
0b01001	512KB	$EPN_{0:44}$ =? $EA_{0:44}$
0b01010	1MB	$EPN_{0:43}$ =? $EA_{0:43}$
0b01011	2MB	$EPN_{0:42}$ =? $EA_{0:42}$
0b01100	4MB	$EPN_{0:41}$ =? $EA_{0:41}$
0b01101	8MB	$EPN_{0:40}$ =? $EA_{0:40}$
0b01110	16MB	$EPN_{0:39}$ =? $EA_{0:39}$
0b01111	32MB	$EPN_{0:38}$ =? $EA_{0:38}$
0b10000	64MB	$EPN_{0:37}$ =? $EA_{0:37}$
0b10001	128MB	$EPN_{0:36}$ =? $EA_{0:36}$
0b10010	256MB	$EPN_{0:35}$ =? $EA_{0:35}$
0b10011	512MB	$EPN_{0:34}$ =? $EA_{0:34}$
0b10100	1GB	$EPN_{0:33}$ =? $EA_{0:33}$
0b10101	2GB	$EPN_{0:32}$ =? $EA_{0:32}$
0b10110	4GB	$EPN_{0:31}$ =? $EA_{0:31}$
0b10111	8GB	$EPN_{0:30}$ =? $EA_{0:30}$
0b11000	16GB	$EPN_{0:29}$ =? $EA_{0:29}$
0b11001	32GB	$EPN_{0:28}$ =? $EA_{0:28}$
0b11010	64GB	$EPN_{0:27}$ =? $EA_{0:27}$
0b11011	128GB	$EPN_{0:26}$ =? $EA_{0:26}$
0b11100	256GB	$EPN_{0:25}$ =? $EA_{0:25}$
0b11101	512GB	$EPN_{0:24}$ =? $EA_{0:24}$
0b11110	1TB	$EPN_{0:23}$ =? $EA_{0:23}$
0b11111	2TB	$EPN_{0:22}$ =? $EA_{0:22}$

Programming Note

An implementation need not support all page sizes.

If the virtual address of the storage access matches a TLB entry in accordance with the selection criteria specified in the preceding paragraph, the value of the Real Page Number field (RPN) of the matching TLB entry provides the real page number portion of the real address. Let *n*=64–$\log_2$(*page size* in bytes) where *page size* is specified by the SIZE field of the TLB entry. Bits *n*:63 of the effective address are appended to bits 0:*n*–1 of the 54-bit RPN field of the matching TLB entry to produce the 64-bit real address (i.e. RA = $RPN_{0:n-1}$ || $EA_{n:63}$) that is presented to main storage to perform the storage access. The page size is determined by the value of the SIZE field of the matching TLB entry. See Table 4 and Table 5. Depending on the page size, certain RPN bits of the matching TLB entry must be zero as shown in Table 4 and Table 5. Otherwise, it is implementation-dependent whether the

address translation is performed as if these RPN bits are 0 or as if the corresponding RA bits are undefined values, or either an Instruction Storage exception (for an instruction fetch) or Data Storage exception (for a data access) occurs. If the specified page size is not supported by the implementation's TLB array, it is implementation-dependent whether the address translation is performed as if the page size was a smaller size or either an Instruction Storage exception (for an instruction fetch) or Data Storage exception (for a data access) occurs.

Table 4: Real Address Generation for MAV = 1.0

SIZE	Page Size (4^{SIZE} KB)	RPN Bits Required to be Equal to 0	Real Address
0b0000	1 KB	none	$RPN_{0:53}$ || $EA_{54:63}$
0b0001	4KB	$RPN_{52:53}$=0	$RPN_{0:51}$ || $EA_{52:63}$
0b0010	16KB	$RPN_{50:53}$=0	$RPN_{0:49}$ || $EA_{50:63}$
0b0011	64KB	$RPN_{48:53}$=0	$RPN_{0:47}$ || $EA_{48:63}$
0b0100	256KB	$RPN_{46:53}$=0	$RPN_{0:45}$ || $EA_{46:63}$
0b0101	1MB	$RPN_{44:53}$=0	$RPN_{0:43}$ || $EA_{44:63}$
0b0110	4MB	$RPN_{42:53}$=0	$RPN_{0:41}$ || $EA_{42:63}$
0b0111	16MB	$RPN_{40:53}$=0	$RPN_{0:39}$ || $EA_{40:63}$
0b1000	64MB	$RPN_{38:53}$=0	$RPN_{0:37}$ || $EA_{38:63}$
0b1001	256MB	$RPN_{36:53}$=0	$RPN_{0:35}$ || $EA_{36:63}$
0b1010	1GB	$RPN_{34:53}$=0	$RPN_{0:33}$ || $EA_{34:63}$
0b1011	4GB	$RPN_{32:53}$=0	$RPN_{0:31}$ || $EA_{32:63}$
0b1100	16GB	$RPN_{30:53}$=0	$RPN_{0:29}$ || $EA_{30:63}$
0b1101	64GB	$RPN_{28:53}$=0	$RPN_{0:27}$ || $EA_{28:63}$
0b1110	256GB	$RPN_{26:53}$=0	$RPN_{0:25}$ || $EA_{26:63}$
0b1111	1TB	$RPN_{24:53}$=0	$RPN_{0:23}$ || $EA_{24:63}$

Table 5: Real Address Generation for MAV = 2.0

SIZE	Page Size (4^{SIZE} KB)	RPN Bits Required to be Equal to 0	Real Address
0b00000	1KB	none	$RPN_{0:53}$ || $EA_{54:63}$
0b00001	2KB	$RPN_{53:53}$=0	$RPN_{0:52}$ || $EA_{53:63}$
0b00010	4KB	$RPN_{52:53}$=0	$RPN_{0:51}$ || $EA_{52:63}$
0b00011	8KB	$RPN_{51:53}$=0	$RPN_{0:50}$ || $EA_{51:63}$
0b00100	16KB	$RPN_{50:53}$=0	$RPN_{0:49}$ || $EA_{50:63}$
0b00101	32KB	$RPN_{49:53}$=0	$RPN_{0:48}$ || $EA_{49:63}$
0b00110	64KB	$RPN_{48:53}$=0	$RPN_{0:47}$ || $EA_{48:63}$
0b00111	128KB	$RPN_{47:53}$=0	$RPN_{0:46}$ || $EA_{47:63}$
0b01000	256KB	$RPN_{46:53}$=0	$RPN_{0:45}$ || $EA_{46:63}$
0b01001	512KB	$RPN_{45:53}$=0	$RPN_{0:44}$ || $EA_{45:63}$
0b01010	1MB	$RPN_{44:53}$=0	$RPN_{0:43}$ || $EA_{44:63}$
0b01011	2MB	$RPN_{43:53}$=0	$RPN_{0:42}$ || $EA_{43:63}$
0b01100	4MB	$RPN_{42:53}$=0	$RPN_{0:41}$ || $EA_{42:63}$
0b01101	8MB	$RPN_{41:53}$=0	$RPN_{0:40}$ || $EA_{41:63}$
0b01110	16MB	$RPN_{40:53}$=0	$RPN_{0:39}$ || $EA_{40:63}$
0b01111	32MB	$RPN_{39:53}$=0	$RPN_{0:38}$ || $EA_{39:63}$
0b10000	64MB	$RPN_{38:53}$=0	$RPN_{0:37}$ || $EA_{38:63}$
0b10001	128MB	$RPN_{37:53}$=0	$RPN_{0:36}$ || $EA_{37:63}$
0b10010	256MB	$RPN_{36:53}$=0	$RPN_{0:35}$ || $EA_{36:63}$
0b10011	512MB	$RPN_{35:53}$=0	$RPN_{0:34}$ || $EA_{35:63}$
0b10100	1GB	$RPN_{34:53}$=0	$RPN_{0:33}$ || $EA_{34:63}$
0b10101	2GB	$RPN_{33:53}$=0	$RPN_{0:32}$ || $EA_{33:63}$
0b10110	4GB	$RPN_{32:53}$=0	$RPN_{0:31}$ || $EA_{32:63}$
0b10111	8GB	$RPN_{31:53}$=0	$RPN_{0:30}$ || $EA_{31:63}$
0b11000	16GB	$RPN_{30:53}$=0	$RPN_{0:29}$ || $EA_{30:63}$
0b11001	32GB	$RPN_{29:53}$=0	$RPN_{0:28}$ || $EA_{29:63}$
0b11010	64GB	$RPN_{28:53}$=0	$RPN_{0:27}$ || $EA_{28:63}$
0b11011	128GB	$RPN_{27:53}$=0	$RPN_{0:26}$ || $EA_{27:63}$
0b11100	256GB	$RPN_{26:53}$=0	$RPN_{0:25}$ || $EA_{26:63}$
0b11101	512GB	$RPN_{25:53}$=0	$RPN_{0:24}$ || $EA_{25:63}$
0b11110	1TB	$RPN_{24:53}$=0	$RPN_{0:23}$ || $EA_{24:63}$
0b11111	2TB	$RPN_{23:53}$=0	$RPN_{0:22}$ || $EA_{23:63}$

A TLB Miss exception occurs if there is no valid matching direct entry in the TLB for the page specified by the virtual address (Instruction or Data TLB Error interrupt) and, if the Embedded.Page Table category is supported, there is no matching indirect entry (see Section 6.7.4). A TLB Miss exception for an instruction fetch will result in an Instruction TLB Miss exception type Instruction TLB Error interrupt. A TLB Miss exception for a data storage access will result in a Data TLB Miss exception type Data TLB Error interrupt. Although the possibility exists to place multiple direct and/or multiple indirect entries into the TLB that match a specific virtual address, assuming a set-associative or fully-associative organization, doing so is a programming error. Either one of the matching entries is used or a Machine Check exception occurs if there are multiple matching direct entries or multiple matching indirect entries for an instruction or data access.

The rest of the matching TLB entry provides the access control bits (UX, SX, UW, SW, UR, SR, VF), and stor-

age control attributes (ACM [implementation-dependent], VLE <VLE>, U0, U1, U2, U3, W, I, M, G, E) for the storage access. The access control bits and storage control attribute bits specify whether or not the access is allowed and how the access is to be performed. See Sections 6.7.6 and 6.11.4.

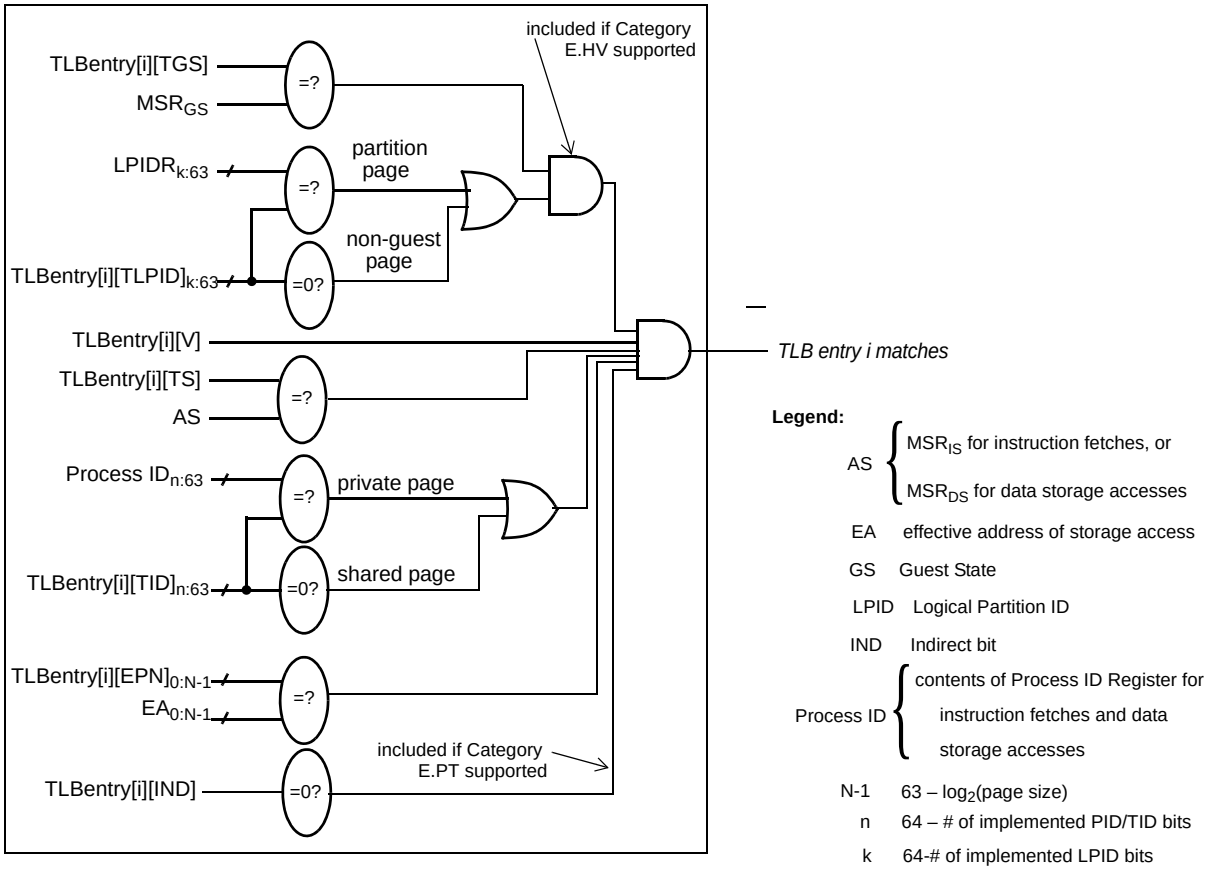


Figure 22. Address Translation: Virtual Address to direct TLB Entry Match Process

6.7.4 Page Table Address Translation [Category: Embedded.Page Table]

A hardware Page Table is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers. There can be many hardware Page Tables. Each Page Table is defined by an indirect TLB entry. An indirect TLB entry is an entry that has its IND bit equal to 1.

An indirect TLB entry matches the virtual address if all fields match per Section 6.7.4 except for the IND bit and the IND bit of the TLB entry is 1. If there is no matching direct TLB entry, but there is one and only one matching indirect entry, the indirect entry is used to access a Page Table Entry (PTE) if the VF bit of the indirect TLB entry is 0. If the VF bit of this indirect TLB entry is 1, a Virtualization Fault exception occurs. If the PTE is a valid entry (V bit = 1), the PTE is used to translate the address. The PTE includes the abbreviated RPN (ARPN), page size (PS), storage control (WIMGE), implementation-dependent bits, and storage access control bits (BAP,R,C) that are used for the access. If the Embedded.Page Table and Embedded.Hypervisor categories are both supported, the Embedded.Hypervisor.LRAT category is supported. In this case, the RPN from the PTE is treated as a Logical Page Number (LPN) and the LPN is translated by the LRAT into an RPN. See Section 6.9. If there is more than one matching direct TLB entry or more than one matching indirect TLB entry, any one of the duplicate entries may be used or Machine Check exception may occur.

See Section 6.7.5 for the rules that software must follow when updating the Page Table.

Programming Note

Even when the Embedded.Hypervisor category is supported, a Page Table can optionally be treated as a guest supervisor resource due to the LRAT.

If the Page Table is treated as a hypervisor resource, the Page Table must be placed in storage to which only the hypervisor has access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data. An LRAT identity mapping (LPN=RPN) can be used when the Page Tables are treated as hypervisor resources, especially if only one LRAT entry is provided. If the LRAT identity mapping converts LPNs into RPNs that extend beyond the memory given to the partition, the Page Table Entries still provide the hypervisor with a mechanism to limit a guest's accesses to memory assigned to the partition, assuming guest execution of *TLB Management* instructions is disabled.

Programming Note

If storage accesses are to scattered virtual pages, an Embedded Page Table could be sparsely used, and, in the worst case, there could be only one valid PTE in the Page Table. In this case it would be more efficient for software to directly load TLB entries rather than have both an indirect TLBE and a direct TLBE, which is loaded from the Page Table.

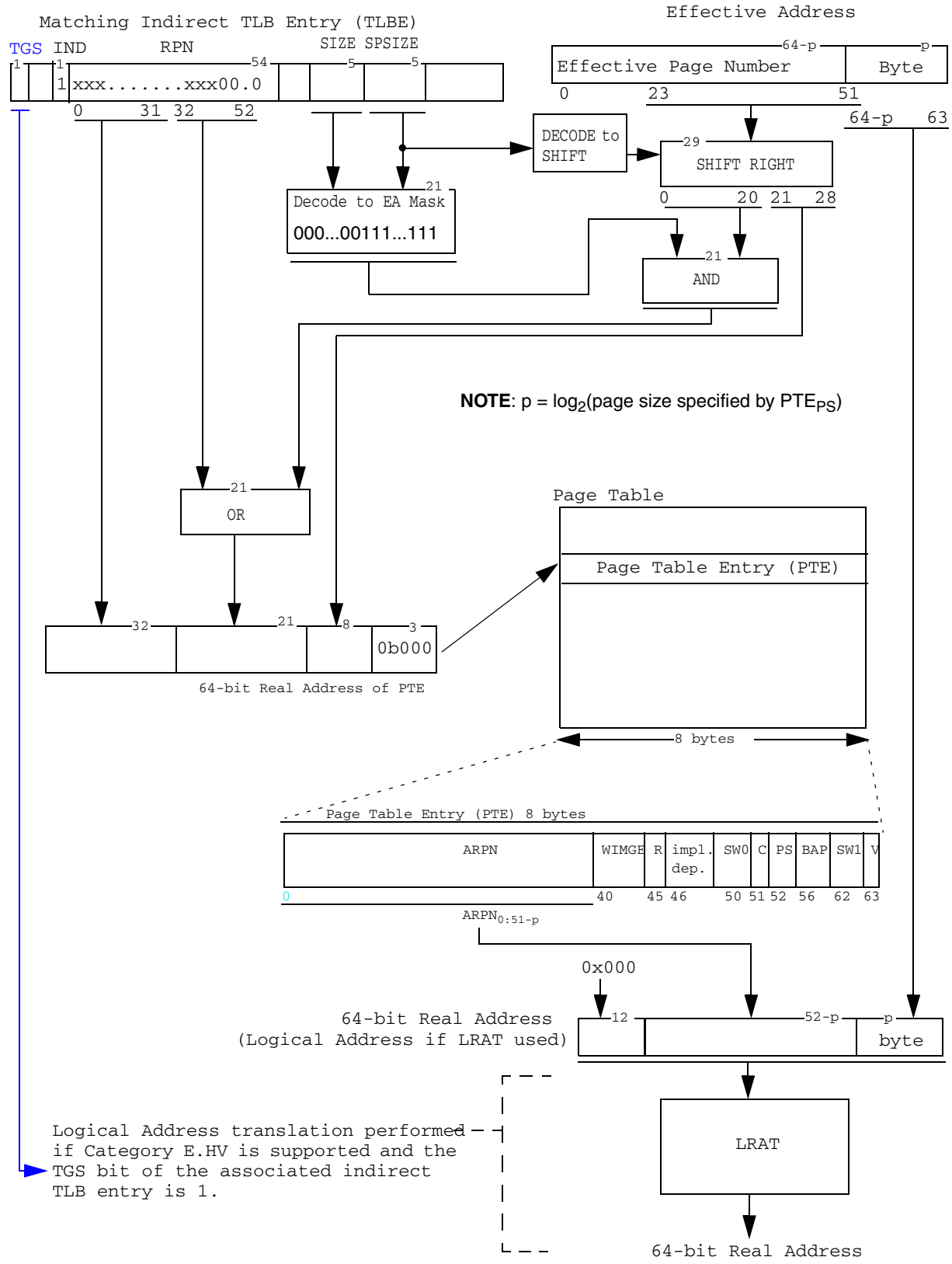


Figure 23. Page Table Translation

Figure 23 depicts the Page Table translation for a matching indirect TLB entry (TLBE). The Page Table Entry that is used to translate the Effective Address is selected by a real address formed from some combination of RPN bits from the TLBE and some EA bits. The low-order m bits of the RPN field in the indirect TLB entry must be zeros, where m is (SIZE – SPSIZE) – 7.

SIZE minus SPSIZE must be greater than 7 (corresponding to a page table size of at least 2 KB; see below under "Page Table Size and Alignment"). The SIZE and SPSIZE fields of the TLBE determine which bits of the RPN and EA are used in the following manner.

1. $EA_{23:51}$ are shifted right q bits, according to a decode of SPSIZE, to produce a 29-bit result S. The value of q is (SPSIZE – 2). Bits shifted out of the rightmost bit position are lost.
2. A 21-bit EA mask is formed based on a decode of SIZE and SPSIZE. The EA mask is $^{(29\ -\ (SIZE\ -\ SPSIZE))}0\ ||\ ^{(SIZE\ -\ SPSIZE)-8}1$.
3. The EA mask from step 2 is ANDed with the high-order 21 bits of the shifted EA result ($S_{0:20}$) from step 1 to form a 21-bit result.
4. $RPN_{32:52}$from the indirect TLB entry is ORed with the 21-bit result from step 3 to form a 21-bit result R.
5. The real address of the PTE is formed as follows:
   $RA = TLBE_{RPN[0:31]}\ ||\ R\ ||\ S_{21:28}\ ||\ 0b000$

The doubleword addressed by the real address result from step 5 is the PTE used to translate the EA if the PTE is valid (Valid bit = 1). If the PTE is valid, PTE_{PS} must be greater than or equal to the SPSIZE of the associated indirect TLB entry and must be less than or equal to the SIZE of the associated indirect TLB entry. The real address (RA) result is formed by concatenating 0x000 with the $ARPN_{0:51-p}$ from the PTE and with the low-order p bits of EA, where p is equal to log_2(page size specified by PTE_{PS}).

$RA = 0x000\ ||\ ARPN_{0:51-p}\ ||\ EA_{64-p:63}$

However, if an implementation supports a real address with only r bits, r<52, and either the Embedded.Hypervisor category is not supported or the TGS bit of the corresponding indirect TLB entry is 0, the high-order 52-r bits of PTE_{ARPN} are ignored and treated as 0s. If the Embedded.Hypervisor category is supported, an implementation supports a logical address with only q bits, q<52, and the TGS bit of the corresponding indirect TLB entry is 1, the high-order 52-q bits of PTE_{ARPN} are ignored and treated as 0s.

If the Embedded.Hypervisor.LRAT category is supported and the TGS bit of the associated indirect TLB entry is 1, the RA formed from the PTE is treated as a logical real address and translated by the LRAT. If there is no matching entry in the LRAT, an LRAT Miss exception occurs. See Section 6.9. If an LRAT Error interrupt results from this exception, ESR_{PT} is set to 1.

If the Page Table Entry that is accessed is invalid (Valid bit = 0), a Page Table Fault exception occurs. An Execute, Read, or Write Access Control exception occurs if a valid PTE is found but the access is not allowed by the access control mechanism. These exceptions are types of Instruction Storage exception or Data Storage exception, depending on whether the effective address is for an instruction fetch or for a data access.See Section 7.6.4 and Section 7.6.5 for additional information about these and other interrupt types. For either of these interrupts caused by a Page Table Fault exception or Execute, Read, or Write Access Control exception due to PTE permissions, ESR_{PT} or $GESR_{PT}$ is set to 1 ($GESR_{PT}$ if the Embedded.Hypervisor.LRAT category is supported and the interrupt is directed to the guest. Otherwise, ESR_{PT}).

Programming Note

If PTE_{PS} is greater than the SPSIZE of the associated indirect TLB entry, $2^{(PS\ -\ SPSIZE)}$ PTEs are needed for the virtual page to ensure there is no Page Table Fault exception for accesses to the page regardless of the location of the access within the page. If a Page Table Fault exception for some accesses to the page is acceptable, there is no requirement that all such PTEs for the page be valid.

Programming Note

The computation of the real address of the PTE can be understood as follows. (Some of the facts mentioned below, such as the fact that the minimum Page Table size is 2K, are covered later in the section.)

1. q is the number of EA bits above bit 52 that are part of the byte offset within the effective page. (The minimum size of a page that is mapped by a PTE is 4K, so $EA_{52:63}$ are always part of the byte offset, and SPSIZE must be at least 2.) S is the low-order 29-q bits of the EPN, prepended with q 0s.
2. The EA-mask has a number of low-order 1 bits equal to the difference between $\log_2$(# PTEs) and $\log_2$(minimum # PTEs) = 8. (The $\log_2$ of the number of PTEs in a Page Table is SIZE - SPSIZE. The minimum Page Table size is 2K and PTE size = 8 bytes, so the minimum number of PTEs is $2^{11} \div 2^3 = 2^8$.) Call this number s; i.e., s = (SIZE - SPSIZE) - 8, and the $\log_2$ of the number of PTEs in the Page Table is s+8.
3. The result is the low-order s bits of the EPN that are immediately above the lowest-order 8 EPN bits (the lowest-order 8 bits are always used to select the PTE), prepended with 21-s 0s. (If s could be greater than (29-q)-8, the "EPN" bits included in the result could include 0 bits that were shifted in step 1. However, this would correspond to (SIZE - SPSIZE) - 8 > (31 - SPSIZE) - 8, which would imply SIZE > 31, which is impossible.)
4. R consists of the high-order 21-s bits of $RPN_{32:52}$ followed by the low-order s bits of the EPN that are immediately above the lowest-order 8 EPN bits.
5. The real address of the PTE thus consists of the high-order 53-s bits of the RPN from the TLB entry, followed by the low-order s+8 bits of the EPN (recall that s+8 is the number of PTEs in the Page Table), followed by 3 0s.

Storage Control Attributes for the Page Table

A Page Table must be located in storage that is Big-Endian, Memory Coherence Required, not Caching Inhibited and not Guarded. If the translation of a virtual address matches an indirect TLB entry that has its storage control attribute E bit equal to 1, M bit equal to 0, I bit equal to 1, or G bit = 1, it is implementation-dependent whether the translation is performed as if valid values were substituted for the invalid values or as if the entry doesn't match, or either an Instruction Storage exception (for an instruction fetch) or Data Storage exception (for a data access) occurs. The Page Table is allowed to be located in storage that is Write Through Required or Not Write Through Required. However, the same W value must be used for a single thread's indirect and direct TLB entries that map the same PTE. The Implementations may require specific values for ACM and U0:U3.

Ordering of Implicit Accesses to the Page Table

The definition of "performed" given in Books II and III-E applies also to the implicit accesses to the Page Table by the thread in performing address translation. Accesses for performing address translation are considered to be loads in this respect. These implicit accesses are ordered by the ***sync*** instruction with L=0 as described below.

The *Synchronize* instruction is described in Section 4.4.3 of Book II, but only at the level required by an application programmer (***sync*** with L=0 or L=1). This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers. The ***sync*** instruction with L=0 (***sync***) has the following additional properties.

- The ***sync*** instruction provides an ordering function for all stores to the Page Table caused by *Store* instructions preceding the ***sync*** instruction with respect to lookups of the Page Table that are performed, by the thread executing the ***sync*** instruction, after the ***sync*** instruction completes. Executing a ***sync*** instruction ensures that all such stores will be performed, with respect to the thread executing the ***sync*** instruction, before any implicit accesses to the affected Page Table Entries, by such Page Table lookups, are performed with respect to that thread.
- In conjunction with the ***tlbivax*** and ***tlbsync*** instructions, the ***sync*** instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the ***tlbsync*** instruction description on page 1141.

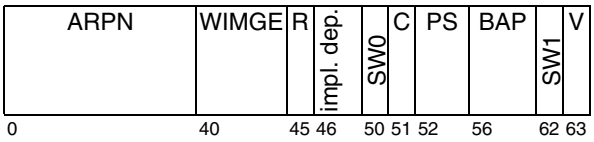
Programming Note

For instructions following a ***sync*** instruction, the memory barrier need not order implicit storage accesses for purposes of address translation.

Page Table Entry

Each Page Table Entry (PTE) maps a VPN to an RPN. If the corresponding indirect TLB entry has an LPID <E.HV> or PID value of zero, multiple VPNs are mapped by a single PTE in a Page Table pointed to by

such an indirect TLB entry. Figure 24 shows the layout of a PTE.



Bit(s)	Name	Description
0:39	ARPN	Abbreviated Real Page Number
40:44	WIMGE	Storage control attributes
45	R	Reference bit
46:49	impl.-dep.	Implementation-dependent These bits can be used to support User-Definable Storage Control Attributes, ACM and VLE. These bits are used in any combination of the following or subsets of the following: ■ 46:49 - User-Definable Storage Control Attributes (U0:U3) ■ 48 - ACM ■ 49 - VLE <VLE>
50	SW0	Available for software use
51	C	Change bit
52:55	PS	Page Size (real)
56:61	BAP	Base Access Permission bits: 0: Base UX 1: Base SX 2: Base UW 3: Base SW 4: Base UR 5: Base SR
62	SW1	Available for software use
63	V	Entry valid (V=1) or invalid (V=0)

Figure 24. Page Table Entry

The Page Size (PS) field encodes page sizes using the same encodes as the TLB_{SIZE}, except that 0b0 is prepended to the 4-bit PS value (0 || PS) to form the equivalent 5-bit encode and PS must specify a page size of 4 KB or larger. See Table 3 on page 1083.

The Abbreviated Real Page Number (ARPN) field contains the least significant 40 bits of the RPN. The full RPN associated with the PTE is formed from the ARPN prepended with 0x000, i.e. RPN = 0x000 || ARPN. Depending on the page size, certain ARPN bits must be zero. Specifically, if p>12, $ARPN_{52-p:39}$ must be zeros, where p = $\log_2$(page size specified by PTE_{PS}). If an implementation supports a real address with only r bits, r<52, and either the Embedded.Hypervisor category is not supported or the TGS bit of the corresponding indirect TLB entry is 0, the high-order 52-r bits of PTE_{ARPN} are ignored and treated as 0s for address translation. If the Embedded.Hypervisor category is supported, an implementation supports a logical address with only q bits, q<52, and the TGS bit of the corresponding indirect TLB entry is 1, the high-order 52-q bits of PTE_{ARPN} are ignored and treated as 0s for address translation.

The Base Access Permission (BAP) bits are used together with the Reference (R) and Change (C) bits to derive the storage access control bits that are used for the access. Table 6 shows how the storage access control bits are derived from the BAP, R, and C bits of the Page Table Entry.

Table 6: Storage Access Control Bits Derived from a Page Table Entry

Derived Storage Access Control	Page Table Values
UX	BAP_0 & R
SX	BAP_1 & R
UW	BAP_2 & R & C
SW	BAP_3 & R & C
UR	BAP_4 & R
SR	BAP_5 & R

Programming Note

Unlike many architectures, the R and C bits in a Page Table entry are not updated by hardware.

Programming Note

The page size specified by PTE_{PS} must be consistent with the page sizes supported by a direct TLB entry of a TLB array that can be loaded from the Page Table.

An implementation need not support all page sizes.

Page Table Size and Alignment

A Page Table's size is $8 \times 2^{(SIZE - SPSIZE)}$. where SIZE is the page size specified by the SIZE field of the indirect TLB entry used to access the Page Table and SPSIZE is the sub-page size specified by the SPSIZE field of this indirect TLB entry. Page Table sizes smaller than 2 KB are not allowed and SPSIZE must be greater than or equal to 2. This implies that the Page Table size s is 2 KB $\leq$ s $\leq$ 4 GB. The Page Table is aligned on a boundary that is a multiple of its size.

TLB Update

As a result of a Page Table translation, a corresponding direct TLB entry is created if no exception occurs, is optionally created if certain exceptions occur, and is not created if certain other exceptions occur.

If no exception occurs, a direct TLB entry is written to create an entry corresponding to the virtual address and the contents of the PTE that was used to translate the virtual address. In this case, hardware selects the TLB array and TLB entry to be written. Any TLB array

that meets all the following criteria can be selected by the hardware.

- The TLB array supports the page size specified by PTE_{PS}.
- The TLB array can be loaded from the Page Table ($TLBnCFG_{PT}$ = 1).

If no TLB array can be selected based on these criteria, then a TLB Ineligible exception occurs. Hardware also selects the entry within the TLB array based on some implementation-dependent algorithm. However, a valid TLB entry with IPROT = 1 must not be overwritten. If all TLB entries that can be used for a specific virtual page have IPROT = 1, then a TLB Ineligible exception occurs. In the absence of a higher priority exception, an Instruction Storage or Data Storage interrupt occurs, depending on whether the Page Table translation was due to an instruction fetch or data access and ESR_{TLBI} is set to 1.

It is implementation-dependent whether a TLB entry is written as a result of a Page Table translation if a Page Table Fault exception occurs, but, if written, the valid bit of the TLB entry is set to 0. It is implementation-dependent whether a TLB entry is written as a result of a Page Table translation if an Execute, Read, or Write Access Control exception occurs. If the Embedded.Hypervisor category is supported, an interrupt caused by a Page Table Translation is directed to the hypervisor or guest as specified by the applicable EPCR bits (DSIGS and ISIGS), except that a DSI or ISI resulting from a TLBI is always directed to the hypervisor.

A TLB entry is not written as a result of a Page Table translation if an LRAT Miss exception occurs or a TLB Ineligible exception occurs.

If a TLB entry is written, the entry is written based on the values shown in Table 7.

.

Table 7: TLB Update after Page Table Translation	
TLB field	**Load Value**
$EPN_{0:p-1}$	$EA_{0:p-1}$, where p = 64 - $\log_2$(*page size* in bytes) and *page size* is specified by PTE_{PS}. Any low-order EPN bits in the TLB entry that correspond to byte offsets with the page are undefined.
TS	TS from indirect TLB entry
SIZE	PTE_{PS}
TLPID [Category: E.HV]	TLPID from indirect TLB entry
TGS [Category: E.HV]	TGS from indirect TLB entry
TID	TID from indirect TLB entry
V	PTE_V
IND	0
RPN	if E.HV.LRAT not supported, then RPN = 0x000 || PTE_{ARPN} || 0b00 else LPN = 0x000 || PTE_{ARPN} || 0b00 RPN = result of LRAT translation of LPN & PTE_{PS}
WIMGE	PTE_{WIMGE}
U0:U3, ACM, VLE	$PTE_{impl.-dep.}$ (which of the implementation-dependent TLB bits are loaded and which of the $PTE_{46:49}$ bits is used to load each TLB bit are implementation-dependent)
UR, UW, UX, SR, SW, SX	Derived Storage Access Control from PTE_{BAP}, PTE_R, and PTE_C. See Table 6.
VF	0
IPROT	0

If implementations write TLB entries for out-of-order Page Table translations, a mechanism for disabling such TLB updates must be provided by the implementation in order for software to preload a TLB array without the possibility of creating multiple direct entries for the same virtual address.

Programming Note

As a hardware simplification the architecture allows a TLB entry to be written with the valid bit set to 0 if a Page Table Fault exception occurs. A replacement of a valid TLB entry by an invalid entry is typically not a significant performance impact since software often swaps in the virtual page and creates a valid PTE for the page.

Programming Note

Only software creates indirect TLB entries, but both software and hardware create direct TLB entries. Unless a *TLB Write Conditional* instruction is used, software must avoid creating a direct TLB entry for a VPN that may also be simultaneously translated via a Page Table by a thread sharing the TLB. Otherwise multiple, direct TLB entries could be created. If software is preloading a TLB with a direct TLB entry and there is already an indirect TLB entry that could be used to translate the same VPN, software must ensure that no program on any thread sharing the TLB is accessing the VPN. Otherwise multiple, direct TLB entries could be created. If the Embedded.TLB Write Conditional category is supported, a *TLB Write Conditional* instruction can be used to create a direct TLB entry for the same VPN that may also be mapped by an existing indirect entry and Page Table Entry, assuming the page size specified by the *TLB Write Conditional* and PTE are identical.

6.7.5 Page Table Update Synchronization Requirements [Category: Embedded.Page Table]

This section describes rules that software must follow when updating the Page Table. Otherwise, TLB entries for outdated PTEs may remain valid. This section includes suggested sequences of operations for some representative cases.

In the sequences of operations shown in the following subsections, any alteration of a Page Table Entry (PTE) that corresponds to a single line in the sequence is assumed to be done using a *Store* instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a store doubleword operation is done using two *Store Word* instructions).

As described in Section 6.5, stores are not performed out-of-order. Moreover, address translations associated with instructions preceding the corresponding Store instructions are not performed again after the stores have been performed. (These address translations must have been performed before the store was determined to be required by the sequential execution model, because they might have caused an exception.) As a result, an update to a PTE need not be preceded by a context synchronizing operation.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the PTE are to be used for address translations associated with subsequent instructions.

As noted in the description of the *Synchronize* instruction in Section 4.4.3 of Book II, address translation

associated with instructions which occur in program order subsequent to the *Synchronize* may actually be performed prior to the completion of the *Synchronize*. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

Programming Note

In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the ***rfi*** instruction that returns from the interrupt handler may provide the required context synchronization.

Page Table Entries must not be changed in a manner that causes an implicit branch.

6.7.5.1 Page Table Updates

When Page Tables are in use, TLBs are non-coherent caches of the Page Table. TLB entries must be invalidated explicitly with one of the methods described in Section 6.11.4.3.

Unsynchronized lookups in the Page Table continue even while it is being modified. Any thread, including a thread on which software is modifying the Page Table, may look in the Page Table at any time in an attempt to translate a virtual address. When modifying a PTE, software must ensure that the PTE's Valid bit is 0 if the PTE is inconsistent (e.g., if the BAP field is not correct for the current ARPN field).

The sequences of operations shown in the following subsections assume a multi-threaded processor environment. In a system consisting of only a single-threaded processor the ***tlbsync*** must be omitted, and the ***mbar*** that separates the ***tlbivax*** from the ***tlbsync*** can be omitted. In a multi-threaded processor environment, when ***tlbilx*** is used instead of ***tlbivax*** in a Page Table update, the synchronization requirements are the same as when ***tlbivax*** is used in a system consisting of only a single-threaded processor.

Programming Note

For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another thread, the ***sync*** instruction at the end of the sequence should be followed by a *Store* instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the ***sync*** instruction ensures that if a *Load* instruction executed by another thread returns the chosen value from location X, the sequence's stores to the Page Table have been performed with respect to that other thread. The *Load* instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE.)

This Note assumes that the Page Table and location X are in storage that is Memory Coherence Required.

6.7.5.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The Valid bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE_ARPN,WIMGE,R,SW0,C,PS,BAP,SW1,V←new values
sync      /* order updates before next
             Page Table lookup and before
             next data access.            */
```

On a 32-bit implementation, the following sequence can be used.

```
PTE_ARPN(0:31) ← new value
mbar      /* order 1st update before 2nd */
PTE_ARPN[32:39],WIMGE,R,SW0,C,PS,BAP,SW1,V ←new values
sync      /* order updates before next
             Page Table lookup and before
             next data access.            */
```

6.7.5.1.2 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
PTE_V ← 0  /* (other fields don't matter)         */
sync       /* order update before tlbivax and
              before next Page Table lookup       */
tlbivax(old_LPID,old_GS,old_PID,old_AS,old_VA,
        old_ISIZE, old_IND)
          /*invalidate old translation*/
mbar       /* order tlbivax before tlbsync       */
tlbsync    /* order tlbivax before sync          */
sync       /* order tlbivax, tlbsync, and update
              before next data access            */
```

6.7.5.1.3 Modifying a Page Table Entry

General Case

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the old PTE can be deleted and a new one added using the sequences described in the two preceding sections, in order to ensure that the translation instantiated by the old entry is no longer available, maintain a consistent state, modify the PTE, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

Modifying the SW0 and SW1 Fields

If the only change being made to a valid entry is to modify the SW0 or SW1 fields, the following sequence suffices because the SW0 and SW1 fields are not used by the thread.

```
loop: ldarx r1 ← PTE /* load of PTE */
      r1 ← new SW0,SW1 /* replace SW0,SW1 in r1*/
      stdcx. PTE ←  r1 /* store of PTE
             if still reserved (new SW0 or SW1
             values, other fields unchanged) */
      bne-   loop    /* loop if lost reservation */
```

A ***lwarx***/***stwcx.*** pair (specifying the low-order word of the PTE) can be used instead of the ***ldarx*** /***stdcx.*** pair shown above.

Modifying a Reference or Change Bit

If the only change being made to a valid entry is to modify the R bit, the C bit or both, the preceding sequence suffices if the precise instant that hardware Page Table translations use the new value doesn't matter. Reference, Change, and Valid bits are in different bytes to facilitate the use of a *Store* instruction of a byte to modify a Reference or Change bit instead of a ***ldarx*** and ***stdcx.***. However, the correctness of doing so is a software issue beyond the scope of this architecture.

6.7.5.2 Invalidating an Indirect TLB Entry

The following sequence can be used to ensure that translations by a Page Table that is mapped via an indirect entry will no longer occur and that the storage used for the Page Table can then be re-used for other purposes.

```
for all valid PTEs mapped by the indirect TLB entry
  PTE_V ← 0/* (other fields don't matter) */
sync       /* order stores to PTEs */
for all valid PTEs mapped by the indirect TLB entry
  tlbivax(old_LPID,old_GS,old_PID,old_AS,old_VA,
          old_ISIZE, MAS6_SIND = 0)
            /*invalidate old PTE translations*/
tbivax(old_LPID,old_GS,old_PID,old_AS,old_VA,
          old_ISIZE, MAS6_SIND = 1)
            /*invalidate old indirect TLB entry */
mbar       /* order tlbivax before tlbsync       */
tlbsync    /* order tlbivax before sync          */
sync       /* order tlbivax, tlbsync, and update
              before next data access to the storage
              locations occupied by the Page Table
              pointed to by the old indirect TLBE */
```

6.7.6 Storage Access Control

After a matching TLB entry has been identified, the access control mechanism selectively grants execute access, read access, and write access separately for user mode versus supervisor mode. If the Embedded.Hypervisor category is supported, the access control mechanism selectively controls an access so that the access can be virtualized by the hypervisor if appropriate. Figure 25 illustrates the access control process and is described in detail in Sections 6.7.6.1 through 6.7.6.6.

An Execute, Read, or Write Access Control exception or Virtualization Fault exception occurs if the appropriate TLB entry is found but the access is not allowed by the access control mechanism (Instruction or Data Storage interrupt). See Section 7.6 for additional information about these and other interrupt types. In certain cases, Execute, Read, and Write Access Control exceptions and Virtualization Fault exceptions may result in the restart of (re-execution of at least part of) a *Load* or *Store* instruction.

Implementations may provide additional access control capabilities beyond those described here.

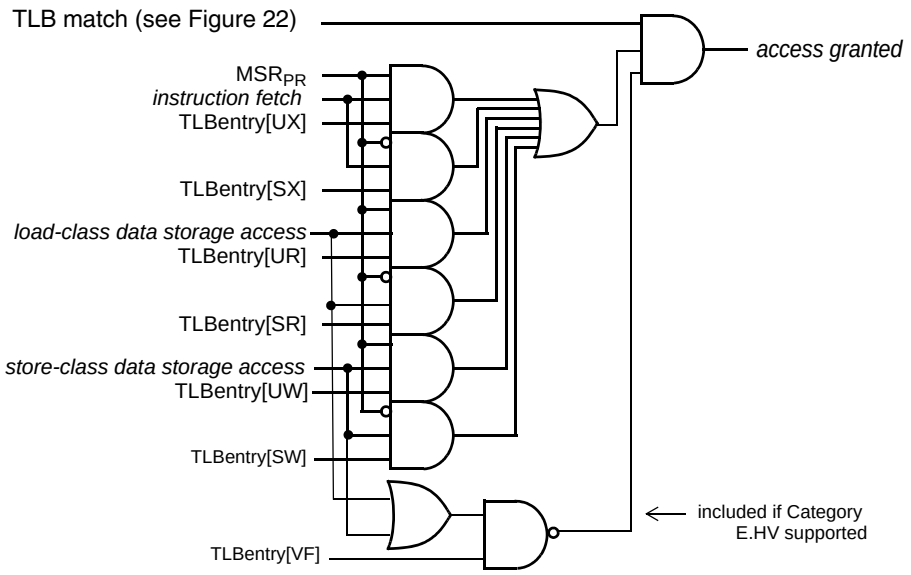


Figure 25. Access Control Process

6.7.6.1 Execute Access

The UX and SX bits of the TLB entry control *execute* access to the page (see Table 8).

Instructions may be fetched and executed from a page in storage while in user state (MSR$_{PR}$=1) if the UX access control bit for that page is equal to 1. If the UX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in user state.

Instructions may be fetched and executed from a page in storage while in supervisor state (MSR$_{PR}$=0) if the SX access control bit for that page is equal to 1. If the SX access control bit is equal to 0, then instructions from that page will not be fetched, and will not be placed into any cache as the result of a fetch request to that page while in supervisor state.

Instructions from no-execute storage may be in the instruction cache if they were fetched into that cache when their effective addresses were mapped to execute permitted storage. Software need not flush a page from the instruction cache before marking it no-execute.

Furthermore, if the sequential execution model calls for the execution of an instruction from a page that is not enabled for execution (i.e. UX=0 when MSR$_{PR}$=1 or SX=0 when MSR$_{PR}$=0), an Execute Access Control exception type Instruction Storage interrupt is taken.

6.7.6.2 Write Access

The UW and SW bits of the TLB entry control *write* access to the page (see Table 8).

Store operations (including *Store*-class *Cache Management* instructions) are permitted to a page in storage while in user state (MSR$_{PR}$=1) if the UW access control bit for that page is equal to 1. If the UW access control bit is equal to 0, then execution of the *Store* instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

Store operations (including *Store*-class *Cache Management* instructions) are permitted to a page in storage while in supervisor state (MSR$_{PR}$=0) if the SW access control bit for that page is equal to 1. If the SW access control bit is equal to 0, then execution of the *Store* instruction is suppressed and a Write Access Control exception type Data Storage interrupt is taken.

6.7.6.3 Read Access

The UR and SR bits of the TLB entry control *read* access to the page (see Table 8).

Load operations (including *Load*-class *Cache Management* instructions) are permitted from a page in storage while in user state (MSR$_{PR}$=1) if the UR access control bit for that page is equal to 1. If the UR access control bit is equal to 0, then execution of the *Load* instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

Load operations (including *Load*-class *Cache Management* instructions) are permitted from a page in storage while in supervisor state (MSR$_{PR}$=0) if the SR access control bit for that page is equal to 1. If the SR access

control bit is equal to 0, then execution of the *Load* instruction is suppressed and a Read Access Control exception type Data Storage interrupt is taken.

6.7.6.4 Virtualized Access <E.HV>

The VF bit of the TLB entry prevents a Load or Store access to the page (see Table 8).

The translation of a *Load* or *Store* (including *Cache Management* instructions) operand address that uses a TLB entry with the Translation Virtualization Fault field equal to 1 causes a Virtualization Fault exception type Data Storage interrupt regardless of the settings of the permission bits and regardless of whether the TLB entry is a direct or indirect entry. The resulting Data Storage interrupt is directed to the hypervisor state.

6.7.6.5 Storage Access Control Applied to Cache Management Instructions

dcbi, ***dci***, ***dcbz***, and ***dcbzep*** instructions are treated as *Stores* since they can change data (or cause loss of data by invalidating dirty lines). As such, they can cause Write Access Control exception type Data Storage interrupts and Virtualization Fault exception type Data Storage interrupts. If an implementation first flushes a line before invalidating it during a ***dcbi***, the ***dcbi*** is treated as a *Load* since the data is not modified.

dcba instructions are treated as *Stores* since they can change data. However, they do not cause Write Access Control exceptions. A ***dcba*** instruction will not cause a virtualization fault ($TLB_{VF} = 1$).

dcblc, ***dcbtls***, ***dcblq.***, ***icblc***, ***icbtls***, ***icblq.***, ***icbi***, and ***icbiep*** instructions are treated as *Loads* with respect to protection. As such, they can cause Read Access Control exception type Data Storage interrupts and Virtualization Fault exception type Data Storage interrupts.

dcbt, ***dcbtep***, and ***icbt*** instructions are treated as *Loads* with respect to protection. However, they do not cause Read Access Control exceptions. A virtualization fault on these instructions will not result in a Data Storage interrupt.

dcbtst and ***dcbtstep*** instructions are treated as *Stores* with respect to protection. However, they do not cause Write Access Control exceptions. A virtualization fault on these instructions will not result in a Data Storage interrupt.

It is implementation-dependent whether ***dcbtsls*** instructions are treated as *Loads* or *Stores* with respect to protection. As such, they can cause either Read Access Control exception type Data Storage interrupts or Write Access Control exception type Data Storage interrupts and can also cause Virtualization Fault exception type Data Storage interrupts.

dcbf, ***dcbfep***, ***dcbst***, and ***dcbstep*** instructions are treated as *Loads* with respect to protection. Flushing or storing a line from the cache is not considered a *Store* since the store has already been done to update the cache and the ***dcbf***, ***dcbfep***, ***dcbst***, or ***dcbstep*** instruction is only updating the copy in main storage. As a *Load*, they can cause Read Access Control exception type Data Storage interrupts and Virtualization Fault exception type Data Storage interrupts.

Table 8: Storage Access Control Applied to Cache Instructions

Instruction	Read Protection Violation	Write Protection Violation	Virtualization Fault[1]
dcba	No	No	No
dcbf	Yes	No	Yes
dcbfep	Yes	No	Yes
dcbi	Yes[3]	Yes[3]	Yes
dcblc	Yes	No	Yes
dcbst	Yes	No	Yes
dcbstep	Yes	No	Yes
dcbt	No	No	No
dcbtep	No	No	No
dcbtls	Yes	No	Yes
dcbtst	No	Yes[5]	No
dcbtstep	No	Yes[5]	No
dcbtstls	Yes[4]	Yes[4]	Yes[4]
dcbz	No	Yes	Yes
dcbzep	No	Yes	Yes
dci	No	No	No
icbi	Yes	No	Yes
icbiep	Yes	No	Yes
icblc	Yes[2]	No	Yes
icblq.	Yes[2]	No	Yes
icbt	No	No	No
icbtls	Yes[2]	No	Yes
ici	No	No	No

1. Category: Embedded.Hypervisor
2. ***icbtls*** and ***icblc*** require execute or read access.
3. ***dcbi*** may cause a Read or Write Access Control Exception based on whether the data is flushed prior to invalidation.
4. It is implementation-dependent whether ***dcbtstls*** is treated as a *Load* or a *Store*.
5. If an exception is detected, the instruction is treated as a no-op and no interrupt is taken.

6.7.6.6 Storage Access Control Applied to String Instructions

When the string length is zero, neither ***lswx*** nor ***stswx*** can cause Data Storage interrupts.

6.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage control attributes may be found in Section 1.6 of Book II and subsections.

6.8.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if the G bit is 1 in the TLB entry that translates the effective address.

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

Instruction fetching is not affected by the G bit.

The following rules apply to in-order execution of *Load* and *Store* instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- *Load* or *Store* instruction that causes an atomic access

  If any portion of the storage operand has been accessed and an asynchronous or imprecise interrupt is pending, the instruction completes before the interrupt occurs.

- *Load* or *Store* instruction that causes an Alignment exception, a Data TLB Error exception, or that causes a Data Storage exception.

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

Programming Note

Instruction fetching from Guarded storage is permitted. If instruction fetches from Guarded storage must be prevented, software must set access control bits for such pages to no-execute (i.e. UX=0 and SX=0).

Bit	Storage Control Attribute
$W^{1,6}$	0 - not Write Through Required 1 - Write Through Required

6.8.1.1 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exception to this rule is the following.

Load Instruction

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

6.8.2 User-Definable

User-definable storage control attributes control user-definable and implementation-dependent behavior of the storage system. The existence of these bits is implementation-dependent. These bits are both implementation-dependent and system-dependent in their effect. These bits may be used in any combination and also in combination with the other storage control attribute bits.

6.8.3 Storage Control Bits

Storage control attributes are specified on a per-page basis. These attributes are specified in storage control bits in the TLB entries. The interpretation of their values is given in Figure 26.

Bit	Storage Control Attribute
I^{6}	0 - not Caching Inhibited 1 - Caching Inhibited
M^{2}	0 - not Memory Coherence Required 1 - Memory Coherence Required

Bit	Storage Control Attribute
G	0 - not Guarded 1 - Guarded
E[3]	0 - Big-Endian 1 - Little-Endian
U0-U3[4]	User-Definable
VLE[5]	0 - non Variable Length Encoding (VLE). 1 - VLE
ACM[7]	0 - not Alternate Coherency Mode 1 - Alternate Coherency Mode (if M=1)

1 Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.
2 Support of the 1 value is optional for implementations that do not support multiprocessing, implementations that do not support this storage control attribute assume the value of the bit to be 0, and setting M=1 in a TLB entry will have no effect.
3 [Category: Embedded.Little-Endian]
4 Support for these attributes is optional.
5 [Category: VLE]
6 [Category: SAO] The combination WIMG = 0b1110 has behavior unrelated to the meanings of the individual bits. See Section 6.8.3.1, "Storage Control Bit Restrictions" for additional information.
7 The coherency method used in Alternate Coherency Mode is implementation-dependent.

Figure 26. Storage control bits

In Section 6.8.3.1 and 6.8.3.2, "access" includes accesses that are performed out-of-order.

Programming Note

In a system consisting of only a single-threaded processor that has caches, correct coherent execution does not require storage to be accessed as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

6.8.3.1 Storage Control Bit Restrictions

All combinations of W, I, M, G, and E values are permitted except those for which both W and I are 1 and M||G ≠ 0b10.

The combination WIMG = 0b1110 is used to identify the Strong Access Ordering (SAO) storage attribute (see Section 1.7.1, "Storage Access Ordering", in Book II). Setting WIMG=0b1110 in a TLB entry causes accesses to the page to behave as if WIMG=0b0010 with additional strong access ordering behavior. Only one SAO setting is provided because this attribute is not intended for general purpose programming, so a single combination of WIMG bits is supported.

References to Caching Inhibited storage (or storage with I=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the fact that the encoding uses using I=1. Conversely, references to storage that is not Caching Inhibited (or storage with I=0) apply to SAO storage or its WIMG encoding. References to Write Through Required storage (or storage with W=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the encoding using W=1. Conversely, references to storage that is not Write Through Required (or storage with W=0) apply to SAO storage or its WIMG encoding.

If a given real page is accessed concurrently as SAO storage and as non-SAO storage, the result may be characteristic of the weakly consistent model.

Programming Note

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0. For implementations that support the SAO category, the operating system should provide a means by which application programs can request SAO storage, in order to avoid confusion with the preceding guideline (since SAO is encoded using WI=0b11).

Accesses to the same storage location using two effective addresses for which the W bit differs meet the memory coherence requirements described in Section 1.6.3 of Book II if the accesses are performed by a single thread. If the accesses are performed by two or more threads, coherence is enforced by the hardware only if the W bit is the same for all the accesses.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

At any given time, data accesses to a given real page may use both Endian modes. When changing the Endian mode of a given real page for instruction fetching, care must be taken to prevent accesses while the change is made and to flush the instruction cache(s) after the change has been completed.

Setting the VLE attribute to 1 and setting the E attribute to 1 is considered a programming error and an attempt to fetch instructions from a page so marked produces an Instruction Storage Interrupt Byte Ordering Exception and sets ESR_{BO} or $GESR_{BO}$ to 1 ($GESR_{BO}$ if the Embedded.Hypervisor category is supported and the interrupt is directed to the guest. Otherwise, ESR_{BO}).

At any given time, the value of the VLE bit must be the same for all accesses to a given real page.

Programming Note

When changing the Endian mode of a given real page used for instruction fetching and the instruction cache is shared between threads, care must be taken to prevent accesses from any thread that shares the instruction cache while the change is made until the instruction cache flush has been completed.

6.8.3.2 Altering the Storage Control Bits

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no thread modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using ***dcbst, dcbstep***, ***dcbf***, ***dcbfep***, or ***dcbi***.

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using ***dcbf, dcbfep,*** or ***dcbi***, and ***icbi*** or ***icbiep*** before permitting any other accesses to the page.

Programming Note

The storage control bit alterations described above are examples of cases in which the directives for application of statements about the W and I bits to SAO given in the third paragraph of the preceding subsection must be applied. A transition from the typical WIMG=0b0010 for ordinary storage to WIMG=0b1110 for SAO storage does not require the flush described above because both WIMG combinations indicate storage that is not Caching Inhibited.

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this to are system-dependent.

Programming Note

For example, when changing the M bit in some directory-based systems, software may be required to execute ***dcbf*** or ***dcbfep*** on each thread to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

The actions required when changing the ACM bit for a given real page are system-dependent.

When changing the value of the VLE bit for a given real page, software must set the VLE bit to the new value, then, if the page was not Caching Inhibited, invalidate copies of all locations in the page from instruction cache using ***icbi*** or ***icbiep***, and then execute an ***isync*** instruction before permitting any other accesses to the page.

Programming Note

This Note suggests one example for managing reference and change recording.

When performing physical page management, it is useful to know whether a given physical page has been referenced or altered. Note that this may be more involved than knowing whether a given TLB entry has been used to reference or alter memory, since multiple TLB entries may translate to the same physical page. If it is necessary to replace the contents of some physical page with other contents, a page which has been referenced (accessed for any purpose) is more likely to be retained than a page which has never been referenced. If the contents of a given physical page are to be replaced, then the contents of that page must be written to the backing store before replacement, if anything in that page has been changed. Software must maintain records to control this process.

Similarly, when performing TLB management, it is useful to know whether a given TLB entry has been referenced. When making a decision about which entry to cast-out of the TLB, an entry which has been referenced is more likely to be retained in the TLB than an entry which has never been referenced.

Execute, Read and Write Access Control exceptions may be used to allow software to maintain reference information for a TLB entry and for its associated physical page. The entry is built, with its UX, SX, UR, SR, UW, and SW bits off, and the index and effective page number of the entry retained by software. The first attempt of application code to use the page will cause an Access Control exception (because the entry is marked “No Execute”, “No Read”, and “No Write”). The Instruction or Data Storage interrupt handler records the reference to the TLB entry and to the associated physical page in a software table, and then turns on the appropriate access control bit. An initial read from the page could be handled by only turning on the appropriate UR or SR access control bits, leaving the page “read-only”.

In a demand-paged environment, when the contents of a physical page are to be replaced, if any storage in that physical page has been altered, then the backing storage must be updated. The information that a physical page is dirty is typically recorded in a “Change” bit for that page.

Write Access Control exceptions may be used to allow software to maintain change information for a physical page. For the example just given for reference recording, the first write access to the page via the TLB entry will create a Write Access Control exception type Data Storage interrupt. The Data Storage interrupt handler records the change status to the physical page in a software table, and then turns on the appropriate UW and SW bits.

6.9 Logical to Real Address Translation [Category: Embedded.Hypervisor.LRAT]

In a partitioned environment, a guest operating system is not allowed to manipulate real page numbers. Instead the hypervisor virtualizes the real memory and the guest operating system manages the virtualized memory using logical page numbers (LPNs). In MMU Architecture Version 2.0, a Logical to Real Address Translation (LRAT) array facilitates this virtualization by providing a hardware translation from an LPN to an RPN without trapping to the hypervisor for every TLB update.

LRAT Entry

Below are shown the field definitions for an LRAT entry.

Name **Description**

V **Valid**
This bit indicates that this LRAT entry is valid and may be used for translation of an LPN to an RPN. The Valid bit for a given entry can be set or cleared with a ***tlbwe*** instruction.

LPID **Logical Partition ID**
This optional field identifies a partition. The Logical Partition ID is compared with LPIDR contents during an LRAT translation. This field is required if category E.PT is supported or if threads that share an LRAT can be in different partitions. Whether the LPID field is supported is indicated by $LRATCFG_{LPID}$.
Note: The number of bits implemented for this field is required to be the same as the TLPID field in a TLB.

LPN **Logical Page Number** (up to 54 bits)
Bits 64-*q*:*n*–1 of the LPN field are compared to bits 64-*q*:*n*–1 of the Logical Page Number (LPN) for the ***tlbwe*** instruction or Page Table translation (where $q = LRATCFG_{LASIZE}$ and $n = 64 - \log_2$(*logical page size* in bytes) and *logical page size* is specified by the LSIZE field of the LRAT entry). See Section 6.7.2. Software must set unused low-order LPN bits to 0.
Note: Bits X:Y of the LPN field are implemented, where $X \geq 0$ and $Y \leq 53$. The bits implemented for LPN are not required to be the same as those implemented for TLB_{RPN}. Unimplemented LRPN bits are treated as if they contain 0s.

LSIZE **Logical Page Size**
The LSIZE field specifies the size of the logical page associated with the LRAT entry as 2^{LSIZE}KB, where $0 \leq LSIZE \leq 31$. Implementations may support any one or more of these logical page sizes (see Section 6.10.3.6), and these logical page sizes need not be the same as the real page sizes that are implemented. However, the smallest logical page is no smaller than the smallest real page. The encodes for LSIZE are the same as the encodes for the TLB SIZE. See Section 6.7.2. This field must be one of the logical pages sizes specified by the LRATPS register.

LRPN **LRAT Real Page Number** (up to 54 bits)
Bits 0:*n*–1 of the LRPN field are used to replace bits 0:*n*–1 of the LPN to produce the RPN that is written to TLB_{RPN} by a ***tlbwe*** instruction or a Page Table translation (where n=64–$\log_2$(*logical page size* in bytes) and *logical page size* is specified by the LSIZE field of the LRAT entry). Software must set unused low-order LRPN bits to 0.
Note: Bits X:Y of the LRPN field are implemented, where $X \geq 0$ and $Y \leq 53$. X = 64 - $MMUCFG_{RASIZE}$. Y = p - 1 where p = 64–$\log_2$(*smallest logical page size* in bytes) and *smallest logical page size* is the smallest page size supported by the implementation as specified by the LRATPS register. Unimplemented LRPN bits are treated as if they contain 0s.

An LRAT entry can be written by the hypervisor using the ***tlbwe*** instruction with $MAS0_{ATSEL}$ equal to 1. The contents of the LRAT entry specified by $MAS0_{ESEL}$, and $MAS2_{EPN}$ are written from MAS registers. See the ***tlbwe*** instruction description in Section 6.11.4.9.

An LRAT entry can be read by the hypervisor using the ***tlbre*** instruction with $MAS0_{ATSEL}$ equal to 1. The contents of the LRAT entry specified by $MAS0_{ESEL}$ and $MAS2_{EPN}$ are read and placed into the MAS registers. See the ***tlbre*** instruction description in Section 6.11.4.9.

Maintenance of LRAT entries is under hypervisor software control. Hypervisor software determines LRAT entry replacement strategy. There is no Next Victim support for the LRAT array.

The LRAT array is a hypervisor resource.

There is at most one LRAT array per thread.

Programming Note

Hypervisor software should not create an LRAT entry that maps any real memory regions for which a TLB entry should have VF equal to 1. Otherwise, a guest operating system could incorrectly create TLB entries, for this memory, with VF=0, assuming hypervisor software normally sets $MAS8_{VF}$=0 before giving control to a guest operating system.

TLB Write

When the guest operating system manipulates the values of RPN fields of MAS registers, the values are treated as forming an LPN. When guest supervisor software attempts to execute ***tlbwe*** on an implementation that supports MMU Architecture Version 1, ***tlbre***, and ***tlbsx***, which operate on a TLB entry's real page number (RPN) or when guest supervisor software attempts to execute a *TLB Management* instruction with guest execution of *TLB Management* instructions disabled ($EPCR_{DGTMI}$=1), an Embedded Hypervisor Privilege exception occurs. Also, if $TLBnCFG_{GTWE}$ = 0 for a TLB array and the guest supervisor executes a ***tlbwe*** to the TLB array, an Embedded Hypervisor Privilege exception occurs. If a ***tlbwe*** caused the exception, the hypervisor can replace the LPN value in the MAS registers with the corresponding RPN, execute a ***tlbwe***, and restore the LPN in the MAS registers before returning to the guest operating system. If a ***tlbre*** or ***tlbsx*** caused the exception, the hypervisor can execute the exception-causing instruction and replace the RPN value in the MAS registers with the corresponding LPN before returning to the guest operating system.

A Logical to Real Address Translation (LRAT) array provides a mechanism that allows a guest operating system to write the TLB without trapping to the hypervisor. When guest supervisor software executes ***tlbwe*** on an implementation that supports MMU Architecture Version 2, guest execution of *TLB Management* instructions is enabled ($EPCR_{DGTMI}$=0), and $TLBnCFG_{GTWE}$ = 1 for the TLB array to be written, an LPN is formed. If MAS7 is implemented, LPN = $MAS7_{RPNU}$ || $MAS3_{RPNL}$. Otherwise, LPN = $^{32}0$ || $MAS3_{RPNL}$. The LPN is translated into an RPN by the LRAT if a matching LRAT entry is found. A matching LRAT entry exists if the following conditions are all true for some LRAT entry.

- The Valid bit of the LRAT entry is one.
- Either the LPID field is not supported in the LRAT ($LRATCFG_{LPID}$=0) or the value of $LPIDR_{LPID}$ is equal to the value of the LPID field of the LRAT entry.
- Bits 64-q:n-1 of the LPN match the corresponding bits of the LPN field of the LRAT entry where n = 64 – $\log_2$(*logical page size* in bytes), *logical page size* is specified by the LSIZE field of the LRAT entry, and q is specified by $LRATCFG_{LASIZE}$.
- Either of the following is true.
  - $MAS1_{IND}$ = 0 and the value of $MAS1_{TSIZE}$ is less than or equal to the value of the LSIZE field of the LRAT entry.
  - $MAS1_{IND}$ = 1 and the value of (3 + ($MAS1_{TSIZE}$ – $MAS3_{SPSIZE}$)) is less than or equal to the value of the (10 + LRAT $entry_{LSIZE}$).

If a matching LRAT entry is found, the LRPN from that LRAT entry provides the upper bits of the RPN that is written to the TLB, and the LPN provides the low order RPN bits written to the TLB. Let n=64–$\log_2$(*logical page size* in bytes) where *logical page size* is specified by the LSIZE field of the LRAT entry. Bits n:53 of the LPN are appended to bits 0:n–1 of the LRPN field of the selected LRAT entry to produce the RPN (i.e. RPN = $LRPN_{0:n-1}$ || $LPN_{n:53}$). The page size specified by the LSIZE of the LRAT entry used to translate the LPN must be one of the values supported by the implementation's LRAT array. If the LRAT does not contain a matching entry for the LPN, an LRAT Miss exception occurs.

When the hypervisor executes a ***tlbwe*** instruction, no LRAT translation is performed and the RPN formed from $MAS7_{RPNU}$ and $MAS3_{RPNL}$ is written to the TLB.

Page Table

A Logical to Real Address Translation (LRAT) array provides a mechanism that allows guest Page Table management and translation without direct hypervisor involvement. When an instruction fetch address or a *Load*, *Store*, or *Cache Management* instruction operand address is translated by the Page Table, the Embedded.Hypervisor category is supported, and the TGS bit of the associated indirect TLB entry is 1, the RPN result of the Page Table translation is treated as an LPN that is translated into an RPN by the LRAT if a matching LRAT entry is found. A matching LRAT entry exists if the following conditions are all true for some LRAT entry.

- The Valid bit of the LRAT entry is one.
- Either the LPID field is not supported in the LRAT ($LRATCFG_{LPID}$=0) or the value of $LPIDR_{LPID}$ is equal to the value of the LPID field of the LRAT entry.
- Bits 64-q:n-1 of the LPN match the corresponding bits of the LPN field of the LRAT entry where n = 64 – $\log_2$(*logical page size* in bytes), *logical page size* is specified by the LSIZE field of the LRAT entry, and q is specified by $LRATCFG_{LASIZE}$.
- The value of PTE_{PS} is less than or equal to the value of the LSIZE field of the LRAT entry.

If a matching LRAT entry is found, the LRPN from that LRAT entry provides the upper bits of the RPN of the translation result and the LPN provides the low order RPN bits of the translation result. Let n=64–$\log_2$(*logical page size* in bytes) where *logical page size* is specified by the LSIZE field of the LRAT entry. Bits

n:51 of the LPN are appended to bits 0:*n*–1 of the LRPN field of the selected LRAT entry to produce the RPN (i.e. RPN = $LRPN_{0:n-1}$ || $LPN_{n:51}$). The page size specified by the LSIZE of the LRAT entry used to translate the LPN must be one of the values supported by the implementation's LRAT array. If the LRAT does not contain a matching entry for the LPN, an LRAT Miss exception occurs.

6.10 Storage Control Registers

In addition to the registers described below, the Machine State Register provides the IS and DS bits, that specify which of the two address spaces the respective instruction or data storage accesses are directed towards. MSR_{PR} bit is also used by the storage access control mechanism. If the Embedded.Hypervisor category is supported, the MSR_{GS} bit is used to identify guest state. The guest supervisor state exists when MSR_{PR} = 0 and MSR_{GS} = 1. MSR_{GS} is used to form the virtual address. Also, see Section 5.3.7 for the registers in the Embedded.External PID category.

6.10.1 Process ID Register

The Process ID Registers are 32-bit registers as shown in Figure 27. Process ID Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The number of bits implemented in a PID register is indicated by the value of the $MMUCFG_{PIDSIZE}$. The Process ID Register provides a value that is used to construct a virtual address for accessing storage.

The Process ID Register is a privileged register. This register can be read using ***mfspr*** and can be written using ***mtspr***. An implementation may opt to implement only the least-significant *n* bits of the Process ID Register, where $1 \le n \le 14$, and *n* must be the same as the number of implemented bits in the TID field of the TLB entry. The most-significant 32–*n* bits of the Process ID Register are treated as reserved.

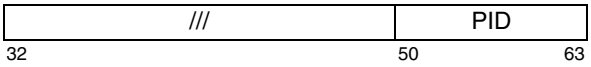


Figure 27. Processor ID Register (PID)

The PID register fields are described below.

Bit	Description
50:63	***Processor ID*** (PID) Identifies a unique process (except for the value of 0) and is used to construct a virtual address for storage accesses.

All other fields are reserved.

Programming Note

The PID register was referred to as PID0 in the Type FSL Storage Control appendix of previous versions of the architecture.

6.10.2 MMU Assist Registers

The MMU Assist Registers (MAS) are used to transfer data to and from the TLB arrays. If the Embedded.Hypervisor.LRAT category is supported, MAS registers are also used to transfer data to and from the LRAT array. MAS registers can be read and written by software using ***mfspr*** and ***mtspr*** instructions. Execution of a ***tlbre*** instruction with $MAS0_{ATSEL}$=0 causes the TLB entry specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$, and $MAS2_{EPN}$ to be copied to the MAS registers if $TLBnCFG_{HES}$ = 0 for the TLB array specified by $MAS0_{TLBSEL}$ whereas the TLB entry is specified by $MAS0_{ESEL}$, $MAS0_{TLBSEL}$, and a hardware generated hash based on $MAS2_{EPN}$, $MAS1_{TID}$, and $MAS1_{TSIZE}$ if $TLBnCFG_{HES}$=1. Execution of a ***tlbwe*** instruction with $MAS0_{ATSEL}$=0 (or in guest supervisor state) causes the TLB entry specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$, and $MAS2_{EPN}$ to be written with contents of the MAS registers if $TLBnCFG_{HES}$ = 0 for the TLB array specified by $MAS0_{TLBSEL}$. If $TLBnCFG_{HES}$ = 1 for a ***tlbwe***, the TLB entry is selected by $MAS0_{TLBSEL}$, a hardware generated hash based on $MAS2_{EPN}$, $MAS1_{TID}$, and $MAS1_{TSIZE}$, and either a hardware replacement algorithm if $MAS0_{HES}$=1 or $MAS0_{ESEL}$ if $MAS0_{HES}$=0. MAS registers may also be updated by hardware as the result of any of the following.

- a ***tlbsx*** instruction
- the occurrence of an Instruction or Data TLB Error interrupt if any of the following is true.
  - The Embedded.Hypervisor category is not supported.
  - MAS Register updates are enabled for interrupts directed to the hypervisor ($EPCR_{DMIUH}$ = 0).
  - The interrupt is directed to the guest state ($EPCR_{ITLBGS}$ = 1 for Instruction TLB Error interrupt and $EPCR_{DTLBGS}$ = 1 for Data TLB Error interrupt).

All MAS registers are privileged, except MAS5 and MAS8, which are hypervisor privileged and are only provided if Category: Embedded.Hypervisor is supported. All MAS registers with the exception of MAS7 and, if Embedded.Hypervisor category is not supported, MAS5 and MAS8, must be implemented. MAS7 is not required to be implemented if the hardware supports 32 bits or less of real address.

The necessary bits of any multi-bit field in a MAS register must be implemented such that only the resources supported are represented. Any non-implemented bits in a field should have no effect when writing and should always read as zero. For example, if only 2 TLB arrays

are implemented, then only the lower-order bit of the $MAS0_{TLBSEL}$ field is implemented.

Programming Note

Operating system developers should be wary of new implementations silently ignoring unimplemented MAS bits on MAS register writes. This is a common error during initial bring-up of a new processor.

6.10.3 MMU Configuration and Control Registers

6.10.3.1 MMU Configuration Register (MMUCFG)

The read-only MMUCFG register provides information about the MMU and its arrays. MMUCFG is a privileged register except that if the Embedded.Hypervisor category is supported, MMUCFG is a hypervisor resource. The layout of the MMUCFG register is shown in Figure 28 for MAV=1.0 and in Figure 29 for MAV=2.0.

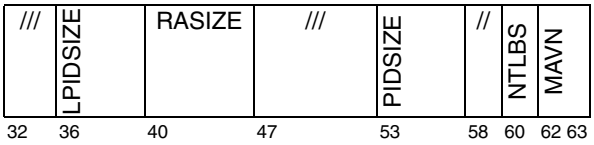


Figure 28. MMU Configuration Register [MAV=1.0]

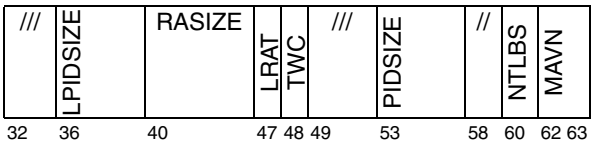


Figure 29. MMU Configuration Register [MAV=2.0]

The MMUCFG fields are described below.

Bit **Description**

36:39 ***LPID Register Size*** (LPIDSIZE)
The value of LPIDSIZE is the number of bits in LPIDR that are implemented. Only the least significant LPIDSIZE bits in LPIDR are implemented. The Embedded.Hypervisor category is supported if and only if LPIDSIZE > 0.

40:46 ***Real Address Size*** (RASIZE)
Number of bits in a real address supported by the implementation.

47 ***LRAT Translation Supported*** (LRAT) [Category: Embedded.Hypervisor.LRAT]
Indicates LRAT translation is supported.

- 0 LRAT translation is not supported. A ***tlbwe*** executed in guest supervisor state results in an Embedded Hypervisor Privilege exception.
- 1 LRAT translation is supported by one or more TLB arrays. See $TLBnCFG_{GTWE}$. The LRATCFG and LRATPS registers are supported.

48 ***TLB Write Conditional*** (TWC) [MAV=2.0]
Indicates whether the Embedded.TLB Write Conditional category is supported.

- 0 E.TWC category is not supported
- 1 E.TWC category is supported. See Section 6.11.4.2.1 for a description of conditional TLB writes. This category also includes support for the ***tlbsrx.*** instruction, $MAS0_{WQ}$, and $MAS6_{ISIZE}$.

53:57 ***PID Register Size*** (PIDSIZE)
The value of PIDSIZE is one less than the number of bits implemented for each of the PID registers implemented. Only the least significant PIDSIZE+1 bits in the PID registers are implemented. The maximum number of PID register bits that may be implemented is 14.

60:61 ***Number of TLBs*** (NTLBS)
The value of NTLBS is one less than the number of software-accessible TLB structures that are implemented. NTLBS is set to one less than the number of TLB structures so that its value matches the maximum value of $MAS0_{TLBSEL}$.

- 00 1 TLB
- 01 2 TLBs
- 10 3 TLBs
- 11 4 TLBs

62:63 ***MMU Architecture Version Number*** (MAVN)
Indicates the version number of the architecture of the MMU implemented.

- 00 Version 1.0
- 01 Version 2.0
- 10 Reserved
- 11 Reserved

All other fields are reserved.

6.10.3.2 TLB Configuration Registers (TLB*n*CFG)

Each TLB*n*CFG read-only register provides configuration information about each corresponding TLB array that is implemented. There is one TLB*n*CFG register implemented for each TLB array that is implemented. TLB*n*CFG corresponds to TLBn for $0 \le n \le MMUCFG_{NTLBS}$. TLB*n*CFG registers are privileged registers except that if the Embedded.Hypervisor category is supported, TLB*n*CFG registers are hypervisor resources. The layout of the TLB*n*CFG registers is

shown in Figure 30 for MAV=1.0 and in Figure 31 for MAV=2.0.

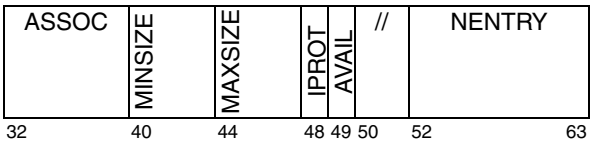


Figure 30. TLB Configuration Register [MAV=1.0]

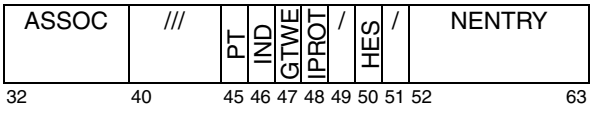


Figure 31. TLB Configuration Register [MAV=2.0]

The TLB*n*CFG fields are described below.

Bit **Description**

32:39 ***Associativity*** (ASSOC)
Total number of entries in a TLB array which can be used for translating addresses with a given EPN. This number is referred to as the associativity level of the TLB array. Some values of assoc have special meanings when used in combination with specific values of NENTRY, as follows. However, if TLBnCFGHES=1, the associativity level of the TLB array is implementation dependent.

NENTRY	ASSOC	Meaning
0	0	no TLB present
0	1	TLB geometry is completely implementation-defined. $MAS0_{ESEL}$ is ignored
0	>1	TLB geometry and number of entries is implementation defined, but has known associativity. For ***tlbre*** and ***tlbwe***, a set of TLB entries is selected by an implementation dependent function of $MAS8_{TGS\ TLPID}$ <E.HV>, $MAS1_{TS\ TID\ TSIZE}$, and $MAS2_{EPN}$. $MAS0_{ESEL}$ is used to select among entries in this set, except on ***tlbwe*** if $MAS0_{HES}$=1.
n > 0	n or 0	TLB is fully associative

40:43 ***Minimum Page Size*** (MINSIZE) [MAV=1.0]
This field defines the minimum page size of the TLB array. Page size encoding is defined in Section 6.7.2.

44:47 ***Maximum Page Size*** (MAXSIZE) [MAV=1.0]
This field defines the maximum page size of the TLB array. Page size encoding is defined in Section 6.7.2.

45 ***Page Table*** (PT) [MAV=2.0 and Category: E.PT]
Indicates that the TLB array can be loaded from the hardware Page Table.

0 TLB array cannot be loaded from the Page Table.
1 TLB array can be loaded from the Page Table.

46 ***Indirect*** (IND) [MAV=2.0 and Category: E.PT]
Indicates that an indirect TLB entry can be created in the TLB array and that there is a corresponding EPTCFG register that defines the SIZE and Sub-Page Size values that are supported.

0 The TLB array treats the IND bit as reserved.
1 The TLB array supports indirect entries.

47 ***Guest TLB Write Enabled*** (GTWE) [MAV=2.0 and Category: Embedded.Hypervisor.LRAT]
Indicates that a guest supervisor can write the TLB array because LRAT translation is supported for the TLB array.

0 A guest supervisor cannot write the TLB array. A ***tlbwe*** executed in guest supervisor state results in an Embedded Hypervisor Privilege exception.
1 A guest supervisor can write the TLB array if guest execution of *TLB Management* instructions is enabled ($EPCR_{DGTMI}$=0).

48 ***Invalidate Protection*** (IPROT)
Invalidate protect capability of the TLB array.

0 Indicates invalidate protection capability not supported.
1 Indicates invalidate protection capability supported.

49 ***Page Size Availability*** (AVAIL) [MAV=1.0]
This defines the page size availability of the TLB array. If the Embedded.Page Table category is supported, this also defines the virtual address space size availability of TLB array. Otherwise, this field is reserved.

0 Fixed selectable page size from MINSIZE to MAXSIZE (all TLB entries are the same size).
1 Variable page size from MINSIZE to MAXSIZE (each TLB entry can be sized separately).

50 ***Hardware Entry Select*** (HES) [MAV=2.0]
Indicates whether the TLB array supports $MAS0_{HES}$ and the associated method for hardware selecting a TLB entry based on $MAS1_{TID\ TSIZE}$ and $MAS2_{EPN}$ for a ***tlbwe*** instruction.

0 $MAS0_{HES}$ is not supported.
1 $MAS0_{HES}$ is supported for ***tlbwe***. See Section 6.10.3.8. For ***tlbre***, $MAS0_{ESEL}$ selects among the TLB entries that can be used for translating addresses with a given $MAS1_{TID\ TSIZE}$ and $MAS2_{EPN}$. The set of TLB entries is determined by a hardware generated hash based on $MAS1_{TID\ TSIZE}$ and $MAS2_{EPN}$. The hash is the same for ***tlbwe*** and ***tlbre*** for a given TLB array but could be different for each TLB array.

52:63 ***Number of Entries*** (NENTRY)
Number of entries in the TLB array.

All other fields are reserved.

6.10.3.3 TLB Page Size Registers (TLB*n*PS) [MAV=2.0]

Each TLB*n*PS read-only register provides page size information about each corresponding TLB array that is implemented in MMU Architecture Version 2.0. Each Page Size bit (PS0-PS31) that is a one indicates that a specific page size is supported by the array. Multiple 1 bits indicate that multiple page sizes are supported concurrently. TLB*n*PS registers are privileged registers except that if the Embedded.Hypervisor category is supported, TLB*n*PS registers are hypervisor resources. The layout of the TLB*n*PS registers is shown in Figure 32.

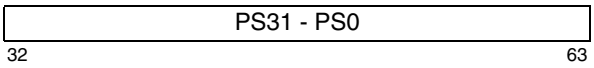


Figure 32. TLB n Page Size Register

The TLB*n*PS fields are described below.

Bit **Description**

32:63 ***Page Size 31 - Page Size 0*** (PS31-PS0)
PSm indicates whether a direct TLB entry page size of 2^m KB is supported by the TLB array. PSm corresponds to bit $TLBnPS_{63-m}$ for m = 0 to 31.

0 Direct TLB entry page size of 2^m KB is not supported.
1 Direct TLB entry page size of 2^m KB is supported.

Table 9 shows the relationship between the Page Size (PSm) bits in TLB*n*PS and page size. The existence and type of mechanism for configuring the use of a subset of supported page sizes is implementation-dependent.

Table 9: Relationship of TLB*n*PS PS bits and LRATPS PS bits to page size

TLBnPS or LRATPS bit	PSm	Page Size
32	PS31	2TB
33	PS30	1TB
34	PS29	512GB
35	PS28	256GB
36	PS27	128GB
37	PS26	64GB
38	PS25	32GB
39	PS24	16GB
40	PS23	8GB
41	PS22	4GB
42	PS21	2GB
43	PS20	1GB
44	PS19	512MB
45	PS18	256MB
46	PS17	128MB
47	PS16	64MB
48	PS15	32MB
49	PS14	16MB
50	PS13	8mb
51	PS12	4MB
52	PS11	2MB
53	PS10	1MB
54	PS9	512KB
55	PS8	256KB
56	PS7	128KB
57	PS6	64KB
58	PS5	32KB
59	PS4	16KB
60	PS3	8KB
61	PS2	4KB
62	PS1	2KB
63	PS0	1KB

6.10.3.4 Embedded Page Table Configuration Register (EPTCFG)

This read-only register consists of 3 pairs of page size (PSi) and sub-page size (SPSi) values, where i = 0 to 2. These combinations are supported for Page Table translations. The page size and sub-page size encodings for PSi and SPSi are the same the $MAS1_{TSIZE}$ encodings, except that an SPSi value of 0b00001 is reserved and a value of zero for the SPSi field means there is no page size and sub-page size combination information supplied by that field. If SPSi is zero, PSi is zero. Zero values of PSi and SPSi pairs are the leftmost fields. See Table 3. For nonzero values of SPSi, PSi minus SPSi is greater than 7.

The EPTCFG register is a privileged register except that if the Embedded.Hypervisor category is supported,

EPTCFG register is a hypervisor resource. The layout of the EPTCFG register is shown in Figure 33.

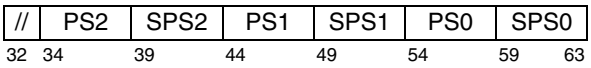


Figure 33. Embedded Page Table Configuration Register

The EPTCFG fields are described below.

Bit	Description
34:38	***Page Size 2*** (PS2) PS2 indicates whether an indirect TLB entry with a page size of 2^{PS2} KB combined with the sub-page size specified by SPS2 is supported.
39:43	***Sub-Page Size 2*** (SPS2) SPS2 indicates whether an indirect TLB entry with a sub-page size of 2^{SPS2} KB combined with the page size specified by PS2 is supported.
44:48	***Page Size 1*** (PS1) PS1 indicates whether an indirect TLB entry with a page size of 2^{PS1} KB combined with the sub-page size specified by SPS1 is supported.
49:53	***Sub-Page Size1*** (SPS1) SPS1 indicates whether an indirect TLB entry with a sub-page size of 2^{SPS1} KB combined with the page size specified by PS1 is supported.
54:58	***Page Size 0*** (PS0) PS0 indicates whether an indirect TLB entry with a page size of 2^{PS0} KB combined with the sub-page size specified by SPS0 is supported.
59:63	***Sub-Page Size 0*** (SPS0) SPS0 indicates whether an indirect TLB entry with a sub-page size of 2^{SPS0} KB combined with the page size specified by PS0 is supported.

6.10.3.5 LRAT Configuration Register (LRATCFG) [Category: Embedded.Hypervisor.LRAT]

The LRATCFG read-only register provides configuration information about the LRAT array. LRATCFG is a hypervisor resource. The layout of the LRATCFG registers is shown in Figure 34.

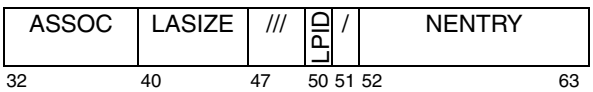


Figure 34. LRAT Configuration Register

The LRATCFG fields are described below.

Bit	Description
32:39	***Associativity*** (ASSOC) Total number of entries in the LRAT array which can be used for translating addresses with a given LPN. This number is referred to as the associativity level of the LRAT array. A value equal to NENTRY or 0 indicates the array is fully-associative.
40:46	***Logical Address Size*** (LASIZE) Number of bits in a logical address supported by the implementation.
50	***LPID Supported*** (LPID) Indicates whether the LPID field in the LRAT is supported. 0 The LPID field in the LRAT is not supported. 1 The LPID field in the LRAT is supported.
52:63	***Number of Entries*** (NENTRY) Number of entries in LRAT array. At least one entry is supported.

All other fields are reserved.

6.10.3.6 LRAT Page Size Register (LRATPS) [Category: Embedded.Hypervisor.LRAT]

LRATPS is a read-only register that provides page size information about the LRAT that is implemented if the Embedded.Hypervisor.LRAT category is supported in MMU Architecture Version 2.0. Each Page Size bit (PS0-PS31) that is a one indicates that a specific logical page size is supported by the array. Multiple 1 bits indicate that multiple page sizes are supported concurrently. LRATPS is a hypervisor resource. The layout of the LRATPS registers is shown in Figure 35.

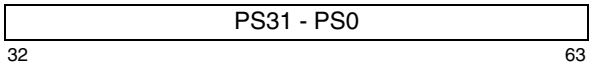


Figure 35. LRAT Page Size Register

The LRATPS fields are described below.

Bit	Description
32:63	***Page Size 31 - Page Size 0*** (PS31-PS0) PSm indicates whether a logical page size of 2^m KB is supported by the LRAT array. PSm corresponds to bit $LRATPS_{64-m}$ for m = 0 to 31. 0 Logical page size of 2^m KB is not supported. 1 Logical page size of 2^m KB is supported.

All other fields are reserved.

Table 9 on page 1106 shows the relationship between the Page Size (PSm) bits in LRATPS and the logical page size.

6.10.3.7 MMU Control and Status Register (MMUCSR0)

The MMUCSR0 register is used for general control of the MMU including page sizes for programmable fixed size TLB arrays [MAV=1.0] and invalidation of the TLB array. For TLB arrays that have programmable fixed sizes, the TLBn_PS fields [MAV=1.0] allow software to specify the page size. MMUCSR0 is a privileged register except that if the Embedded.Hypervisor category is supported, MMUCSR0 is a hypervisor resource.

The layout of the MMUCSR0 is shown in Figure 36 for MAV=1.0 and in Figure 37 for MAV=2.0.

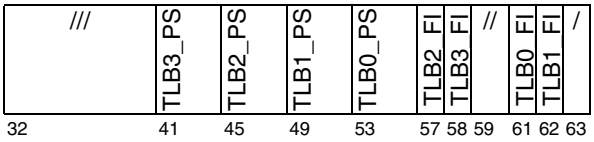


Figure 36. MMU Control and Status Register 0 [MAV=1.0]

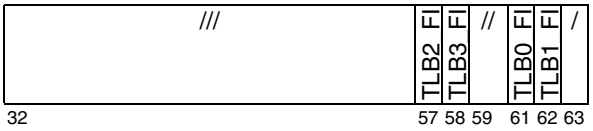


Figure 37. MMU Control and Status Register 0 [MAV=2.0]

The MMUCSR0 fields are described below.

Bit Description

41:56 ***TLBn Array Page Size*** [MAV=1.0]
A 4-bit field specifies the page size for TLB*n* array. Page size encoding is defined in Section 6.7.2. If the value of TLBn_PS is not between $TLBnCFG_{MINSIZE}$ and $TLBnCFG_{MAXSIZE}$, the page size is set to $TLBnCFG_{MINSIZE}$. A TLBn_PS field is implemented only for a TLB array that can be programmed to support only one of several fixed page sizes. For each TLB array *n* (for $0 \le n < MMUCFG_{NTLBS}$), this field is implemented only if the following are all true.

- $TLBnCFG_{AVAIL} = 0$.
- $TLBnCFG_{MINSIZE} \neq TLBnCFG_{MAXSIZE}$.

Bit Description

41:44 ***TLB3 Array Page Size*** (TLB3_PS)
Page size of the TLB3 array.

45:48 ***TLB2 Array Page Size*** (TLB2_PS)
Page size of the TLB2 array.

49:52 ***TLB1 Array Page Size*** (TLB1_PS)
Page size of the TLB1 array.

53:56 ***TLB0 Array Page Size*** (TLB0_PS)
Page size of the TLB0 array.

Programming Note

Changing the fixed page size of an entire array must be done with great care. If any entries in the array are valid, changing the page size may cause those entries to overlap, creating a serious programming error. It is suggested that the entire TLB array be invalidated and any entries with IPROT have their V bits set to zero before changing page size.

57:62 ***TLBn Invalidate All***
TLB invalidate all bit for the TLB*n* array.

0 If this bit reads as a 1, an invalidate all operation for the TLB*n* array is in progress. Hardware will set this bit to 0 when the invalidate all operation is completed. Writing a 0 to this bit during an invalidate all operation is ignored.

1 TLB*n* invalidation operation. Hardware initiates a TLB*n* invalidate all operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidate all operation produces an undefined result. If the TLB array supports IPROT, entries that have IPROT set will not be invalidated.

57 ***TLB2 Invalidate All*** (TLB2_FI)
TLB invalidate all bit for the TLB2 array.

58 ***TLB3 Invalidate All*** (TLB3_FI)
TLB invalidate all bit for the TLB3 array.

61 ***TLB0 Invalidate All*** (TLB0_FI)
TLB invalidate all bit for the TLB0 array.

62 ***TLB1 Invalidate All*** (TLB1_FI)
TLB invalidate all bit for the TLB1 array.

All other fields are reserved.

6.10.3.8 MAS0 Register

The MAS0 register contains fields for identifying and selecting a TLB entry. If the Embedded.Hypervisor.LRAT category is supported, the MAS0 register is also used to select an LRAT entry as well as select between a TLB array and the LRAT array. MAS0 register fields are loaded by the execution of the ***tlbsx*** instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions.

MAS0 is a privileged register. The layout of the MAS0 register is shown in Figure 38.

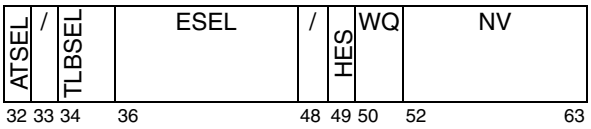


Figure 38. MAS0 register

The MAS0 fields are described below.

Bit **Description**

32 ***Array Type Select*** (ATSEL) [Category: Embedded.Hypervisor.LRAT]
Selects LRAT or TLB for access for ***tlbwe*** and ***tlbre***. In guest state, $MAS0_{ATSEL}$ is treated as if it were zero such that a TLB array is always selected.

0 TLB
1 LRAT

34:35 ***TLB Select*** (TLBSEL)
If ATSEL=0 or MSR_{GS}=1, selects TLB for access.
If ATSEL=1, TLBSEL is treated as reserved.

00 TLB0
01 TLB1
10 TLB2
11 TLB3

36:47 ***Entry Select*** (ESEL)
Identifies an entry in the selected array to be used for ***tlbwe*** and ***tlbre***. Valid values for ESEL are from 0 to TLB*n*CFG_{ASSOC} - 1 for a TLB array and 0 to $LRATCFG_{ASSOC}$ - 1 for the LRAT. That is, ESEL selects the entry in the selected array from the set of entries which can be used for translating addresses with the EPN (if $TLBnCFG_{HES}$=0 and either $MAS0_{ATSEL}$=0 or MSR_{GS}=1), the combination of EPN, SIZE, and PID (for ***tlbwe*** if $TLBnCFG_{HES}$=1, $MAS0_{HES}$=0, and either $MAS0_{ATSEL}$=0 or MSR_{GS}=1, and for ***tlbre*** if $TLBnCFG_{HES}$=1, and either $MAS0_{ATSEL}$=0 or MSR_{GS}=1), or the LPN (if $MAS0_{ATSEL}$=1 and MSR_{GS}=0) specified by $MAS2_{EPN}$. For fully-associative TLB or LRAT arrays, ESEL ranges from 0 to TLB*n*CFG_{NENTRY} - 1 or 0 to $LRATCFG_{NENTRY}$ - 1, respectively.

49 ***Hardware Entry Select*** (HES) [MAV=2.0]
Determines how the TLB entry within the selected TLB array is selected by ***tlbwe*** if a TLB entry is to be written ($MAS0_{ATSEL}$=0 or MSR_{GS}=1). If an LRAT entry is to be written ($MAS0_{ATSEL}$=1 and MSR_{GS}=0) by a ***tlbwe***, HES must be 0. Otherwise the result is undefined. This field has no effect on other *TLB Management* instructions. Whether an implementation supports this bit for a TLB array is indicated by $TLBnCFG_{HES}$. If $TLBnCFG_{HES}$ = 0, HES is ignored and treated as 0 for ***tlbwe***.

0 The entry is selected by $MAS0_{ESEL}$ and a hardware generated hash based on $MAS1_{TID\ TSIZE}$, and $MAS2_{EPN}$.
1 The entry is selected by a hardware replacement algorithm and a hardware generated hash based on $MAS1_{TID\ TSIZE}$, and $MAS2_{EPN}$.

50:51 ***Write Qualifier*** (WQ) [MAV=2.0 and Category: Embedded.TLB Write Conditional]
Qualifies the TLB write operation performed by ***tlbwe*** if a TLB entry is to be written ($MAS0_{ATSEL}$=0 or MSR_{GS}=1). If an LRAT entry is to be written ($MAS0_{ATSEL}$=1 and MSR_{GS}=0) by a ***tlbwe*** and the Embedded.TLB Write Conditional category is supported, WQ must be 0b00. Otherwise the result is undefined. This field has no effect on other *TLB Management* instructions. Whether an implementation supports this field is indicated by $MMUCFG_{TWC}$. If $MMUCFG_{TWC}$ = 0, WQ is ignored and treated as 0b00 for ***tlbwe***.

00 The selected TLB entry is written regardless of the TLB-reservation. The TLB-reservation is cleared.
01 The selected TLB entry is written if and only if the TLB reservation exists. A tlbwe with this value is called a *TLB Write Conditional*. The TLB-reservation is cleared. See Section 6.11.4.2.1, “TLB Write Conditional [Embedded.TLB Write Conditional]”.
10 The TLB-reservation is cleared; no TLB entry is written.
11 Reserved

52:63 ***Next Victim*** (NV)
NV is a hint to software to identify the next victim to be targeted for a TLB miss replacement operation for those TLBs that support the NV function. If the TLB selected by $MAS0_{TLBSEL}$ does not support the NV function, this field is undefined. The method of determining the next victim is implementation-dependent. NV is updated on ***tlbsx*** hit and miss cases as shown in Table 11 on page 1116, on execution of ***tlbre*** if the TLB array being accessed supports the NV field, and on TLB Error interrupts if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor ($EPCR_{DMIUH}$ = 0), or the interrupt is directed to the guest state. When NV is updated by a supported TLB array, the NV field will always present a value that can be used in the $MAS0_{ESEL}$ field. The LRAT array does not support Next Victim.

All other fields are reserved.

6.10.3.9 MAS1 Register

The MAS1 register contains fields used for reading and writing an LRAT <E.HV.LRAT> or TLB entry. MAS1 register fields are also loaded by the execution of the ***tlbsx*** instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions. TLB fields loaded from the MAS1 register are used for selecting a TLB entry during translation. If the Embedded.Hypervisor.LRAT category is supported, LRAT fields V and LSIZE, which are loaded from $MAS1_{V\ TSIZE}$, are used for selecting an LRAT entry for translating LPNs when ***tlbwe*** is executed in guest supervisor state and, if the Embedded.Page Table category is supported, during page table lookups performed when the PTE_{ARPN} is treated as an LPN (The Embedded.Hypervisor.LRAT category is supported and the TGS bit of the corresponding indirect TLB entry is 1). MAS1 is a privileged register. The layout of the MAS1 register is shown in Figure 39 for MAV=1.0 and in Figure 40 for MAV=2.0.

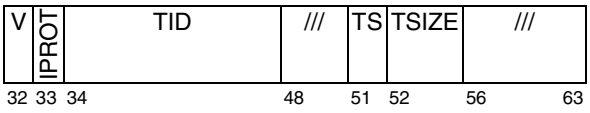


Figure 39. MAS1 register [MAV=1.0]

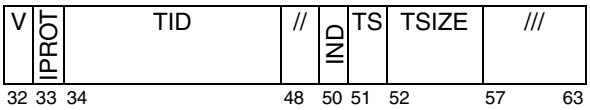


Figure 40. MAS1 register [MAV=2.0]

The MAS1 fields are described below.

Bit **Definition**

32 ***Valid Bit*** (V)
See the corresponding TLB bit definition in Section 6.7.1 and the corresponding LRAT bit definition in Section 6.9.

33 ***Invalidate Protect*** (IPROT)
See the corresponding TLB bit definition in Section 6.7.1.

34:47 ***Translation Identity*** (TID)
See the corresponding TLB field definition in Section 6.7.1.

50 ***Indirect*** (IND) [MAV=2.0 and Category: Embedded.Page Table]
See the corresponding TLB bit definition in Section 6.7.1.

51 ***Translation Space*** (TS)
See the corresponding TLB bit definition in Section 6.7.1.

52:55 ***Translation Size*** (TSIZE) [MAV=1.0]
See the TLB SIZE field definition in Section 6.7.1.

52:56 ***Translation Size*** (TSIZE) [MAV=2.0]
See the TLB SIZE field definition in Section 6.7.1 and the LRAT LSIZE field definition in Section 6.9.

All other fields are reserved.

6.10.3.10 MAS2 Register

The MAS2 register is a 64-bit register which can be read and written as a 64-bit register in 64-bit implementations and as a 32-bit register in 32-bit implementations. In both 64-bit and 32-bit implementations, the MAS2U register can be used to read or write $EPN_{0:31}$ as a 32-bit SPR access. The MAS2 register contains fields used for reading and writing an LRAT <E.HV.LRAT> or TLB entry. MAS2 register fields are also loaded by the execution of the ***tlbsx*** instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions. The register contains fields for specifying the effective page address and the storage control attributes for a TLB entry. If the Embedded.Hypervisor.LRAT category is supported, the MAS2 register EPN field can also be used for specifying the logical page number for an LRAT entry. The only MAS2 field used for the LRAT array is EPN. MAS2 is a privileged register. The layout of the MAS2 register is shown in Figure 41 for MAV=1.0 and in Figure 42 for MAV=2.0.

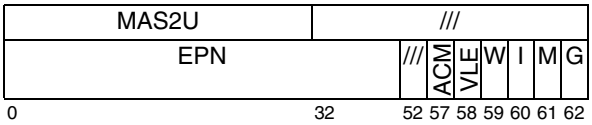


Figure 41. MAS2 register [MAV = 1.0]

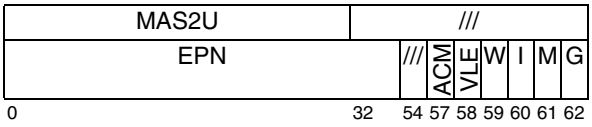


Figure 42. MAS2 register [MAV = 2.0]

The MAS2 fields are described below.

Bit **Description**

0:31 ***MAS2 Upper*** (MAS2U)
MAS2U is an SPR that corresponds to $EPN_{0:31}$ of MAS2.

0:51 ***Effective Page Number*** (EPN) [MAV=1.0]
See the corresponding TLB bit definition in Section 6.7.1. Bits that correspond to an offset within the smallest virtual page implemented need not be implemented. Unimplemented EPN bits are treated as 0s.

0:53 ***Effective Page Number*** (EPN) [MAV=2.0
See the corresponding TLB bit definition in Section 6.7.1 and the LRAT LPN field defini-

tion in Section 6.9. Bits that correspond to an offset within the smallest virtual page implemented need not be implemented. Unimplemented EPN bits are treated as 0s.

57 ***Alternate Coherency Mode*** (ACM)
See the corresponding TLB bit definition in Section 6.7.1. If ACM is not supported by the implementation, this bit is treated as reserved.

Programming Note

Some previous implementations may have a TLB storage bit accessed via this bit position and labeled as X0. Software should not use the presence of this bit (the ability to set to 1 and read a 1) to determine if the implementation supports the Alternate Coherency Mode.

58 ***VLE Mode*** (VLE) [Category: VLE]
See the corresponding TLB bit definition in Section 6.7.1. If the VLE category is not supported, this bit is treated as reserved.

Programming Note

Some previous implementations may have a TLB storage bit accessed via this position and labeled as X1. Software should not use the presence of this bit (the ability to set to 1 and read a 1) to determine if the implementation supports the VLE.

59 ***Write Through*** (W)
See the corresponding TLB bit definition in Section 6.7.1.

60 ***Caching Inhibited*** (I)
See the corresponding TLB bit definition in Section 6.7.1.

61 ***Memory Coherence Required*** (M)
See the corresponding TLB bit definition in Section 6.7.1.

62 ***Guarded*** (G)
See the corresponding TLB bit definition in Section 6.7.1.

63 ***Endianness*** (E)
See the corresponding TLB bit definition in Section 6.7.1.

All other fields are reserved.

6.10.3.11 MAS3 Register

The MAS3 register contains fields used for reading and writing an LRAT <E.HV.LRAT> or TLB entry. MAS3 register fields are also loaded by the execution of the ***tlbsx*** instruction and by the occurrence of an Instruction or Data TLB Error interrupt under certain conditions. The MAS3 register contains fields for specifying the real page address, user defined attributes, and the permission attributes for a TLB entry. if the Embedded.Page Table category is supported, MAS3 also contains a field specifying the minimum page size specified by each Page Table Entry that is mapped by the indirect TLB entry. if the Embedded.Hypervisor.LRAT category is supported, the low-order LRPN bits of the LRAT array can be read into or written from $MAS3_{RPNL}$ by hypervisor software. If the Embedded.Hypervisor.LRAT category is supported, the RPN specified by MAS7 and MAS3 is treated as an LPN for ***tlbwe*** executed in guest supervisor state (see Section 6.9). MAS3 is a privileged register. if the Embedded.Page Table category is supported, MAS3 has different meanings depending on the $MAS1_{IND}$ value. For $MAS1_{IND}$ = 0, the layout of the MAS3 register is shown in Figure 43 for MAV=1.0 and in Figure 44 for MAV=2.0.

Figure 43. MAS3 register for $MAS1_{IND}$=0 [MAV=1.0]

Figure 44. MAS3 register for $MAS1_{IND}$=0 [MAV=2.0]

For $MAS1_{IND}$ = 1, the layout of the MAS3 register is shown in Figure 45.

Figure 45. MAS3 register for $MAS1_{IND}$=1 [MAV=2.0 and Category: E.PT]

The MAS3 fields are described below.

Bit **Description**

32:51 ***Real Page Number (bits 32:51)*** (RPNL or $RPN_{32:51}$) [MAV=1.0]
The real page number is formed by the upper n bits of ($MAS7_{RPNU}$ || $MAS3_{RPNL}$), where n = 64 - $\log_2$(*page size* in bytes) and *page size* is specified by $MAS1_{TSIZE}$ for a ***tlbwe*** instruction and by the SIZE field of the TLB entry if a TLB entry is being read by a ***tlbre*** or ***tlbsx*** instruction. $RPN_{0:31}$ are accessed through MAS7. RPNL bits corresponding to bits that are not implemented in the RPN field of the TLB are treated as reserved.

32:53 ***Real Page Number (bits 32:53)*** (RPNL or $RPN_{32:53}$) [MAV=2.0]
The real page number is formed by the upper n bits of ($MAS7_{RPNU}$ || $MAS3_{RPNL}$), where n = 64 - $\log_2$(*page size* in bytes) and *page size* is

specified by $MAS1_{TSIZE}$ for a ***tlbwe*** instruction, by the SIZE field of the TLB entry if a TLB entry is being read by a ***tlbre*** or ***tlbsx*** instruction or by the LSIZE field of the LRAT entry if an LRAT entry is being read by a ***tlbre*** instruction. RPNL bits corresponding to bits that are not implemented in the RPN field of the TLB are treated as reserved.

54:57 ***User Bits*** (U0:U3)
See the corresponding TLB bit definition in Section 6.7.1. If one or more of these bits is not implemented in the TLB, the corresponding MAS3 bit is treated as reserved.

If $MAS1_{IND}$ = 0, $MAS3_{58:63}$ are defined as follows:

58 ***User State Execute Enable (UX)***
See the corresponding TLB bit definition in Section 6.7.1.

59 ***Supervisor State Execute Enable (SX)***
See the corresponding TLB bit definition in Section 6.7.1.

60 ***User State Write Enable (UW)***
See the corresponding TLB bit definition in Section 6.7.1.

61 ***Supervisor State Write Enable (SW)***
See the corresponding TLB bit definition in Section 6.7.1.

62 ***User State Read Enable (UR)***
See the corresponding TLB bit definition in Section 6.7.1.

63 ***Supervisor State Read Enable (SR)***
See the corresponding TLB bit definition in Section 6.7.1.

If $MAS1_{IND}$ = 1, $MAS3_{58:63}$ are defined as follows:

58:62 ***Sub-Page Size (SPSIZE)***
See the corresponding TLB field definition in Section 6.7.1.

63 ***Undefined)*** (UND)
The value of this bit is undefined after a ***tlbre*** or ***tlbsx***.

All other fields are reserved.

6.10.3.12 MAS4 Register

The MAS4 register contains fields for specifying default information to be pre-loaded on an Instruction or Data TLB Error interrupt if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor ($EPCR_{DMIUH}$ = 0), or the interrupt is directed to the guest state. See Section 6.11.4.7 for more information. MAS4 is a privileged register. The layout of the MAS4 register is shown in Figure 46 for MAV=1.0 and in Figure 47 for MAV=2.0.

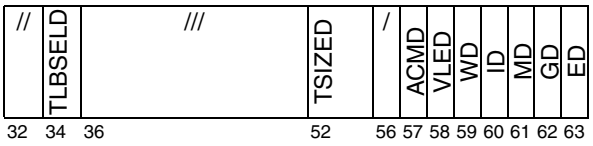


Figure 46. MAS4 register [MAV=1.0]

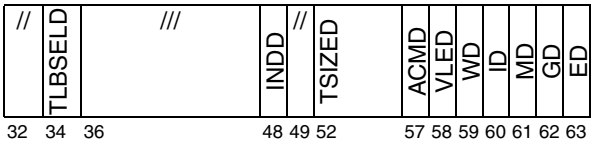


Figure 47. MAS4 register [MAV=2.0]

The MAS4 fields are described below.

Bit	Description
34:35	***TLBSEL Default Value*** (TLBSELD) Specifies the default value loaded in $MAS0_{TLBSEL}$ on the interrupt.
48	***IND Default Value*** (INDD) Specifies the default value loaded in $MAS1_{IND}$ and $MAS6_{SIND}$ on the interrupt.
52:55	***Default TSIZE Value*** (TSIZED) [MAV=1.0] Specifies the default value loaded into $MAS1_{TSIZE}$ on a TLB miss exception.
52:56	***Default TSIZE Value*** (TSIZED) [MAV=2.0] Specifies the default value loaded into $MAS1_{TSIZE}$ on a TLB miss exception. If $MMUCFG_{TWC}$ = 1, TSIZED is also the default value loaded into $MAS6_{ISIZE}$ on the interrupt.
57	***Default ACM Value*** (ACMD) Specifies the default value loaded into $MAS2_{ACM}$ on the interrupt.
58	***Default VLE Value*** (VLED) [Category: VLE] Specifies the default value loaded into $MAS2_{VLE}$ on the interrupt.
59	***Default W Value*** (WD) Specifies the default value loaded into $MAS2_{W}$ on the interrupt.
60	***Default I Value*** (ID) Specifies the default value loaded into $MAS2_{I}$ on the interrupt.
61	***Default M Value*** (MD) Specifies the default value loaded into $MAS2_{M}$ on the interrupt.
62	***Default G Value*** (GD) Specifies the default value loaded into $MAS2_{G}$ on the interrupt.

63 ***Default E Value*** (ED)
Specifies the default value loaded into $MAS2_E$ on the interrupt.

All other fields are reserved.

6.10.3.13 MAS5 Register

The MAS5 register contains fields for specifying LPID and GS values to be used when searching TLB entries with the ***tlbsrx.*** <E.TWC> and ***tlbsx*** instructions. The SLPID and SGS fields are used to match TLPID and TGS fields in the TLB entry. The MAS5 fields are also used for selecting TLB entries to be invalidated by the ***tlbilx*** or ***tlbivax*** instructions. MAS5 is a hypervisor resource. The layout of the MA5 register is shown in Figure 48.

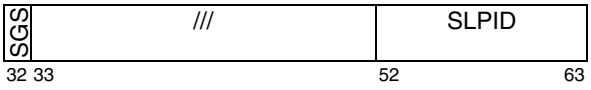


Figure 48. MAS5 register

The MAS5 fields are described below.

Bit **Description**

32 ***Search GS*** (SGS)
Specifies the GS value used when searching the TLB during execution of ***tlbsrx.*** <E.TWC> and ***tlbsx*** and for selecting TLB entries to be invalidated by ***tlbilx*** or ***tlbivax***. The SGS field is compared with the TGS field of each TLB entry to find a matching entry.

52:63 ***Search Logical Partition ID*** (SLPID)
Specifies the LPID value used when searching the TLB during execution of ***tlbsrx.*** <E.TWC> and ***tlbsx*** and for selecting TLB entries to be invalidated by ***tlbilx*** or ***tlbivax***. The SLPID field is compared with the TLPID field of each TLB entry to find a matching entry. Only the least significant $MMUCFG_{LPIDSIZE}$ bits of SLPID are implemented.

All other fields are reserved.

Programming Note

Hypervisor software should generally treat MAS5 as part of the partition state.

6.10.3.14 MAS6 Register

The MAS6 register contains fields for specifying PID, IND, and AS values to be used when searching TLB entries with the ***tlbsx*** instruction and, if $MMUCFG_{TWC}$ = 1 or $TLBnCFG_{HES}$ = 1, for specifying the PID, IND, AS, and size of the virtual address space to be used for selecting TLB entries to be invalidated by the ***tlbilx*** T=3 or ***tlbivax*** instructions. MAS6 is a privileged register.

The layout of the MAS6 register is shown in Figure 49 for MAV=1.0 and in Figure 50 for MAV=2.0.

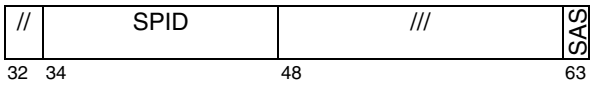


Figure 49. MAS6 register [MAV = 1.0]

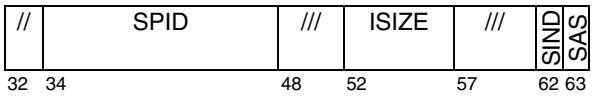


Figure 50. MAS6 register [MAV = 2.0]

The MAS6 fields are described below.

Bit **Description**

34:47 ***Search PID*** (SPID)
Specifies the value of PID used when searching the TLB during execution of ***tlbsx***. It also defines the PID of the TLB entry to be invalidated by ***tlbilx*** with T=1 or T=3 and ***tlbivax*** with EA_{61}=0. The number of bits implemented is the same as the number of bits implemented in the PID register.

Programming Note

The SPID field was referred to as SPID0 in previous versions of the architecture.

52:56 ***Invalidation Size*** (ISIZE) [MAV=2.0]
ISIZE defines the size of the virtual address space mapped by the TLB entry to be invalidated by ***tlbilx*** T=3 and ***tlbivax***. This field is only supported if $MMUCFG_{TWC}$ = 1 or, for any TLB array, $TLBnCFG_{HES}$ = 1. Otherwise this field is reserved.

Programming Note

To make code more portable across implementations, software should always set ISIZE before executing ***tlbilx*** T=3 and ***tlbivax***.

62 ***Indirect Value for Searches*** (SIND) [MAV=2.0 and Category: Embedded.Page Table]
Specifies the value of IND used when searching the TLB during execution of ***tlbsx***. It also defines the Indirect (IND) value of the TLB entry to be invalidated by ***tlbilx*** T=3 and ***tlbivax***.

63 ***Address Space Value for Searches*** (SAS)
Specifies the value of AS used when searching the TLB during execution of ***tlbsx***. It also defines the TS value of the TLB entry to be invalidated by ***tlbilx*** T=3 and ***tlbivax***.

All other fields are reserved.

6.10.3.15 MAS7 Register

The MAS7 register contains a field used for reading and writing an LRAT <E.HV.LRAT> or TLB entry. MAS7 register field is also loaded by the execution of the ***tlbsx*** instruction. The MAS7 register contains the high order address bits of the RPN for a TLB entry in implementations that support more than 32 bits of physical address. if the Embedded.Hypervisor.LRAT category is supported by such implementations, the high-order LRPN bits of the LRAT array can be read into or written from MAS7 by hypervisor software. If the Embedded.Hypervisor.LRAT category is supported, the RPN specified by MAS7and MAS3 is treated as an LPN for ***tlbwe*** executed in guest supervisor state (see Section 6.9). If no more than 32 bits of physical addressing are supported, it is implementation-dependent whether MAS7 is implemented. MAS7 is a privileged register. The layout of the MAS7 is shown in Figure 51.

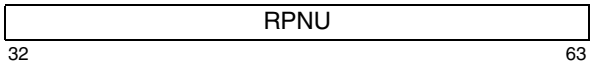


Figure 51. MAS7 register

The MAS7 fields are described below.

Bit	Description
32:63	***Real Page Number*** (bits 0:31) (RPNU or $RPN_{0:31}$) $RPN_{32:53}$ are accessed through MAS3. RPNU bits corresponding to bits that are not implemented in the RPN field of the TLB are treated as reserved.

6.10.3.16 MAS8 Register [Category: Embedded.Hypervisor]

The MAS8 register contains fields used for reading and writing an LRAT <E.HV.LRAT> or TLB entry. The MAS8 register fields are also loaded from a matching TLB entry by execution of a ***tlbsx*** instruction that is successful. The associated TLB fields are used to select a TLB entry during TLB address translation and TLB searches, and to force a Data Storage Interrupt to be directed to the hypervisor state. MAS8 is an hypervisor resource. The LPID field of the LRAT is used to select an LRAT entry during LRAT translation. The layout of the MAS8 register is shown in Figure 52.

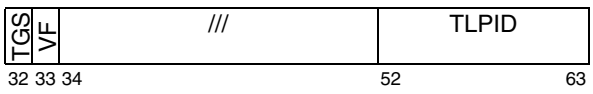


Figure 52. MAS8 register

The MAS8 fields are described below.

Bit	Description
32	***Translation Guest State*** (TGS) See the corresponding TLB bit definition in Section 6.7.1.
33	***Translation Virtualization Fault*** (VF) See the corresponding TLB bit definition in Section 6.7.1.
52:63	***Translation Logical Process ID*** (TLPID) See the corresponding TLB bit definition in Section 6.7.1 and the LRAT LPID field definition in Section 6.9.

All other fields are reserved.

Programming Note

Hypervisor software should generally treat MAS8 as part of the partition state. After executing ***tlbsx*** and ***tlbre***, hypervisor software may need to restore MAS8 before returning to guest state. This is especially important if the Embedded.Hypervisor.LRAT category is supported because a guest can execute a ***tlbwe*** instruction that writes a TLB entry with the MAS8 values.

Programming Note

For a TLB entry with VF=1, hypervisor software should have the execution permission bits set so that an instruction fetch of the page is prevented.

The VF bit can be used to force a Data Storage interrupt for virtualization of MMIO.

6.10.3.17 Accesses to Paired MAS Registers

In 64-bit implementations, certain MAS registers can be accessed in pairs with a single ***mtspr*** or ***mfspr*** instruction. The registers that can be accessed this way are shown in Table 10. These register pairs are treated as if they are a single 64-bit register by a ***mtspr*** or ***mfspr*** instruction.

Table 10:MAS Register Pairs

64-bit Pairs	SPR Number	Privileged mtspr & mfspr	Cat[2]
MAS5 || MAS6	348	hypv[1]	E.HV; 64
MAS8 || MAS1	349	hypv[1]	E.HV; 64
MAS7 || MAS3	372	yes	64
MAS0 || MAS1	373	yes	64

1 This register is a hypervisor resource, and can be accessed by one of these instructions only in hypervisor state (see Chapter 2).

2 See Section 1.3.5 of Book I. If multiple categories are listed, the register pair is only provided if all categories are supported. Otherwise the SPR number is treated as reserved.

6.10.3.18 MAS Register Update Summary

Table 11 summarizes how MAS registers are modified by Instruction TLB Error interrupt, Data TLB Error interrupt and the *TLB Management* instructions.

Table 11:MAS Register Update Summary for TLB operations

MAS Field Updated	Value Loaded on Event					
	Data TLB Error Interrupt on Load or Store that is not category E.PD or Instruction TLB Error Interrupt[2]	**Data TLB Error Interrupt on External Process ID Load <E.PD>[2]**	**Data TLB Error Interrupt on External Process ID Store <E.PD>[2]**	***tlbsx* hit**	***tlbsx* miss**	***tlbre***
$MAS0_{ATSEL}$ <E.HV>	0	0	0	0	0	—
$MAS0_{TLBSEL}$	$MAS4_{TLBSELD}$	$MAS4_{TLBSELD}$	$MAS4_{TLBSELD}$	TLB array that hit	$MAS4_{TLBSELD}$	—
$MAS0_{ESEL}$	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined	index of entry that hit	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** hardware hint, ***else*** undefined	—
$MAS0_{HES}$	$TLBnCFG_{HES}$ for array specified by $MAS4_{TLBSELD}$	$TLBnCFG_{HES}$ for array specified by $MAS4_{TLBSELD}$	$TLBnCFG_{HES}$ for array specified by $MAS4_{TLBSELD}$	0	$TLBnCFG_{HES}$ for array specified by $MAS4_{TLBSELD}$	—
$MAS0_{WQ}$ <E.TWC>	0b01	0b01	0b01	0b01	0b01	—
$MAS0_{NV}$	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** TLB array with the matching entry supports next victim ***then*** hardware hint, ***else*** undefined	***if*** TLB array [$MAS4_{TLBSELD}$] supports next victim ***then*** next hardware hint, ***else*** undefined	***if*** TLB array [$MAS0_{TLBSEL}$] supports next victim ***then*** hardware hint, ***else*** undefined
$MAS1_{V}$	1	1	1	1	0	TLB_{V}
$MAS1_{IPROT}$	0	0	0	TLB_{IPROT}	0	TLB_{IPROT}
$MAS1_{TID}$	PID	$EPLC_{EPID}$	$EPSC_{EPID}$	TLB_{TID}	$MAS6_{SPID}$	TLB_{TID}
$MAS1_{IND}$ <E.PT>	$MAS4_{INDD}$	$MAS4_{INDD}$	$MAS4_{INDD}$	TLB_{IND}	$MAS4_{INDD}$	TLB_{IND}
$MAS1_{TS}$	MSR_{IS} or MSR_{DS}	$EPLC_{EAS}$	$EPSC_{EAS}$	TLB_{TS}	$MAS6_{SAS}$	TLB_{TS}
$MAS1_{TSIZE}$	$MAS4_{TSIZED}$	$MAS4_{TSIZED}$	$MAS4_{TSIZED}$	TLB_{SIZE}	$MAS4_{TSIZED}$	TLB_{SIZE}
$MAS2_{EPN}$	$EA_{0:53}$[1]	$EA_{0:53}$	$EA_{0:53}$	TLB_{EPN}	undefined	TLB_{EPN}
$MAS2_{ACM}$	$MAS4_{ACMD}$	$MAS4_{ACMD}$	$MAS4_{ACMD}$	TLB_{ACM}	$MAS4_{ACMD}$	TLB_{ACM}
$MAS2_{VLE}$ <VLE>	$MAS4_{VLED}$	$MAS4_{VLED}$	$MAS4_{VLED}$	TLB_{VLE}	$MAS4_{VLED}$	TLB_{VLE}
$MAS2_{W\ I\ M\ G\ E}$	$MAS4_{WD\ ID\ MD\ GD\ ED}$	$MAS4_{WD\ ID\ MD\ GD\ ED}$	$MAS4_{WD\ ID\ MD\ GD\ ED}$	$TLB_{W\ I\ M\ G\ E}$	$MAS4_{WD\ ID\ MD\ GD\ ED}$	$TLB_{W\ I\ M\ G\ E}$
$MAS3_{RPNL}$	0	0	0	$TLB_{RPN[32:53]}$	0	$TLB_{RPN[32:53]}$

Table 11:MAS Register Update Summary for TLB operations

MAS Field Updated	Value Loaded on Event					
	Data TLB Error Interrupt on Load or Store that is not category E.PD or Instruction TLB Error Interrupt[2]	**Data TLB Error Interrupt on External Process ID Load <E.PD>[2]**	**Data TLB Error Interrupt on External Process ID Store <E.PD>[2]**	***tlbsx* hit**	***tlbsx* miss**	***tlbre***
$MAS3_{U0:U3}$	0	0	0	$TLB_{U0:U3}$	0	$TLB_{U0:U3}$
$MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$	0	0	0	***if*** Category: E.PT is not supported or TLB_{IND} = 0 ***then*** $TLB_{UX\ SX\ UW\ SW\ UR\ SR}$	0	***if*** Category: E.PT is not supported or TLB_{IND} = 0 ***then*** $TLB_{UX\ SX\ UW\ SW\ UR\ SR}$
$MAS3_{SPSIZE}$	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	***if*** Category: E.PT supported and TLB_{IND} = 1 ***then*** TLB_{SPSIZE}	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	***if*** Category: E.PT supported and TLB_{IND} = 1 ***then*** TLB_{SPSIZE}
$MAS3_{UND}$	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	***if*** Category: E.PT supported and TLB_{IND} = 1 ***then*** undefined	(see the entry for $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$)	***if*** Category: E.PT supported and TLB_{IND} = 1 ***then*** undefined
MAS5 <E.HV> & MAS4	—[3]	—[3]	—[3]	—	—	—
$MAS6_{SPID}$	PID	$EPLC_{EPID}$	$EPSC_{EPID}$	—	—	—
$MAS6_{ISIZE\ if\ TLBnCFG_{HES}=1\ or\ <E.TWC>}$	$MAS4_{TSIZED}$	$MAS4_{TSIZED}$	$MAS4_{TSIZED}$	—	—	—
$MAS6_{SAS}$	MSR_{IS} or MSR_{DS}	$EPLC_{EAS}$	$EPSC_{EAS}$	—	—	—
$MAS6_{SIND}$ <E.PT>	$MAS4_{INDD}$	$MAS4_{INDD}$	$MAS4_{INDD}$	—	—	—
$MAS7_{RPNU}$	0	0	0	$TLB_{RPN[0:31]}$	0	$TLB_{RPN[0:31]}$
$MAS8_{TGS\ VF\ TLPID}$ <E.HV>	—[3]	—[3]	—[3]	$TLB_{TGS\ VF\ TLPID}$	—	$TLB_{TGS\ VF\ TLPID}$

1. If MSR_{CM}=0 (32-bit mode) at the time of the exception, $EPN_{0:31}$ are set to 0.
2. If the E.HV category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor ($EPCR_{DMIUH}$ = 0), or the interrupt is directed to the guest state.
3. MAS5 and MAS8 are not updated on a Data or Instruction TLB Error interrupt. The hypervisor should ensure they already contain values appropriate to the partition.

6.11 Storage Control Instructions

6.11.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a ***dcbz*** or ***dcba*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the hardware need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 6.6) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the thread enters any power conserving mode in which data cache contents are not retained.

Data Cache Block Invalidate *X-form*

dcbi RA,RB

31	///	RA	RB	470	/
0	6	11	16	21	31

```
if RA=0 then b ← 0
else         b ← (RA)
EA ← b + (RB)
InvalidateDataCacheBlock( EA )
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any thread, then the block is invalidated in those data caches. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in any such data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of this thread, then the block is invalidated in that data cache. On some implementations, before the block is invalidated, if any locations in the block are considered to be modified in that data cache, those locations are written to main storage and additional locations in the block may be written to main storage.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Store* (see Section 6.7.6.5) on implementations that invalidate a block without first writing to main storage all locations in the block that are considered to be modified in the data cache, except that the invalidation is not ordered by ***mbar***. On other implementations this instruction is treated as a *Load* (see the section cited above).

If a thread holds a reservation and some other thread executes a ***dcbi*** that specifies a location in the same reservation granule, the reservation may be lost only if the ***dcbi*** is treated as a *Store*.

dcbi may cause a cache locking exception, the details of which are implementation-dependent.

This instruction is privileged.

Special Registers Altered:

None

6.11.2 Cache Locking [Category: Embedded Cache Locking]

The *Embedded Cache Locking* category defines instructions and methods for locking cache blocks for frequently used instructions and data. Cache locking allows software to instruct the cache to keep latency sensitive data readily available for fast access. This is accomplished by marking individual cache blocks as locked.

A locked block differs from a normal block in the cache in the following way:

- blocks that are locked in the cache do not participate in the normal replacement policy when a block must be replaced.

6.11.2.1 Lock Setting, Query, and Clearing

Cache Locking instructions are used by software to indicate which blocks in a cache should be locked, unlocked, or queried for lock status.

Blocks are locked into the cache by software using *Lock Set* instructions. The following instructions are provided to lock data items into the data and instruction cache:

- ***dcbtls*** - Data cache block touch and lock set.
- ***dcbtstls*** - Data cache block touch for store and lock set.
- ***icbtls*** - Instruction cache block touch and lock set.

The RA and RB operands in these instructions are used to identify the block to be locked. The CT field indicates which cache in the cache hierarchy should be targeted. (See Section 4.3 of Book II.)

These instructions are similar in nature to the ***dcbt***, ***dcbtst***, and ***icbt*** instructions, but are not hints and thus locking instructions do not execute speculatively and may cause additional exceptions. For unified caches, both the instruction lock set and the data lock set target the same cache.

Blocks are unlocked from the cache by software using *Lock Clear* instructions. The following instructions are provided to unlock instructions and data in their respective caches:

- ***dcblc*** - Data cache block lock clear.
- ***icblc*** - Instruction cache block lock clear.

The RA and RB operands in these instructions are used to identify the block to be unlocked. The CT field indicates which cache in the cache hierarchy should be targeted.

Additionally, an implementation-dependent method can be provided for software to clear all the locks in the cache.

The status of whether a cache block is locked in the cache can be queried by software using *Lock Query* instructions:

- ***dcblq.*** - Data cache block lock query.
- ***icblq.*** - Instruction cache block lock query.

The RA and RB operands in these instructions are used to identify the block to be queried. The CT field indicates which cache in the cache hierarchy should be targeted. These instructions set CR0 to indicate whether the block is locked or is not locked.

An implementation is not required to unlock blocks that contain data that has been invalidated unless it is explicitly unlocked with a ***dcblc*** or ***icblc*** instruction; if the implementation does not unlock the block upon invalidation, the block remains locked even though it contains invalid data. If the implementation does not clear locks when the associated block is invalidated, the method of locking is said to be *persistent*; otherwise it is *not persistent*. An implementation may choose to implement locks as persistent or not persistent; however, the preferred method is persistent.

It is implementation-dependent if cache blocks are implicitly unlocked in the following ways:

- A locked block is invalidated as the result of a ***dcbi***, ***dcbf***, ***dcbfep***, ***icbi***, or ***icbiep*** instruction.
- A locked block is evicted because of an overlocking condition.
- A snoop hit on a locked block that requires the block to be invalidated. This can occur because the data the block contains has been modified external to the thread, or another thread has explicitly invalidated the block.
- The entire cache containing the locked block is invalidated.

6.11.2.2 Error Conditions

Setting locks in the cache can fail for a variety of reasons. A *Lock Set* instruction addressing a byte in storage that is not allowed to be accessed by the storage access control mechanism (see Section 6.7.6) will cause a Data Storage interrupt (DSI). Addresses referenced by *Cache Locking* instructions are always translated as data references; therefore, ***icbtls*** instructions that fail to translate or are not allowed by the storage access control mechanism cause Data TLB Error interrupts and Data Storage interrupts, respectively. *Cache Locking* and clearing operations can fail due to non-privileged access. The methods for determining other failure conditions such as unable-to-lock or overlocking (see below), is implementation-dependent.

If the Embedded.Hypervisor category is supported and $MSRP_{UCLEP}$ = 1, an attempt to execute a *Cache Locking* instruction in guest state results in an Embedded Hypervisor Privilege exception if MSR_{PR} = 0 or a cache

locking exception if MSR_{PR} = 1. When the Embedded.Hypervisor category is not supported, $MSRP_{UCLEP}$ = 0, or MSR_{GS} = 0, then if a *Cache Locking* instruction is executed in user mode and MSR_{UCLE} is 0, a cache locking exception occurs. If a Data Storage interrupt occurs as a result of a cache locking exception, one of the following ESR or GESR bits is set to 1 (GESR if the Embedded.Hypervisor category is supported and the interrupt is directed to the guest. Otherwise, ESR).

Bit	Description
42	***DLK_0***
	0 Default setting.
	1 A ***dcbtls***, ***dcbtstls***, ***dcblq.***, or ***dcblc*** instruction was executed in user mode.
43	***DLK_1***
	0 Default setting.
	1 An ***icbtls*** or ***icblc*** instruction was executed in user mode.

[Category:Embedded.Hypervisor]
The behavior of *Cache Locking* instructions in guest privileged state (***dcbtls***, ***dcbtstls***, ***dcblc***, ***dcblq.***, ***icbtls***, ***icblc***, ***icblq.***) is dependent on the setting of $MSRP_{UCLEP}$. When $MSRP_{UCLEP}$ = 0, *Cache Locking* instructions are permitted to execute normally in the guest privileged state. When $MSRP_{UCLEP}$ = 1, cache locking instructions are not permitted to execute in the guest privileged state and cause an Embedded Hypervisor Privilege exception when executed. See Section 4.2.2, “Machine State Register Protect Register (MSRP)”.

6.11.2.2.1 Overlocking

If no exceptions occur for the execution of an ***dcbtls***, ***dcbtstls***, or ***icbtls*** instruction, an attempt is made to lock the specified block into the cache. If all of the available cache blocks into which the specified block may be loaded are already locked, an overlocking condition occurs. The overlocking condition may be reported in an implementation-dependent manner.

If an overlocking condition occurs, it is implementation-dependent whether the specified block is not locked into the cache or if another locked block is evicted and the specified block is locked.

The selection of which block is replaced in an overlocking situation is implementation-dependent. The overlocking condition is still said to exist, and is reflected in any implementation-dependent overlocking status.

An attempt to lock a block that is already present and valid in the cache will not cause an overlocking condition.

If a cache block is to be loaded because of an instruction other than a *Cache Management* or *Cache Locking* instruction and all available blocks into which the block can be loaded are locked, the instruction executes and completes, but no cache blocks are unlocked and the block is not loaded into the cache.

Programming Note

Since caches may be shared among threads, an overlocking condition may occur when loading a block even though a given thread has not locked all the available cache blocks. Similarly. blocks may be unlocked as a result of invalidations by other threads.

6.11.2.2.2 Unable-to-lock, Unable-to-unlock, and Unable-to-query Conditions

If no exceptions occur and no overlocking condition exists, an attempt to set, query, or unlock a lock may fail if any of the following are true:

- The target address is marked Caching Inhibited, or the storage control attributes of the address use a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.
- Any other implementation-specific error conditions are detected.

If an unable-to-lock or unable-to-unlock condition occurs, the *Lock Set* or *Lock Clear* instruction is treated as a no-op and the condition may be reported in an implementation-dependent manner. If an unable-to-query condition occurs, the CR0 status bit for the query is set to 0.

6.11.2.3 Cache Locking Instructions

Data Cache Block Lock Query X-form

dcblq. CT,RA,RB

31	/	CT	RA	RB	422	1
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The block containing the byte addressed by EA in the data cache specified by the CT field is queried to determine its lock status and CR0 is set as follows:

CR0 = 0b00 || status || XER_{SO}

Status is set to 1 if the block is locked in the data cache specified by the CT field. Status is set to 0 if the block is not locked in the data cache specified by the CT field.

The instruction is treated as a *Load*.

If an unable-to-query condition occurs (see Section 6.11.2.2.2) status is set to 0.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

CR0

Instruction Cache Block Lock Query X-form

icblq. CT,RA,RB

31	/	CT	RA	RB	198	1
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The block containing the byte addressed by EA in the instruction cache specified by the CT field is queried to determine its lock status and CR0 is set as follows:

CR0 = 0b00 || status || XER_{SO}

Status is set to 1 if the block is locked in the instruction cache specified by the CT field. Status is set to 0 if the block is not locked in the instruction cache specified by the CT field.

The instruction is treated as a *Load*.

If an unable-to-query condition occurs (see Section 6.11.2.2.2) status is set to 0.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

CR0

Data Cache Block Touch and Lock Set *X-form*

dcbtls CT,RA,RB

31	/	CT	RA	RB	166	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbtls*** instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded and locked into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

If the block already exists in the cache, the block is locked without accessing storage. An unable-to-lock condition may occur (see Section 6.11.2.2.2), or an overlocking condition may occur (see Section 6.11.2.2.1).

The ***dcbtls*** instruction may complete before the operation it causes has been performed.

The instruction is treated as a *Load*.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

None

Data Cache Block Touch for Store and Lock Set *X-form*

dcbtstls CT,RA,RB

31	/	CT	RA	RB	134	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbtstls*** instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded and locked into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

If the block already exists in the cache, the block is locked without accessing storage. An unable-to-lock condition may occur (see Section 6.11.2.2.2), or an overlocking condition may occur (see Section 6.11.2.2.1).

The ***dcbtstls*** instruction may complete before the operation it causes has been performed.

The instruction is treated as a *Store*.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

None

Instruction Cache Block Touch and Lock Set *X-form*

icbtls CT,RA,RB

31	/	CT	RA	RB	486	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***icbtls*** instruction causes the block containing the byte addressed by EA to be loaded and locked into the instruction cache specified by CT, and provides a hint that the program will probably soon execute code from the block. See Section 4.3 of Book II for a definition of the CT field.

If the block already exists in the cache, the block is locked without refetching from memory.

This instruction treated as a *Load* (see Section 4.3 of Book II).

If an unable-to-lock condition occurs (see Section 6.11.2.2.2) no operation is performed.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

None

Instruction Cache Block Lock Clear *X-form*

icblc CT,RA,RB

31	/	CT	RA	RB	230	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The block containing the byte addressed by EA in the instruction cache specified by the CT field is unlocked.

The instruction is treated as a *Load*.

If an unable-to-lock condition occurs (see Section 6.11.2.2.2) no operation is performed. If the block containing the byte addressed by EA is not locked in the specified cache, no cache operation is performed.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

None

Data Cache Block Lock Clear *X-form*

dcblc CT,RA,RB

31	/	CT	RA	RB	390	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The block containing the byte addressed by EA in the data cache specified by the CT field is unlocked.

The instruction is treated as a *Load*.

If an unable-to-lock condition occurs (see Section 6.11.2.2.2) no operation is performed. If the block containing the byte addressed by EA is not locked in the specified cache, no cache operation is performed.

This instruction is privileged unless the Embedded Cache Locking.User Mode category is supported. If the Embedded Cache Locking.User Mode category is supported, this instruction is privileged only if MSR_{UCLE}=0.

Special Registers Altered:

None

Programming Note

The ***dcblc*** and ***icblc*** instructions are used to remove locks previously set by the corresponding lock set instructions.

6.11.3 Synchronize Instruction

The *Synchronize* instruction is described in Section 4.4.3 of Book II, but only at the level required by an application programmer. This section describes properties of the instruction that are relevant only to operating system programmers.

When L=1, the ***Sync*** instruction provides an ordering function for the operations caused by the *Message Send* instruction and previous *Stores* for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The stores must be performed with respect to the thread receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt is taken.

In conjunction with the ***tlbie*** and ***tlbsync*** instructions, when L=0, the ***sync*** instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the ***tlbsync*** instruction description on page 1141.

When L=0, the ***Sync*** instruction also provides an ordering function for the operations caused by the *Message Send* instruction and previous *Stores*. The stores must be performed with respect to the thread receiving the message prior to any access caused by or associated with any instruction executed after the corresponding interrupt is taken.

6.11.4 LRAT [Category: Embedded.Hypervisor.LRAT] and TLB Management

Unless the Embedded.Page Table category is supported, no format for the Page Tables or the Page Table Entries is implied. Software has significant flexibility in implementing a custom replacement strategy. For example, software may choose to set IPROT=1 for TLB entries that correspond to frequently used storage, so that those entries are never cast out of the TLB and TLB Miss exceptions to those pages never occur. At a minimum, software must maintain a TLB entry or entries for the Instruction and Data TLB Error interrupt handlers.

TLB management is performed in software with some hardware assist. This hardware assist consists of a minimum of:

- Automatic recording of the effective address causing a TLB Error interrupt. For Instruction TLB Error interrupts, the address is saved in the Save/ Restore Register 0. For Data TLB Error interrupts, the address is saved in the Data Exception Address Register.
- Automatic updating of the MAS register on the occurrence of a TLB Error interrupt if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor ($EPCR_{DMIUH}$ = 0), or the interrupt is directed to the guest state.
- Instructions for reading, writing, searching, invalidating, and synchronizing the TLB. If the Embedded.Hypervisor.LRAT category is supported, a subset of these instructions can also be used for reading and writing the LRAT.

Programming Note

If the Embedded.Hypervisor category is supported and if $EPCR_{ITLBGS\ DTLBGS}$ = 0b00, the hypervisor can virtualize the physical TLB by keeping a software copy of at least the guest operating system TLB entries with IPROT=1 and avoid keeping the guest Instruction and Data TLB Error interrupt handlers in the physical TLB.

6.11.4.1 Reading TLB or LRAT Entries

TLB entries can be read by executing ***tlbre*** instructions. If the Embedded.Hypervisor.LRAT category is supported, LRAT entries can be read by also executing ***tlbre*** instructions. At the time of ***tlbre*** execution, a TLB array is selected if $MAS0_{ATSEL}$=0 or MSR_{GS}=1. The LRAT array is selected if $MAS0_{ATSEL}$=1 and MSR_{GS}=0. The LRAT can be read only when in hypervisor state.

If a TLB array is selected, $MAS0_{TLBSEL}$ selects the TLB array to be read. If $TLBnCFG_{HES}$=0, the TLB entry in the selected TLB array is selected by $MAS0_{ESEL}$ and $MAS2_{EPN}$. In this case, $MAS0_{ESEL}$ selects one of the possible entries that can be used for a given $MAS2_{EPN}$. If $TLBnCFG_{HES}$=1, the TLB entry in the selected TLB array is selected by $MAS0_{ESEL}$ and by a hardware generated hash based on $MAS1_{TID\ TSIZE}$, and $MAS2_{EPN}$. In this case, $MAS0_{ESEL}$ selects one of the possible entries that can be used for a given $MAS1_{TID\ TSIZE}$ and $MAS2_{EPN}$.

If an LRAT array is selected, the LRAT entry is selected by $MAS0_{ESEL}$ and $MAS2_{EPN}$. In this case, $MAS0_{ESEL}$ selects one of the possible entries that can be used for a given $MAS2_{EPN}$.

Specifying invalid values for $MAS0_{TLBSEL}$, $MAS0_{ESEL}$, $MAS2_{EPN}$ or, if $TLBnCFG_{HES}$=1, $MAS1_{TID}$ or $MAS1_{TSIZE}$, results in $MAS1_{V}$ being set to 0 and undefined results in the other MAS register fields that are loaded by the ***tlbre*** instruction. Which values are invalid is implementation-dependent. For example, even though an implementation has only one TLB array, the implementation could simply ignore $MAS0_{TLBSEL}$, read the selected entry in the TLB array, and load the MAS registers from the TLB entry regardless of the $MAS0_{TLBSEL}$ value.

Programming Note

When reading TLB entries, $MAS2_{EPN}$ or a subset of the bits in $MAS2_{EPN}$ is used to form the index for accessing the TLB array, i.e. $MAS2_{EPN}$ isn't necessarily an effective page number.

6.11.4.2 Writing TLB or LRAT Entries

TLB entries can be written by executing ***tlbwe*** instructions. If the Embedded.Hypervisor.LRAT category is supported, LRAT entries can also be written by executing ***tlbwe*** instructions. At the time of ***tlbwe*** execution, a TLB array is selected if $MAS0_{ATSEL}$=0 or MSR_{GS}=1. The LRAT array is selected if $MAS0_{ATSEL}$=1 and MSR_{GS}=0. The LRAT can be written only when in hypervisor state.

If a TLB array is selected, $MAS0_{TLBSEL}$ selects the TLB array to be written. If $TLBnCFG_{HES}$=0, the TLB entry in the selected TLB array is selected by $MAS0_{ESEL}$ and $MAS2_{EPN}$. In this case, $MAS0_{ESEL}$ selects one of the possible entries that can be used for a given $MAS2_{EPN}$. If $TLBnCFG_{HES}$=1 and $MAS0_{HES}$=0, the TLB entry in the selected TLB array is selected by a hardware generated hash based on $MAS1_{TID\ TSIZE}$, $MAS2_{EPN}$, and $MAS0_{ESEL}$. In this case, $MAS0_{ESEL}$ selects one of the possible entries that can be used for a given $MAS0_{TLBSEL}$, $MAS1_{TID\ TSIZE}$, and $MAS2_{EPN}$. If $TLBnCFG_{HES}$=1 and $MAS0_{HES}$=1, the TLB entry in the selected TLB array is selected by a hardware replacement algorithm and a hardware generated hash based on $MAS1_{TID\ TSIZE}$ and $MAS2_{EPN}$.

If an LRAT array is selected, the LRAT entry is selected by $MAS0_{ESEL}$ and $MAS2_{EPN}$. In this case, $MAS0_{ESEL}$ selects one of the possible entries that can be used for a given $MAS2_{EPN}$. LRAT entries can be written only with $MAS0_{HES}$= 0.

At the time of ***tlbwe*** execution, the MAS registers contain the contents to be written to the indexed TLB entry. Upon completion of the ***tlbwe*** instruction, the contents of the MAS registers corresponding to TLB entry fields will be written to the indexed TLB entry, except that if the Embedded.Hypervisor.LRAT category is supported, guest execution of *TLB Management* instructions is enabled ($EPCR_{DGTMI}$=0), MSR_{PR} = 0, MSR_{GS} = 1, and, for the TLB array to be written, $TLBnCFG_{GTWE}$ = 1, the RPN from the MAS registers is treated like an LPN and translated by the LRAT, and the RPN from the LRAT is written to the TLB entry if the translation is successful. See Section 6.9. If the LRAT translation fails, an LRAT Miss exception occurs.

If the Embedded.Hypervisor.LRAT category is supported and a guest supervisor executes a ***tlbwe*** instruction with $MAS1_{IPROT}$=1 or for which the entry to be overwritten has IPROT=1, an Embedded Hypervisor Privilege exception occurs. However, if $MAS0_{WQ}$=0b01, it is implementation-dependent whether the Embedded Hypervisor Privilege exception occurs.

If the Embedded.Hypervisor.LRAT category is supported and a guest supervisor executes a ***tlbwe*** instruction with $MAS0_{HES}$ =0, it is implementation-dependent whether the Embedded Hypervisor Privilege exception occurs.

If a TLB entry is being written with $MAS0_{HES}$ =1, the hardware replacement algorithm picks an entry in the selected array from the set of entries which can be used for translating addresses with the specified TID, TSIZE and EPN. Whenever possible, an entry with IPROT equal to 0 is selected. However, an Embedded Hypervisor Privilege exception occurs on a ***tlbwe*** if all the following conditions are met.

- The Embedded.Hypervisor.LRAT category is supported.
- The ***tlbwe*** is executed in guest supervisor state.
- IPROT=1 for all entries which can be used for translating addresses with the specified TID, TSIZE and EPN.
- $MAS0_{WQ}$= 0b00

If $MAS0_{WQ}$= 0b01, $MAS0_{HES}$ =1, and the first three of the above conditions are met, it is implementation-dependent whether an Embedded Hypervisor Privilege exception occurs.

For TLBs with $TLBnCFG_{HES}$=1, the relationship between the TLB entry selected by a ***tlbwe*** with $MAS0_{HES}$=0 versus the TLB entry selected by a ***tlbwe*** with $MAS0_{HES}$=1 is implementation-dependent and may depend on the history of TLB use.

If an invalid value is specified for $MAS0_{TLBSEL}$ $MAS0_{ESEL}$ or $MAS2_{EPN}$, either no TLB entry is written by the ***tlbwe***, or the ***tlbwe*** is performed as if some implementation-dependent, valid value were substituted for the invalid value, or an Illegal Instruction exception occurs. If the page size specified by $MAS1_{TSIZE}$ is not supported by the specified array, the ***tlbwe*** may be performed as if TSIZE were some implementation-dependent value, or an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported but the Embedded.Hypervisor.LRAT category is not supported, the ***tlbwe*** instruction is hypervisor privileged. Otherwise, this instruction is privileged.

Programming Note

Since a hardware replacement algorithm selects the entry for a ***tlbwe*** instruction with $MAS0_{HES}$ = 1, it is typically not possible to write the same entry using a second ***tlbwe*** instruction with $MAS0_{HES}$ = 1. Doing so might create multiple entries for the same virtual page. If software needs to change the value of any of the TLB fields, software should generally invalidate the original entry before executing the second ***tlbwe*** instruction with the new values.

6.11.4.2.1 TLB Write Conditional [Embedded.TLB Write Conditional]

The ***tlbsrx.*** <E.TWC> instruction and ***tlbwe*** instruction with $MAS0_{WQ}$ = 0b01 together permit a convenient way for software to write a TLB entry while ensuring that the entry is not a duplicate entry and is not a stale, invalid value. Without the *TLB Write Conditional* facility, software must hold a software lock during the process of creating a TLB entry in order to prevent other threads from updating a shared TLB or invalidating a TLB entry. The ***tlbsrx.*** <E.TWC> instruction has two effects that occur either at the same time or in the following order.

1. A TLB-reservation is established for a virtual address, and, if the Embedded Page Table category is supported, an associated IND value.
2. A search of the selected TLB array is performed.

The TLB-reservation is used by a subsequent ***tlbwe*** instruction that writes a TLB entry (i.e. $MAS0_{ATSEL}$ = 0 or MSR_{GS}=1) with $MAS0_{WQ}$ = 0b01. The TLB is only written by this ***tlbwe*** if the TLB-reservation still exists at the instant the TLB is written. A ***tlbwe*** that writes the TLB is said to "succeed". *TLB Write Conditional* cannot be used for the LRAT.

TLB-reservation

A TLB-reservation is set by a ***tlbsrx.*** <E.TWC> instruction. The TLB-reservation has an associated IND <E.PT> and virtual address consisting of AS, PID, $EA_{0:53}$, LPID <E.HV>, and GS <E.HV>. These values come from $MAS1_{IND}$ <E.PT>, $MAS1_{TS}$ $MAS1_{TID}$, $MAS2_{EPN}$, $MAS5_{SLPID}$ <E.HV>, and $MAS5_{SGS}$ <E.HV>, respectively. There is no specific page size associated with the TLB-reservation. The TLB-reservation applies to any virtual page that contains the virtual address. There is only one TLB-reservation in a thread.

The TLB-reservation is cleared by any of the following.

- The thread holding the TLB-reservation executes another ***tlbsrx.*** <E.TWC>: This clears the first TLB-reservation and establishes a new one.
- A ***tlbivax*** is executed by any thread and all the following conditions are met.
  - Either the Embedded.Hypervisor category is not supported or the $MAS5_{SGS}$ and $MAS5_{SLPID}$ values used by the ***tlbivax*** match the GS and LPID values associated with the TLB-reservation.
  - The $MAS6_{SPID}$ and $MAS6_{SAS}$ values used by the ***tlbivax*** match the PID and AS values associated with the TLB-reservation.
  - The $EA_{0:n-1}$ values of the ***tlbivax*** match the $EA_{0:n-1}$ values associated with the TLB-reservation, where n=64-$\log_2$(*page size* in bytes) and *page size* is specified by the $MAS6_{ISIZE}$.
- The thread holding the TLB-reservation or another thread that shares the TLB with this thread executes a ***mtspr*** to MMUCSR0 that performs a TLB invalidate all operation and LPIDR contents of the thread executing the ***mtspr*** matches the LPID value associated with the TLB-reservation.
- If Category: Embedded.Hypervisor is supported, and a ***tlbilx*** with T = 0 is executed by the thread holding the TLB-reservation or by a thread that shares the TLB with this thread, and the $MAS5_{SLPID}$ value used by the ***tlbilx*** matches the LPID value associated with the TLB-reservation.
- If Category: Embedded.Hypervisor is supported, and a ***tlbilx*** with T = 1 is executed by the thread holding the TLB-reservation or by a thread that shares the TLB with this thread, and the $MAS5_{SLPID}$ and $MAS6_{SPID}$ values used by the ***tlbilx*** match the LPID and PID values associated with the TLB-reservation.
- If Category: Embedded.Hypervisor is supported, and a ***tlbilx*** with T = 3 is executed by the thread holding the TLB-reservation or by a thread that shares the TLB with this thread, the $MAS5_{SGS}$, $MAS5_{SLPID}$, $MAS6_{SPID}$, and $MAS6_{SAS}$ values used by the ***tlbilx*** match the GS, LPID, PID, and AS values associated with the TLB-reservation, and $EA_{0:n-1}$ values of the ***tlbilx*** match the $EA_{0:n-1}$ values associated with the TLB-reservation, where n=64-$\log_2$(*page size* in bytes) and *page size* is specified by the $MAS6_{ISIZE}$.
- A ***tlbwe*** instruction is executed by the thread holding the TLB-reservation or by a thread that shares the TLB with this thread, and all the following are true.
  - An interrupt does not occur as a result of the ***tlbwe*** instruction.
  - The Embedded.Hypervisor category is not supported or the $MAS8_{TLPID}$ value used by the ***tlbwe*** match the LPID value associated with the TLB-reservation.
  - The Embedded.Hypervisor category is not supported or the $MAS8_{TGS}$ value used by the ***tlbwe*** match the GS value associated with the TLB-reservation.
  - The $MAS1_{TID}$ value used by the ***tlbwe*** matches the PID value associated with the TLB-reservation.
  - The Embedded.Page Table category is not supported or the $MAS1_{IND}$ value used by the ***tlbwe*** matches the IND value associated with the TLB-reservation.
  - The $MAS1_{TS}$ value used by the ***tlbwe*** matches the AS value associated with the TLB-reservation.
  - Bits 0:(n-1) of $MAS2_{EPN}$ used by the ***tlbwe*** match the $EA_{0:n-1}$ values associated with the TLB-reservation, where n=64-$\log_2$(*page size* in bytes) and *page size* is specified by the $MAS1_{TSIZE}$ used by the ***tlbwe***.
  - Either of the following conditions are met.
    - The $MAS0_{WQ}$ used by the ***tlbwe*** instruction is 0b00.

- The $MAS0_{WQ}$ used by the ***tlbwe*** instruction is 0b01 and the TLB-reservation for the thread executing the ***tlbwe*** exists.

- The thread that has the TLB-reservation or another thread that shares the TLB with this thread that, as a result of a Page Table translation, writes a TLB entry and all the following conditions are met.
  - The TS and $EA_{0:n-1}$ values for the new TLB entry match the corresponding values associated with the TLB-reservation where n = 64-$\log_2$(*page size* in bytes), where *page size* is specified by the SIZE value written to the TLB entry.
  - The Embedded.Hypervisor category is not supported or TLPID for the new TLB entry matches the LPID associated with the TLB-reservation.
  - The Embedded.Hypervisor category is not supported or TGS for the new TLB entry matches the GS associated with the TLB-reservation.
  - The TID for the new TLB entry matches the PID associated with the TLB-reservation.
  - The Valid bit for the new TLB entry is 1.
  - The IND value associated with the TLB-reservation is 0.

Implementations are allowed to clear a TLB-reservation for conditions other than those specified above. The architecture assures that a TLB-reservation will be cleared when required per the above requirements, but does not guarantee that these are the only conditions for clearing a TLB-reservation. However, the occurrence of an interrupt does not clear a TLB-reservation.

Programming Note

Software running on two threads that share a TLB should not attempt to create two TLB entries that would both translate a specific virtual address and where the TID or LPID values differ, i.e. one of the values is zero and the other is nonzero. The TLB-reservation will not protect against this case, since a TLB-reservation is not cleared by a ***tlbwe*** unless there is an exact match on the PID and LPID values.

Likewise software should not attempt to create a Page Table entry and a TLB entry where both entries would translate a specific virtual address, where the TLB array written by the Page Table translation is used by the same thread that uses this TLB entry, and where the TID or LPID values of the PTE and TLB entry differ, i.e. one of the values is zero and the other is nonzero. The TLB-reservation will not protect against this case, since a TLB-reservation is not cleared by a TLB write resulting from a Page Table translation unless there is an exact match on the PID and LPID values.

Synchronization of TLB-reservation

The side-effect of a ***tlbsrx.*** <E.TWC> instruction setting the TLB-reservation can be synchronized by a context synchronizing instruction or event.

Programming Note

A common operation is to ensure that the TLB-reservation has been set by a ***tlbsrx.*** <E.TWC> instruction before executing a subsequent *Load* instruction of a software page table entry in order to ensure the TLB-reservation detects an invalidation of the entry that was accessed. Beside using a context synchronizing instruction, software can also ensure the TLB-reservation has been set by a ***tlbsrx.*** <E.TWC> instruction by reading the CR0 field or CR with a ***mfocrf*** or ***mfcr*** instruction after the ***tlbsrx.*** <E.TWC> and creating a dependency between the data read from CR0 or CR and the address used for the subsequent *Load* instruction.

Serialization of TLB operations

Regardless of which threads initiated the operations, all operations (reads, writes, invalidates, and searches) involving a single TLB are defined to be serialized such that only one operation occurs at a time. This operation is consistent with the program order of the thread performing the TLB operation. This also applies to a TLB that is shared by multiple threads. Even if there is no matching TLB entry on a ***tlbivax***, the TLB is still searched to determine there is no matching entry and this search is still referred to as the TLB invalidation.

If two threads share a TLB and both simultaneously execute a ***tlbsrx.*** <E.TWC> instruction for a virtual address in a virtual page V, and then both threads execute a *TLB Write Conditional* to create a TLB entry for the virtual page V, at most one of these ***tlbwe*** instructions succeeds.

If, after thread P1 establishes a TLB-reservation for a virtual address in a virtual page V, another thread P2 executes a ***tlbivax*** that invalidates a TLB entry for the virtual page V and thread P1 does a *TLB Write Conditional* to create a TLB entry for the virtual page V, then one of the following occurs.

- The TLB invalidation occurs before the TLB write. Thus the TLB-reservation is lost and the *TLB Write Conditional* does not succeed.
- The TLB write occurs before the TLB invalidation. Thus the *TLB Write Conditional* succeeds and the resulting TLB entry created by the ***tlbwe*** is invalidated by the ***tlbivax***.

Forward progress

Forward progress in loops that use ***tlbsrx.*** <E.TWC> and ***tlbwe*** with $MAS0_{WQ}$ = 0b01 is achieved by a cooperative effort among hardware and system software.

The architecture guarantees that when a thread executes a ***tlbsrx.*** <E.TWC> to set a TLB-reservation for virtual address X and then a *TLB Write Conditional* to write a TLB entry, either

1. the *TLB Write Conditional* succeeds and the TLB entry is written, or
2. the *TLB Write Conditional* fails because the TLB-reservation was reset because some other thread invalidated all TLB entries in the system for the virtual page containing the virtual address X or some other thread wrote a shared TLB entry for the virtual page containing the virtual address X, or
3. the *TLB Write Conditional* fails because the thread's TLB-reservation was lost for some other reason.

In Case 1 forward progress is made in the sense that the thread successfully wrote the TLB entry. In Case 2, the system as a whole makes progress in the sense that either some thread successfully invalidated TLB entries for virtual address X or some thread that shares the TLB wrote a TLB entry for the virtual page containing virtual address X. Case 3 covers TLB-reservation loss required for correct operation of the rest of the system. This includes TLB-reservation loss caused by some other thread invalidating all entries in a shared TLB, as well as TLB-reservation loss caused by system software invalidating all entries for the PID value associated with virtual address X. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of TLB-reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

Programming Note

The architecture does not include a "fairness guarantee". In competing for a TLB-reservation, two threads can indefinitely lock out a third.

6.11.4.3 Invalidating TLB Entries

TLB entries may be invalidated by three different methods or if the Embedded.Hypervisor category is supported, by four different methods.

- The TLB entry can be invalidated as the result of a ***tlbwe*** instruction that sets the $MAS1_V$ bit in the entry to 0.
- TLB entries may be invalidated as a result of a ***tlbivax*** instruction or from an invalidation resulting from a ***tlbivax*** on another thread.
- TLB entries may be invalidated as a result of an invalidate all operation specified through appropriate settings in the MMUCSR0.
- If the Embedded.Hypervisor category is supported, TLB entries may be invalidated as a result of a ***tlbilx*** instruction.

See Section 6.11.4.4 for the effects of the above methods on TLB lookaside information.

In systems consisting of a single-threaded processor as well as in systems consisting of multi-threaded processors, invalidations can occur on a wider set of TLB entries than intended. That is, a virtual address presented for invalidation may cause not only the intended TLB targeted for invalidation to be invalidated, but may also invalidate other TLB entries depending on the implementation. This is because parts of the translation mechanism may not be fully specified to the hardware at invalidate time. This is especially true in SMP systems, where the invalidation address must be supplied to all threads in the system, and there may be other limitations imposed by the hardware implementation. This phenomenon is known as generous invalidates. The architecture assures that the intended TLB will be invalidated, but does not guarantee that it will be the only one. A TLB entry invalidated by writing the V bit of the TLB entry to 0 by use of a ***tlbwe*** instruction is guaranteed to invalidate only the selected TLB entry. Invalidates occurring from ***tlbilx*** or ***tlbivax*** instructions or from ***tlbivax*** instructions on another thread may cause generous invalidates.

The architecture provides a method to protect against generous invalidations. This is important since there are certain virtual memory regions that must be properly mapped to make forward progress. To prevent this, the architecture specifies an IPROT bit for TLB entries. If the IPROT bit is set to 1 in a given TLB entry, that entry is protected from invalidations resulting from ***tlbilx*** <E.HV> and ***tlbivax*** instructions, or from invalidate all operations. TLB entries with the IPROT field set may only be invalidated by explicitly writing the TLB entry and specifying a 0 for the V ($MAS1_V$) field. This does not preclude the possibility that a TLB entry with the IPROT field set can be replaced by a ***tlbwe*** executing with hypervisor privilege when $MAS0_{HES}$=1. A subsequent ***tlbivax*** or ***tlbilx*** can then invalidate the replaced TLB entry.

To invalidate one or more individual virtual pages from all TLB arrays in all threads without the involvement of software running on other threads, software can execute the following sequence of instructions.

one or more ***tlbivax*** instructions
mbar or ***sync***
tlbsync
sync

Programming Note

Implementations are permitted to have a restriction on the number of threads doing a ***tlbivax*-*mbar*/*sync*-*tlbsync*-*sync*** sequence. This restriction could be imposed by the system or the hardware.

Other instructions, excluding ***tlbivax***, may be interleaved with the instruction sequence shown above, but the instructions in the sequence must appear in the order shown. On systems consisting of only a single-threaded processor or on systems where every thread shares every TLB, the ***tlbsync*** and the preceding ***mbar*** or ***sync*** can be omitted.

Programming Note

For the preceding instruction sequence, the ***mbar*** or first ***sync*** instruction prevents the reordering of ***tlbivax*** instructions previously executed by the thread with respect to the subsequent ***tlbsync*** instruction. The ***tlbsync*** instruction and the subsequent ***sync*** instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required and Alternate Coherency Mode attributes, before any data accesses caused by instructions following the ***sync*** instruction are performed with respect to that thread or mechanism.

Programming Note

The most obvious issue with generous invalidations is the code memory region that serves as the exception handler for MMU faults. If this region does not have a valid mapping, an MMU exception cannot be handled because the first address of the exception handler will result in another MMU exception.

Programming Note

Not all TLB arrays in a given implementation will implement the IPROT attribute. It is likely that implementations that are suitable for demand page environments will implement it for only a single array, while not implementing it for other TLB arrays.

Programming Note

Operating systems and hypervisors need to use great care when using protected (IPROT) TLB entries, particularly in SMP systems. A system that contains TLB entries on other threads will require a cross thread interrupt or some other synchronization mechanism to assure that each thread performs the required invalidation by writing its own TLB entries.

Programming Note

For MMU Architecture Version 1.0, to ensure a TLB entry that is not protected by IPROT is invalidated if software does not know which TLB array the entry is in, software should issue a ***tlbivax*** instruction targeting each TLB in the implementation with the EA to be invalidated.

Programming Note

The preferred method of invalidating entire TLB arrays is invalidation using MMUCSR0, however ***tlbilx*** may be more efficient.

Programming Note

Invalidations using MMUCSR0 only affect the TLB array on the thread that performs the invalidation. To perform invalidations on all threads in a coherence domain on a multi-threaded processor or on a system containing multiple single-threaded processors, software should use ***tlbivax***. If a large number of TLB entries need to be invalidated, using MMUCSR0 or, if the Embedded.Hypervisor category is supported, ***tlbilx***, on each thread may be more efficient.

Programming Note

Since a hardware replacement algorithm selects the entry for a ***tlbwe*** instruction with $MAS0_{HES}$ = 1, it is typically not possible to invalidate the entry using a second ***tlbwe*** instruction with $MAS0_{HES}$ = 1 and $MAS1_{V}$ = 0. If software needs to invalidate a single entry that was written with $MAS0_{HES}$ = 1, software should generally invalidate the entry using ***tlbilx*** with T=3 or ***tlbivax***.

6.11.4.4 TLB Lookaside Information

For performance reasons, most implementations also have implementation-specific lookaside information that is used in address translation. This lookaside information is a cache of recently used TLB entries.

If $TLBnCFG_{HES}$=0, lookaside information for the associated TLB array is kept coherent with the TLB and is invisible to software. Any write to the TLB array that displaces or updates an entry will be reflected in the

lookaside information, invalidating the lookaside information corresponding to the previous TLB entry. Any type of invalidation of an entry in TLB will also invalidate the corresponding entry in the lookaside information.

If $TLBnCFG_{HES}$=1, lookaside information for the associated TLB array is not required to be kept coherent with the TLB. Only in the following conditions will the lookaside information be kept coherent with the TLB. The MMUCSR0 TLB invalidate all will invalidate all lookaside information. The ***tlbilx*** and ***tlbivax*** instructions invalidate lookaside information corresponding to TLB entry values that they are specified to invalidate as well as those TLB entry values that would have been invalidated except for their IPROT=1 value.

The same instructions that synchronize invalidations of TLB entries also synchronize invalidation of TLB lookaside information.

Programming Note

If $TLBnCFG_{HES}$=1 for a TLB array and it is important that the lookaside information corresponding to a TLB entry be invalidated, software should use ***tlbilx*** or ***tlbivax*** to invalidate the virtual address.

6.11.4.5 Invalidating LRAT Entries

There is only one mechanism for invalidating LRAT entries. An LRAT entry can be invalidated as the result of a ***tlbwe*** instruction that overwrites the LRAT entry with a new valid entry or that sets $LRAT_V$ = 0. Only one LRAT entry is invalidated by a single ***tlbwe***.

6.11.4.6 Searching TLB Entries

Software may search the MMU by using the ***tlbsx*** instruction, and, if Category: TLB Write Conditional category is supported, the ***tlbsrx.*** <E.TWC> instruction. The ***tlbsrx.*** <E.TWC> and ***tlbsx*** instructions use IND, PID, and AS values from the MAS registers instead of the PID registers and the MSR, and, if the Embedded.Hypervisor category is supported, these instructions use an LPID and GS value from the MAS registers instead LPIDR and MSR. This allows software to search address spaces that differ from the current address space defined by the PID registers. This is useful for TLB fault handling.

6.11.4.7 TLB Replacement Hardware Assist

The architecture provides mechanisms to assist software in creating and updating TLB entries when certain MMU related exceptions occur. This is called TLB Replacement Hardware Assist. Hardware will update the MAS registers on the occurrence of a Data TLB Error Interrupt or Instruction TLB Error interrupt if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor ($EPCR_{DMIUH}$ = 0), or the interrupt is directed to the guest state.

When a Data or Instruction TLB Error interrupt (TLB miss) occurs and if the Embedded.Hypervisor category is not supported, MAS Register updates are enabled for interrupts directed to the hypervisor ($EPCR_{DMIUH}$ = 0), or the interrupt is directed to the guest state, then MAS0, MAS1, and MAS2 are automatically updated using the defaults specified in MAS4 as well as the AS and EPN values corresponding to the access that caused the exception. MAS6 is updated to set $MAS6_{SPID}$ to the value of PID ($EPLC_{EPID}$ for *External PID Load* instructions or $EPSC_{EPID}$ for *External PID Store* instructions), $MAS6_{SIND}$ to the value of $MAS4_{INDD}$, and $MAS6_{SAS}$ to the value of MSR_{DS} or MSR_{IS} depending on the type of access (data or instruction) that caused the error. In addition, if $MAS4_{TLBSELD}$ identifies a TLB array that supports NV (Next Victim), $MAS0_{ESEL}$ is loaded with a value that hardware predicts represents the best TLB entry to victimize to create a new TLB entry and $MAS0_{NV}$ is updated with the TLB entry index of what hardware predicts to be the next victim for the set of entries which can be used for translating addresses with the EPN that caused the exception. Thus $MAS0_{ESEL}$ identifies the current TLB entry to be replaced, and $MAS0_{NV}$ points to the next victim. When software writes the TLB entry, the $MAS0_{NV}$ field is written to the TLB array's set of next victim values. The algorithm used by the hardware to determine which TLB entry should be targeted for replacement is implementation-dependent.

Next Victim support is provided for TLB arrays that are set associative and that have $TLBnCFG_{HES}$=0. Next Victim support is not provided for TLB arrays that are fully associative.

The automatic update of the MAS registers sets up all the necessary fields for creating a new TLB entry with the exception of RPN, the U0-U3 attribute bits, and the permission bits. With the exception of the upper 32 bits of RPN and the page attributes (should software desire to specify changes from the default attributes), all the remaining fields are located in MAS3, requiring only the single MAS register manipulation by software before writing the TLB entry.

For Instruction Storage interrupt (ISI) and Data Storage interrupt (DSI) related exceptions, the MAS registers are not updated. Software must explicitly search the TLB to find the appropriate entry.

The update of MAS registers through TLB Replacement Hardware Assist is summarized in Table 11 on page 1116.

Programming Note

Next Victim support is not provided for a fully associative array because such an array is intended for mostly static mappings of addresses.

6.11.4.8 32-bit and 64-bit Specific MMU Behavior

MMU behavior is largely unaffected by whether the thread is in 32-bit computation mode (MSR_{CM}=0) or 64-bit computation mode (MSR_{CM}=1). The only differences occur in the EPN field of the TLB entry and the EPN field of MAS2. The differences are summarized here.

- Executing a ***tlbwe*** instruction in 32-bit mode will set bits 0:31 of the TLB EPN field to zero unless $MAS0_{ATSEL}$ is set, in which case those bits are not written to zero.
- For an update to MAS registers via TLB Replacement Hardware Assist (see Section 6.11.4.7), an update to bits 0:53 of the EPN field occurs regardless of the computation mode of the thread at the time of the exception or the interrupt computation mode in which the interrupt is taken. If the instruction causing the exception was executing in 32-bit mode, bits 0:31 of the EPN field in MAS2 will be set to 0.
- Executing a ***tlbre*** instruction in 32-bit mode will set bits 0:31 of the MAS2 EPN field to an undefined value.
- In 32-bit implementations, MAS2U can be used to read or write $EPN_{0:31}$ of MAS2.

Programming Note

This allows a 32-bit OS to operate seamlessly in 32-bit mode on a 64-bit implementation and a 64-bit OS to easily support 32-bit applications.

6.11.4.9 TLB Management Instructions

The ***tlbivax*** instruction is used to invalidate TLB entries. Additional instructions are used to read and write, and search TLB entries, and to provide an ordering function for the effects of ***tlbivax***. If the Embedded.Hypervisor category is supported, the ***tlbilx*** instruction is used to invalidate TLB entries in the thread executing the ***tlbilx***.

TLB Invalidate Virtual Address Indexed X-form

tlbivax RA,RB

31	///	RA	RB	786	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
for each thread
  if MAV = 1.0 then for TLB array = EA59:60
  if MAV = 2.0 then for each TLB array
    for each TLB entry
      if MMUCFGTWC = 1 or TLBnCFGHES = 1 then
        c ← MAS6ISIZE
      else
        c ← entrySIZE
      if MAV = 1.0 then
         m ← ¬((1 << (2 × c)) - 1)
      else
         m ← ¬((1 << c) - 1)
      if ((EA0:53 & m) = (entryEPN & m)) &
       entrySIZE = MAS6ISIZE &
        entryTID = MAS6SPID & entryTS = MAS6SAS &
       (E.PT not supported | entryIND = MAS6SIND) &
        (E.HV not supported |
         (entryTLPID = MAS5SLPID &
          entryTGS = MAS5SGS)) |
        ((MAV = 1.0) & (EA61 = 1))
        then
          if entryIPROT = 0 then entryV ← 0
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The EA is interpreted as show below.

$EA_{0:53}$ $EA_{0:53}$

$EA_{54:58}$ Reserved

$EA_{59:60}$ TLB array selector [MAV = 1.0]

00 TLB0
01 TLB1
10 TLB2
11 TLB3

EA_{61} TLB invalidate all [MAV = 1.0]

$EA_{62:63}$ Reserved

If EA_{61}=0, all TLB entries on all threads that have all of the following properties are made invalid. The MAS registers listed are those in the thread executing the ***tlbivax***.

- The MMU architecture version is 2.0 or the entry is in the TLB array targeted by $EA_{59:60}$.
- The logical AND of $EA_{0:53}$ and m is equal to the logical AND of the EPN value of the TLB entry and m, where m is based on the following.
  - If $MMUCFG_{TWC}$ = 1 or $TLBnCFG_{HES}$ = 1, c is equal $MAS6_{ISIZE}$. Otherwise, c is equal to $entry_{SIZE}$.
  - If MMU Architecture Version 1.0 is supported, m is equal to the logical NOT of ((1 << (2 × c)) - 1). Otherwise, m is equal to the logical NOT of ((1 << c) - 1).
- The TID value of the TLB entry is equal to $MAS6_{SPID}$ and the TS value of the TLB entry is equal to $MAS6_{SAS}$.
- The implementation does not support the Embedded.Page Table category or the IND value of the TLB entry is equal to $MAS6_{SIND}$.
- Either of the following is true:
  - The implementation does not support the Embedded.Hypervisor category.
  - The TLPID value of the TLB entry is equal to $MAS5_{SLPID}$ and the TGS value of the TLB entry is equal to $MAS5_{SGS}$.
- $entry_{IPROT}$ = 0.

In MMU Architecture Version 1.0 if EA_{61}=1, all entries in all threads not protected by the IPROT attribute in the TLB array targeted by $EA_{59:60}$ are made invalid.

If the instruction specifies a TLB array that does not exist, the instruction is treated as if the instruction form is invalid. If the implementation requires the page size to be specified by $MAS6_{ISIZE}$ ($MMUCFG_{TWC}$ = 1 or, for the specified TLB array, $TLBnCFG_{HES}$ = 1) and the page size specified by $MAS6_{ISIZE}$ is not supported by the implementation, the instruction is treated as if the instruction form is invalid.

If the operation isn't a TLB invalidate all and there are multiple entries in a single thread's TLB array(s) that match the complete VPN, then zero or more matching entries with IPROT=0 are invalidated or a Machine Check interrupt occurs. If the Embedded.Hypervisor category is supported, this Machine Check interrupt must be precise.

The operation performed by this instruction is ordered by the ***mbar*** (or ***sync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the thread executing the ***tlbivax*** instruction. The operations caused by ***tlbivax*** and ***tlbsync*** are ordered by ***mbar*** as a set of operations which is independent of the other sets that ***mbar*** orders.

The effects of the invalidation on this thread are not guaranteed to be visible to the programming model

until the completion of a context synchronizing operation.

Invalidations may occur for other TLB entries in the designated array, but in no case will any TLB entries with the IPROT attribute set be made invalid.

If RA does not equal 0, it is implementation-dependent whether an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

Special Registers Altered:

None

Programming Note

Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

For backward compatibility, implementations may ignore a TLB entry's TS and TID fields when determining whether an entry should be invalidated. Since this and other such generous invalidation can be performed, consideration should be given to protecting a TLB entry that maps an interrupt vector by setting TLB_{IPROT} =1.

Programming Note

The ***tlbilx*** instruction is the preferred way of performing TLB invalidations for operating systems running as a guest to the hypervisor since the invalidations are partitioned and do not require hypervisor privilege.

Programming Note

The TLB invalidate all function (EA_{61}=1) only exists in MMU Architecture Version 1.0 implementations. It should only be used when running existing software is deemed important.

TLB Invalidate Local Indexed

X-form

tlbilx RA,RB [Category: Embedded.Phased In]]

31	///	T	RA	RB	18	/
0	6	9	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
for each TLB array
  for each TLB entry
    if MMUCFG_TWC = 1 or TLBnCFG_HES = 1 then
      c ← MAS6_ISIZE
    else
      c ← entry_SIZE
    if MAV = 1.0 then
      m ← ¬((1 << (2 × c)) - 1)
    else
      m ← ¬((1 << c) - 1)
    if (entry_IPROT = 0) & (entry_TLPID = MAS5_SLPID) then
      if T = 0 then entry_V ← 0
      if T = 1 & entry_TID = MAS6_SPID then entry_V ← 0
      if T = 3 & entry_TGS = MAS5_SGS &
         ((EA_0:53 & m) = (entry_EPN & m)) &
         entry_SIZE = MAS6_ISIZE &
         entry_TID = MAS6_SPID & entry_TS = MAS6_SAS &
        (E.PT not supported | entry_IND = MAS6_SIND)
        then
          entry_V ← 0
```

Let the effective address (EA) be the sum (RA|0) + (RB).

The **tlbilx** instruction invalidates TLB entries in the thread that executes the **tlbilx** instruction. TLB entries which are protected by the IPROT attribute ($entry_{IPROT}$ = 1) are not invalidated.

If T = 0, all TLB entries that have all of the following properties are made invalid on the thread executing the ***tlbilx*** instruction.

- The TLPID of the entry matches $MAS5_{SLPID}$.
- The IPROT of entry is 0.

If T = 1, all TLB entries that have all of the following properties are made invalid on the thread executing the ***tlbilx*** instruction.

- The TLPID of the entry matches $MAS5_{SLPID}$.
- The TID of the entry matches $MAS6_{SPID}$.
- The IPROT of entry is 0.

If T = 3, all TLB entries in the thread executing the ***tlbilx*** instruction that have all of the following properties are made invalid.

- The TLPID value of the TLB entry is equal to $MAS5_{SLPID}$ and the TGS value of the TLB entry is equal to $MAS5_{SGS}$.
- The logical AND of $EA_{0:53}$ and m is equal to the logical AND of the EPN value of the TLB entry and m, where m is based on the following.
  - If $MMUCFG_{TWC}$ = 1 or $TLBnCFG_{HES}$ = 1, c is equal $MAS6_{ISIZE}$. Otherwise, c is equal to $entry_{SIZE}$.
  - If MMU Architecture Version 1.0 is supported, m is equal to the logical NOT of ((1 << (2 × c)) - 1). Otherwise, m is equal to the logical NOT of ((1 << c) - 1).
- The TID value of the TLB entry is equal to $MAS6_{SPID}$ and the TS value of the TLB entry is equal to $MAS6_{SAS}$.
- The implementation does not support the Embedded.Page Table category or the IND value of the TLB entry is equal to $MAS6_{SIND}$.
- The IPROT of entry is 0.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation.

Invalidations may occur for other TLB entries on the thread executing the **tlbilx** instruction, but in no case will any TLB entries with the IPROT attribute set be made invalid.

If T = 2, the instruction form is invalid.

If T = 3 and the implementation requires the page size to be specified by $MAS6_{ISIZE}$ ($MMUCFG_{TWC}$ = 1 or, for any TLB array, $TLBnCFG_{HES}$ = 1) and the page size specified by $MAS6_{ISIZE}$ is not supported by the implementation, the instruction is treated as if the instruction form is invalid.

If T=3 and there are multiple entries in the TLB array(s) that match the complete VPN, then zero or more matching entries with IPROT=0 are invalidated or a Machine Check interrupt occurs. If the Embedded.Hypervisor category is supported, this Machine Check interrupt must be precise.

If RA does not equal 0, it is implementation-dependent whether an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported and guest execution of *TLB Management* instructions is disabled ($EPCR_{DGTMI}$=1), this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *TLB Invalidate Local*:

Extended:		Equivalent to:	
tlbilxlpid		tlbilx	0,0,0
tlbilxpid		tlbilx	1,0,0
tlbilxva	RA,RB	tlbilx	3,RA,RB
tlbilxva	RB	tlbilx	3,0,RB

Programming Note

tlbilx is the preferred way of performing TLB invalidations, especially for operating systems running as a guest to the hypervisor since the invalidations are partitioned and do not require hypervisor privilege.

Programming Note

When dispatching a guest operating system, hypervisor software should always set $MAS5_{SLPID}$ to the guest's corresponding LPID value.

Programming Note

Executing a ***tlbilx*** instruction with T=0 or T=1 may take many cycles to perform. Software should only issue these operations when an LPID or a PID value is reused or taken out of use.

TLB Search Indexed *X-form*

tlbsx RA,RB

31	///	RA	RB	914	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
Valid_matching_entry_exists ← 0
for each TLB array
  for each TLB entry
    if MAV = 1.0 then
      m ← ¬((1 << (2 × entrySIZE)) - 1)
    else
      m ← ¬((1 << entrySIZE) - 1)
    if ((EA0:53 & m) = (entryEPN & m)) &
       (entryTID = MAS6SPID | entryTID = 0) &
       entryTS = MAS6SAS &
       (E.PT not supported | entryIND = MAS6SIND) &
       (E.HV not supported | (entryTGS = MAS5SGS &
      (entryTLPID = MAS5SLPID | entryTLPID = 0))) then
      Valid_matching_entry_exists ← 1
      exit for loops
if Valid_matching_entry_exists
  entry ← matching entry found
  array ← TLB array number where TLB entry found
  index ← index into TLB array of TLB entry found
  if TLB array supports Next Victim then
    hint ← hardware hint for Next Victim
  else
    hint ← undefined
  rpn ← entryRPN
  MAS0ATSEL ← 0
  MAS0TLBSEL ← array
  MAS0ESEL ← index
  if MAS0HES supported
    MAS0HES ← 0
  if Next Victim supported then
    if TLB array specified by MAS0TLBSEL supports NV
      then
        MAS0NV ← hint
      else
        MAS0NV ← undefined
  MAS1V ← 1
  MAS1TID TS TSIZE ← entryTID TS SIZE
  if TLB array supports IPROT then
    MAS1IPROT ← entryIPROT
  else
    MAS1IPROT ← 0
  if category E.PT supported then
    if TLB array supports indirect entries then
      MAS1IND ← entryIND
      if entryIND = 1
        MAS3SPSIZE ← entrySPSIZE
      else
        MAS3UX SX UW SW UR SR ← entryUX SX UW SW UR SR
    else
      MAS1IND ← 0
      MAS3UX SX UW SW UR SR ← entryUX SX UW SW UR SR
  else
    MAS3UX SX UW SW UR SR ← entryUX SX UW SW UR SR
  MAS2EPN W I M G E ← entryEPN W I M G E
  if category VLE supported then MAS2VLE ← entryVLE
  if ACM supported then MAS2ACM ← entryACM
  MAS3RPNL ← rpn32:53
  MAS3U0:U3 ← entryU0:U3
  MAS7RPNU ← rpn0:31
 if category E.HV supported then
   MAS8TGS VF TLPID ← entryTGS VF TLPID
else
  MAS0ATSEL ← 0
  MAS0TLBSEL ← MAS4TLBSELD
  if Next Victim supported then
    if TLB array specified by MAS4TLBSELD supports
      Next Victim then
        MAS0ESEL ← hint
        MAS0NV ← hint for next replacement
      else
        MAS0ESEL ← undefined
        MAS0NV ← undefined
  else
    MAS0ESEL ← undefined
  if MAS0HES supported
    MAS0HES ← TLBnCFGHES for the TLB array specified
              by MAS4TLBSELD
  MAS1V IPROT ← 0
  MAS1TID TS ← MAS6SPID SAS
  MAS1TSIZE ← MAS4TSIZED
  if Embedded.Page Table category supported then
    MAS1IND ← MAS4INDD
  MAS2W I M G E ← MAS4WD ID MD GD ED
  if category VLE supported then MAS2VLE ← MAS4VLED
  if ACM supported, then MAS2ACM ← MAS4ACMD
  MAS2EPN ← undefined
  MAS3RPNL ← 0
  MAS3U0:U3 UX SX UW SW UR SR ← 0
  MAS7RPNU ← 0
if category E.TWC supported then MAS0WQ← 0b01
```

Let the effective address (EA) be the sum (RAl0)+ (RB).

If any TLB array contains a valid entry matching the $MAS1_{IND}$ <E.PT> and virtual address formed by $MAS5_{SGS}$ <E.HV>, $MAS5_{SLPID}$ <E.HV>, $MAS1_{TS\ TID}$, and EA, the search is considered successful. A TLB entry matches if all the following conditions are met.

- The valid bit of the TLB entry is 1.
- The logical AND of $EA_{0:53}$ and m is equal to the logical AND of the EPN value of the TLB entry and m, where m is determined as follows:
  - If MMU Architecture Version 1.0 is supported, m is equal to the logical NOT of ((1 << (2 × $entry_{SIZE}$)) - 1). Otherwise, m is equal to the logical NOT of ((1 << $entry_{SIZE}$) - 1)
- The TID value of the TLB entry is equal to $MAS6_{SPID}$ or is zero.
- The TS value of the TLB entry is equal to $MAS6_{SAS}$.
- Either the Embedded.Page Table category is not supported or the IND value of the TLB entry is equal to $MAS6_{SIND}$.
- Either of the following is true:
  - The implementation does not support the Embedded.Hypervisor category.
  - The TGS value of the TLB entry is equal to $MAS5_{SGS}$ and either the TLPID value of the TLB entry is equal to $MAS5_{SLPID}$ or is zero.

If the search is successful, MAS register fields are loaded from the matching TLB entry according to the following.

- $MAS0_{ATSEL}$ is set to 0.
- $MAS0_{TLBSEL}$ is set to the number of the TLB array with the matching entry.
- $MAS0_{ESEL}$ is set to the index of the matching entry.
- If $MAS0_{HES}$ is supported, $MAS0_{HES}$ is set to 0.
- If Next Victim is supported for any TLB array, the following applies.
  - If the TLB array with the matching entry supports Next Victim, $MAS0_{NV}$ is $MAS0_{NV}$ is set to the hardware hint for the index of the entry to be replaced. Otherwise, $MAS0_{NV}$ is set to an implementation-dependent undefined value.
- $MAS1_{V}$ is set to 1.
- $MAS1_{TID\ TS\ TSIZE}$ are loaded from the TID, TS, and SIZE fields of the TLB entry.
- If the TLB array supports IPROT, $MAS1_{IPROT}$ is loaded from the IPROT bit of the TLB entry. Otherwise, $MAS1_{IPROT}$ is set to 0.
- $MAS2_{EPN\ W\ I\ M\ G\ E}$ are loaded from the EPN, W, I, M, G, and E fields of the TLB entry.
- If the VLE category is supported, $MAS2_{VLE}$ is loaded from the VLE bit of the TLB entry.
- If Alternate Coherency Mode is supported, $MAS2_{ACM}$ is loaded from the ACM bit of the TLB entry.
- $MAS3_{RPNL}$ is loaded from the lower 22-bits of the RPN field of the TLB entry, and, if implemented, $MAS7_{RPNU}$ is loaded from the upper 32-bits of the RPN field of the TLB entry.
- The supported User-Defined storage control bits in $MAS3_{U0:U3}$ are loaded from the respective supported U0:U3 bits of the TLB entry.
- If the Embedded.Page Table category is not supported, $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry. Otherwise, the following applies.
  - if the TLB array does not support indirect entries, $MAS1_{IND}$ is set to 0 and $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry. Otherwise, the following applies.
    - $MAS1_{IND}$ is loaded from the IND bit of the TLB entry.
    - If the IND bit of the TLB entry is 1, $MAS3_{SPSIZE}$ is loaded from the SPSIZE field of the TLB entry, and $MAS3_{UND}$ is set to an implementation-dependent undefined value.
    - If the IND bit of the TLB entry is 0, $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry.
- If the Embedded.Hypervisor category is implemented, $MAS8_{TGS\ VF\ TLPID}$ are loaded from the TGS, VF, and TLPID fields of the TLB entry.

If no valid matching translation exists, $MAS1_{V}$ is set to 0 and the MAS register fields are loaded according to the following in order to facilitate a TLB replacement.

- $MAS0_{ATSEL}$ is set to 0.
- $MAS0_{TLBSEL}$ is loaded from $MAS4_{TLBSELD}$.
- If Next Victim is not supported for any TLB array, $MAS0_{ESEL}$ is set to an implementation-dependent undefined value. Otherwise, the following applies.
  - If the TLB array specified by $MAS4_{TLBSELD}$ supports Next Victim, $MAS0_{ESEL}$ is set to the hardware hint for the index of the entry to be replaced and $MAS0_{NV}$ is set to the hardware hint for the index of the next entry to be replaced. Otherwise, $MAS0_{ESEL}$ and $MAS0_{NV}$ are set to implementation-dependent undefined values.
- If $MAS0_{HES}$ is supported, $MAS0_{HES}$ is set to the value of $TLBnCFG_{HES}$ for the TLB array specified by $MAS4_{TLBSELD}$.
- $MAS1_{IPROT}$ is set to 0.
- $MAS1_{TID\ TS}$ are loaded from $MAS6_{SPID\ SAS}$.
- $MAS1_{TSIZE}$ is loaded from $MAS4_{TSIZED}$.
- If the Embedded.Page Table category is supported, $MAS1_{IND}$ is set to $MAS4_{INDD}$.
- $MAS2_{EPN}$ is set to an implementation-dependent undefined value.
- $MAS2_{W\ I\ M\ G\ E}$ are loaded from $MAS4_{WD\ ID\ MD\ GD\ ED}$.
- If the VLE category is supported, $MAS2_{VLE}$ is loaded from $MAS4_{VLED}$.
- If Alternate Coherency Mode is supported, $MAS2_{ACM}$ is loaded from $MAS4_{ACMD}$.
- $MAS3_{RPNL}$ and, if implemented, $MAS7_{RPNU}$ are set to 0s.
- The supported User-Defined storage control bits bits in $MAS_{U0:U3}$ are set to 0s.
- $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are set to 0s.

If the Embedded.TLB Write Conditional category is supported, $MAS0_{WQ}$ is set to 0b01.

If a ***tlbsx*** is successful, it is considered to “hit”. Otherwise, it is considered to “miss”.

If there are multiple matching TLB entries, either one of the matching entries is used or a Machine Check exception occurs. If the Embedded.Hypervisor category is supported, this Machine Check interrupt must be precise.

If RA does not equal zero, it is implementation-dependent whether an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

Special Registers Altered:

MAS0 MAS1 MAS2 MAS3 MAS7
MAS8 (if category E.HV supported)

TLB Search and Reserve Indexed X-form

tlbsrx. RA,RB [Category: Embedded.TLB Write Conditional]

31	///	RA	RB	850	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
pid ← MAS1_TID
as ← MAS1_TS
if Embedded.Page Table category supported then
  ind ← MAS1_IND
if category E.HV supported then
  gs ← MAS5_SGS
  lpid ← MAS5_SLPID
  va ← gs || lpid || as || pid || EA
else
  va ← as || pid || EA
TLB-RESERVE ← 1
if Embedded.Page Table category supported then
  TLB-RESERVE_IND_N_ADDR ← ind || va
else
  TLB-RESERVE_ADDR ← va
Valid_matching_entry_exists ← 0
for each TLB array
  for each TLB entry
    if MAV = 1.0 then
      m ← ¬((1 << (2 × entry_SIZE)) - 1)
    else
      m ← ¬((1 << entry_SIZE) - 1)
    if ((EA_0:53 & m) = (entry_EPN & m)) &
       (entry_TID = MAS1_TID | entry_TID = 0) &
       entry_TS = MAS1_TS &
       (E.PT not supported | entry_IND = MAS1_IND) &
       (E.HV not supported | (entry_TGS = MAS5_SGS &
      (entry_TLPID = MAS5_SLPID | entry_TLPID = 0))) then
      Valid_matching_entry_exists ← 1
      exit for loops
if Valid_matching_entry_exists then
  CR0 ← 0b0010
else
  CR0 ← 0b0000
```

Let the effective address (EA) be the sum (RAl0)+ (RB).

If any TLB array contains a valid entry matching the $MAS1_{IND}$ <E.PT> and virtual address formed by $MAS5_{SGS}$ <E.HV>, $MAS5_{SLPID}$ <E.HV>, $MAS1_{TS\ TID}$, and EA, the search is considered successful. A TLB entry matches if all the following conditions are met.

- The valid bit of the TLB entry is 1.
- Either the Embedded.Page Table category is not supported or the IND value of the TLB entry is equal to $MAS1_{IND}$.
- The logical AND of $EA_{0:53}$ and m is equal to the logical AND of the EPN value of the TLB entry and m, where m is determined as follows:
  - If MMU Architecture Version 1.0 is supported, m is equal to the logical NOT of ((1 << (2 × $entry_{SIZE}$)) - 1). Otherwise, m is equal to the logical NOT of ((1 << $entry_{SIZE}$) - 1)
- The TID value of the TLB entry is equal to $MAS1_{TID}$ or is zero.
- The TS value of the TLB entry is equal to $MAS1_{TS}$.
- Either of the following is true:
  - The implementation does not support the Embedded.Hypervisor category.
  - The TGS value of the TLB entry is equal to $MAS5_{SGS}$ and either the TLPID value of the TLB entry is equal to $MAS5_{SLPID}$ or is zero.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the search was successful.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || n || 0

This instruction creates a TLB-reservation for use by a *TLB Write* instruction. The virtual address described above is associated with the TLB-reservation, and replaces any address previously associated with the TLB-reservation. (The TLB-reservation is created regardless of whether the search succeeds.)

If there are multiple matching TLB entries, either one of the matching entries is used or a Machine Check exception occurs. If the Embedded.Hypervisor category is supported, this Machine Check interrupt must be precise.

If RA does not equal zero, it is implementation-dependent whether an Illegal Instruction exception occurs.

If the Embedded.Hypervisor category is supported and guest execution of *TLB Management* instructions is disabled ($EPCR_{DGTMI}$=1), this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

Special Registers Altered:

CR0

TLB Read Entry *X-form*

tlbre

31	///	///	///	946	/
0	6	11	16	21	31

```
if MAS0_ATSEL= 0 then
  if TLBnCFG_HES = 0 then
    entry ← SelectTLB(MAS0_TLBSEL,MAS0_ESEL, MAS2_EPN)
  else
    entry ← SelectTLB(MAS0_TLBSEL, MAS1_TID TSIZE,
         MAS2_EPN, MAS0_ESEL)
  if Next Victim supported then
    if TLB array specified by MAS0_TLBSEL supports NV
      then
        MAS0_NV ← hint
      else
        MAS0_NV ← undefined
  if TLB entry is found then
    rpn ← entry_RPN
    MAS1_V TID TS TSIZE ← entry_V TID TS SIZE
    if TLB array supports IPROT then
      MAS1_IPROT ← entry_IPROT
    else
      MAS1_IPROT ← 0
    if category E.PT supported then
      if TLB array supports indirect entries then
        MAS1_IND ← entry_IND
        if entry_IND = 1
          MAS3_SPSIZE ← entry_SPSIZE
        else
          MAS3_UX SX UW SW UR SR ← entry_UX SX UW SW UR SR
      else
        MAS1_IND ← 0
        MAS3_UX SX UW SW UR SR ← entry_UX SX UW SW UR SR
    else
      MAS3_UX SX UW SW UR SR ← entry_UX SX UW SW UR SR
    MAS2_EPN W I M G E ← entry_EPN W I M G E
    if category VLE supported then MAS2_VLE←entry_VLE
    if ACM supported then MAS2_ACM ← entry_ACM
    MAS3_RPNL ← rpn_32:53
    MAS3_U0:U3 ← entry_U0:U3
    MAS7_RPNU ← rpn_0:31
    if category E.HV supported then
      MAS8_TGS VF TLPID ← entry_TGS VF TLPID
  else
    MAS1_V ← 0
    MAS1_IPROT TID TS TSIZE ← undefined
    if Embedded.Page Table supported then
        MAS1_IND ← undefined
    MAS2_EPN W I M G E ← undefined
    if category VLE supported then
      MAS2_VLE ← undefined
    if ACM supported then MAS2_ACM ← undefined
    MAS3_RPNL U0:U3 UX SX UW SW UR SR ← undefined
    MAS7_RPNU ← undefined
    if category E.HV supported then
      MAS8_TGS VF TLPID ← undefined
else
  entry ← SelectLRAT(MAS0_ESEL, MAS2_EPN)
  MAS0_NV ← undefined
  if LRAT entry is found then
    rpn ← entry_LRPN
    MAS1_V TSIZE ← entry_V LSIZE
    MAS1_IPROT TID TS ← 0 0 0
    if Embedded.Page Table supported then
      MAS1_IND ← 0
    MAS2_EPN ← entry_LPN
    MAS2_W I M G E ← 0 0 0 0 0
    if category VLE supported then
      MAS2_VLE ← 0
    if ACM supported then MAS2_ACM ← 0
    MAS3_RPNL ← rpn_32:53
    MAS3_U0:U3 UX SX UW SW UR SR ← 0 0 0 0 0 0 0
    MAS7_RPNU ← rpn_0:31
    MAS8_TGS VF ← 0 0
    if the LRAT supports LPID
      MAS8_TLPID ← entry_LPID
    else
      MAS8_TLPID ← undefined
  else
    MAS1_V ← 0
    MAS1_IPROT TID TS TSIZE ← undefined
    if Embedded.Page Table supported then
      MAS1_IND ← undefined
    MAS2_EPN W I M G E ← undefined
    if category VLE supported then
      MAS2_VLE ← undefined
    if ACM supported then MAS2_ACM ← undefined
    MAS3_RPNL U0:U3 UX SX UW SW UR SR ← undefined
    MAS7_RPNU ← undefined
    MAS8_TGS VF TLPID ← undefined
```

If the Embedded.Hypervisor.LRAT category is not supported or $MAS0_{ATSEL}$ is 0, then the following applies.

- If Next Victim is supported for any TLB array, the following applies.
  - If the TLB array specified by $MAS0_{TLBSEL}$ supports Next Victim, $MAS0_{NV}$ is set to the hardware hint for the index of the entry to be replaced. Otherwise, $MAS0_{NV}$ is set to an implementation-dependent undefined value.
- A TLB entry is selected in one of the following ways.
  - if $TLBnCFG_{HES}$=0 for the TLB array selected by $MAS0_{TLBSEL}$, the TLB entry is specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$ and $MAS2_{EPN}$.
  - if $TLBnCFG_{HES}$=1 for the TLB array selected by $MAS0_{TLBSEL}$, the TLB entry is specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$ and a hardware generated hash based on $MAS1_{TID}$, $MAS1_{TSIZE}$, and $MAS2_{EPN}$.

If the selected TLB entry exists, MAS register fields are loaded according to the following.

- $MAS1_{V\ TID\ TS\ TSIZE}$ are loaded from the V, TID, TS, and SIZE fields of the TLB entry.
- If the TLB array supports IPROT, $MAS1_{IPROT}$ is loaded from the IPROT bit of the TLB entry. Otherwise, $MAS1_{IPROT}$ is set to 0.
- $MAS2_{EPN\ W\ I\ M\ G\ E}$ are loaded from the EPN, W, I, M, G, and E fields of the TLB entry.
- If the VLE category is supported, $MAS2_{VLE}$ is loaded from the VLE bit of the TLB entry.
- If Alternate Coherency Mode is supported, $MAS2_{ACM}$ is loaded from the ACM bit of the TLB entry.

- $MAS3_{RPNL}$ is loaded from the lower 22-bits of the RPN field of the TLB entry, and, if implemented, $MAS7_{RPNU}$ is loaded from the upper 32-bits of the RPN field of the TLB entry.
- The supported User-Defined storage control bits in $MAS3_{U0:U3}$ are loaded from the respective supported U0:U3 bits of the TLB entry.
- If the Embedded.Page Table category is not supported, $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry. Otherwise, the following applies.
  - if the TLB array does not support indirect entries, $MAS1_{IND}$ is set to 0 and $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry. Otherwise, the following applies.
    - $MAS1_{IND}$ is loaded from the IND bit of the TLB entry.
    - If the IND bit of the TLB entry is 1, $MAS3_{SPSIZE}$ is loaded from the SPSIZE field of the TLB entry, and $MAS3_{UND}$ is set to an implementation-dependent undefined value.
    - If the IND bit of the TLB entry is 0, $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are loaded from the UX, SX, UW, SW, UR, and SR bits of the TLB entry.
- If the Embedded.Hypervisor category is implemented, $MAS8_{TGS\ VF\ TLPID}$ are loaded from the TGS, VF, and TLPID fields of the TLB entry.

If the Embedded.Hypervisor.LRAT category is supported, the LRAT array is specified ($MAS0_{ATSEL}$ = 1), then the following applies.

- $MAS0_{NV}$ is set to an implementation-dependent undefined value.
- If the LRAT entry specified by $MAS0_{ESEL}$ and $MAS2_{EPN}$ exists, MAS register fields are loaded from the LRAT entry according to the following.
  - $MAS1_{V\ TSIZE}$ are loaded from the V and LSIZE fields of the LRAT entry.
  - $MAS1_{IPROT\ TID\ TS}$, $MAS2_{W\ I\ M\ G\ E}$, and $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ are set to 0s.
  - If the Embedded.Page Table category is supported, $MAS1_{IND}$ is set to 0.
  - $MAS2_{EPN}$ is loaded from the LPN field of the LRAT entry.
  - If the VLE category is supported, $MAS2_{VLE}$ is set to 0.
  - If Alternate Coherency Mode is supported, $MAS2_{ACM}$ is set to 0.
  - $MAS3_{RPNL}$ is loaded from the lower 22-bits of the LRPN field of the LRAT entry, and, if implemented, $MAS7_{RPNU}$ is loaded from the upper 32-bits of the LRPN field of the LRAT entry.
  - The supported User-Defined storage control bits in $MAS3_{U0:U3}$ are set to 0s.
  - $MAS8_{TGS\ VF}$ are set to 0s.
  - If the LPID field in the LRAT is supported ($LRATCFG_{LPID}$ = 1), $MAS8_{TLPID}$ is loaded from the TLPID field of the LRAT entry.

If $TLBnCFG_{HES}$ = 1 and the page size specified by $MAS1_{TSIZE}$ is not supported by the specified array, the ***tlbre*** may be performed as if TSIZE were some implementation-dependent value or, as described below, as if the entry can not be found, or an Illegal Instruction exception occurs.

It is implementation-dependent whether a TLB or LRAT entry can not be found or whether larger values of the fields that select an entry are simply mapped to existing entries. If the specified TLB or LRAT entry does not exist, $MAS1_{V}$ is set to 0 and the following MAS register fields are set to implementation-dependent undefined values.

- $MAS1_{IPROT\ TID\ TS\ TSIZE}$, $MAS2_{EPN\ W\ I\ M\ G\ E}$, $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$, $MAS3_{RPNL}$, and, if implemented, $MAS7_{RPNU}$
- If the VLE category is supported, $MAS2_{VLE}$
- If Alternate Coherency Mode is supported, $MAS2_{ACM}$
- The supported User-Defined storage control bits in $MAS3_{U0:U3}$
- If the Embedded.Page Table category is supported, $MAS1_{IND}$
- If the Embedded.Hypervisor category is implemented, $MAS8_{TGS\ VF\ TLPID}$

If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

Special Registers Altered:

MAS0 MAS1 MAS2 MAS3 MAS7
MAS8 (if category E.HV is supported)

Programming Note

Hypervisor software should generally prevent guest operating system visibility of the RPN. After executing a ***tlbsx*** or ***tlbre*** on behalf of a guest, the hypervisor should replace the RPN fields in the MAS3 and MAS7 registers with the corresponding values from the appropriate LPN.

TLB Synchronize X-form

tlbsync

31	///	///	///	566	/
0	6	11	16	21	31

The ***tlbsync*** instruction provides an ordering function for the effects of all ***tlbivax*** instructions executed by the thread executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***sync*** instruction executed by the same thread. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All TLB invalidations caused by ***tlbivax*** instructions preceding the ***tlbsync*** instruction will have completed on any other thread before any data accesses caused by instructions following the ***sync*** instruction are performed with respect to that thread.
- All storage accesses by other threads for which the address was translated using the translations being invalidated will have been performed with respect to the thread executing the ***sync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***sync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***mbar*** or ***sync*** instruction with respect to preceding ***tlbivax*** instructions executed by the thread executing the ***tlbsync*** instruction. The operations caused by ***tlbivax*** and ***tlbsync*** are ordered by ***mbar*** as a set of operations, which is independent of the other sets that ***mbar*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbivax*** instructions preceding the ***tlbsync*** instruction have been performed.

If the Embedded.Hypervisor category is supported, this instruction is hypervisor privileged. Otherwise, this instruction is privileged.

Special Registers Altered:

None

Programming Note

Care must be taken on some implementations when using the ***tlbsync*** instruction as there may be a system-imposed restriction of only one ***tlbsync*** allowed on the bus at a given time in the system.

TLB Write Entry X-form

tlbwe

31	///	///	///	978	/
0	6	11	16	21	31

```
if MAS0_WQ = 0b00 | MAS0_WQ = 0b01 then
  if MAS0_ATSEL = 0 or MSR_GS = 1 then
    if TLBnCFG_HES = 0 then
      entry ← SelectTLB(MAS0_TLBSEL,MAS0_ESEL, MAS2_EPN)
    else
      if MAS0_HES = 1 then
        entry ← SelectTLB(MAS0_TLBSEL, MAS1_TID TSIZE,
         MAS2_EPN, hardware_replacement_algorithm)
      else
        entry ← SelectTLB(MAS0_TLBSEL, MAS1_TID TSIZE,
         MAS2_EPN, MAS0_ESEL)
    if TLB array specified by MAS0_TLBSEL supports NV
      &((MAS0_WQ = 0b00) | (category E.TWC supported
      & (MAS0_WQ = 0b01) & (TLB-reservation))) then
        hint ← MAS0_NV
    if TLB entry is found &
      ((MAS0_WQ = 0b00) | ((category E.TWC supported)
      & (MAS0_WQ = 0b01) & (TLB-reservation))) then
    if category E.HV.LRAT supported & (MSR_GS=1) &
        (MAS1_V=1) then
          rpn ← translate_logical_to_real(MAS7_RPNU
            || MAS3_RPNL, MAS8_TLPID)
      else
        if MAS7 implemented then
          rpn ← MAS7_RPNU || MAS3_RPNL
        else rpn ← 32 0 || MAS3_RPNL
      entry_V IPROT TID TS SIZE ← MAS1_V IPROT TID TS TSIZE
      entry_EPN VLE W I M G E ACM ← MAS2_EPN VLE W I M G E ACM
      entry_U0:U3 ← MAS3_U0:U3
      if category E.PT supported and
        TLB array supports indirect entries then
          entry_IND ← MAS1_IND
          if MAS1_IND = 0 then
            entry_UX SX UW SW UR SR ← MAS3_UX SX UW SW UR SR
          else
            entry_SPSIZE ← MAS3_SPSIZE
        else
          entry_UX SX UW SW UR SR ← MAS3_UX SX UW SW UR SR
      entry_RPN ← rpn
      if (category E.HV is supported)
        entry_TGS VF TLPID ← MAS8_TGS VF TLPID
    if category E.TWC supported
      TLB-reservation ← 0
  else
    entry ← SelectLRAT(MAS0_ESEL,MAS2_EPN)
    if LRAT entry is found &
      (MAS0_WQ = 0b00) & (MAS0_HES = 0b0) then
      hint ← MAS0_NV
      entry_V LSIZE ← MAS1_V TSIZE
      entry_LPN ← MAS2_EPN
      entry_RPN ← MAS7_RPNU || MAS3_RPNL
      if LRATCFG_LPID = 1
        entry_LPID ← MAS8_TLPID
else
  if category E.TWC supported
    TLB-reservation ← 0
```

If the Embedded.TLB Write Conditional category is not supported, $MAS0_{WQ}$ is treated as if it were 0b00 in the following description.

If the Embedded.Hypervisor.LRAT category is not supported or MSR_{GS}=1, $MAS0_{ATSEL}$ is treated as if it were zero in the following description.

If a TLB array is specified ($MAS0_{ATSEL}$ = 0 or MSR_{GS} = 1) and $TLBnCFG_{HES}$=0 for the TLB array selected by $MAS0_{TLBSEL}$, $MAS0_{HES}$ is treated as 0 in the following description.

If the Embedded.Page Table category is supported, a TLB array is specified ($MAS0_{ATSEL}$ = 0 or MSR_{GS} = 1), and the specified TLB array does not support indirect entries, $MAS1_{IND}$ is treated as 0.

If a TLB array is specified ($MAS0_{ATSEL}$ = 0 or MSR_{GS} = 1) and $MAS0_{WQ}$ is 0b00 or 0b01, the following applies.

- If the Embedded.Hypervisor is not supported, the ***tlbwe*** instruction is executed in hypervisor state, or $MAS1_{V}$=0, an RPN is formed by concatenating $MAS7_{RPNU}$ with $MAS3_{RPNL}$ (RPN = $MAS7_{RPNU}$ || $MAS3_{RPNL}$).
- If the Embedded.Hypervisor category is supported, the ***tlbwe*** instruction is executed in guest state, and $MAS1_{V}$=1, an LPN is formed by concatenating $MAS7_{RPNU}$ with $MAS3_{RPNL}$ (LPN = $MAS7_{RPNU}$ || $MAS3_{RPNL}$). However, if MAS7 is not implemented, LPN = $^{32}0$ || $MAS3_{RPNL}$. This LPN is translated by the LRAT to obtain the RPN. If there is no LRAT entry that translates this LPN for the LPID specified by $MAS8_{TLPID}$, an LRAT Miss exception occurs. However, if $MAS0_{WQ}$ is 0b01 and no TLB-reservation exists, it is implementation-dependent whether the LRAT Miss exception occurs.
- If $TLBnCFG_{HES}$ for the TLB array selected by $MAS0_{TLBSEL}$ is 0, the TLB entry is specified by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$, and $MAS2_{EPN}$. If $TLBnCFG_{HES}$ is 1 and $MAS0_{HES}$ is 1, the TLB entry is selected by $MAS0_{TLBSEL}$, a hardware replacement algorithm, and a hardware generated hash based on $MAS1_{TID\ TSIZE}$, and $MAS2_{EPN}$. If $TLBnCFG_{HES}$ is 1 and $MAS0_{HES}$ is 0, the TLB entry is selected by $MAS0_{TLBSEL}$, $MAS0_{ESEL}$ and a hardware generated hash based on $MAS0_{TLBSEL}$, $MAS1_{TID\ TSIZE}$, and $MAS2_{EPN}$.
- The selected TLB entry is written (see the following major bulleted item) if all the following conditions are met.
  - There is no LRAT Miss exception.
  - $MAS0_{WQ}$ is 0b00 or both the following are true.
    - $MAS0_{WQ}$ is 0b01
    - A TLB-reservation exists.
  - $MAS1_{IPROT}$ is 0, the Embedded.Hypervisor category is not supported, or MSR_{GS} = 0.
  - The selected TLB entry has IPROT = 0, the Embedded.Hypervisor category is not supported, or MSR_{GS} = 0.
- If the Embedded.Hypervisor category is supported, use the first of the following sub-bullets that applies.
  - If $EPCR_{DGTMI}$=1 and MSR_{GS}=1, no TLB entry is written and a Hypervisor Privilege exception occurs.
  - If the selected entry exists, the selected entry has IPROT=1, $MAS0_{WQ}$=0b00, and MSR_{GS}=1, no TLB entry is written, and a Hypervisor Privilege exception occurs.
  - If $MAS0_{WQ}$=0b00, $MAS1_{V}$=1, $MAS1_{IPROT}$=1, and MSR_{GS}=1, no TLB entry is written, and a Hypervisor Privilege exception occurs.
  - If $MAS0_{WQ}$=0b00, $MAS1_{V}$=0, $MAS1_{IPROT}$=1, and MSR_{GS}=1, no TLB entry is written, and it is implementation-dependent whether a Hypervisor Privilege exception occurs.
  - If the selected entry has IPROT=1, $MAS0_{WQ}$=0b01, and MSR_{GS}=1, no TLB entry is written, and it is implementation-dependent whether a Hypervisor Privilege exception occurs.
  - If $MAS0_{WQ}$=0b01, $MAS1_{IPROT}$=1, and MSR_{GS}=1, no TLB entry is written, and it is implementation-dependent whether a Hypervisor Privilege exception occurs.
  - If $TLBnCFG_{HES}$=1, $MAS0_{HES}$=0, and MSR_{GS}=1, no TLB entry is written, and it is implementation-dependent whether a Hypervisor Privilege exception occurs.

If a TLB entry is to be written per the preceding description, then regardless of whether the selected TLB entry exists, $MAS0_{NV}$ provides a suggestion to hardware of what the hardware hint for replacement should be when the next Data or Instruction TLB Error Interrupt for a virtual address that uses the set of TLB entries containing the entry written by the ***tlbwe*** instruction.

If the selected TLB entry exists and the TLB entry is to be written per the preceding description, the fields of the TLB entry are loaded from the MAS registers according to the following.

- The V, TID, TS, and SIZE fields of the TLB entry are loaded from $MAS1_{V\ TID\ TS\ TSIZE}$.
- If the TLB array supports IPROT, the IPROT bit of the TLB entry is loaded from $MAS1_{IPROT}$.
- The EPN, W, I, M, G, and E fields of the TLB entry are loaded from $MAS2_{EPN\ W\ I\ M\ G\ E}$.
- If the VLE category is supported, the VLE bit of the TLB entry is loaded from $MAS2_{VLE}$.
- If Alternate Coherency Mode is supported, the ACM bit of the TLB entry is loaded from $MAS2_{ACM}$.
- The RPN field of the TLB entry is loaded from the RPN described above.
- The supported User-Defined storage control bits (U0:U3) of the TLB entry are loaded from the respective bits in $MAS3_{U0:U3}$.
- If the Embedded.Page Table category is supported and the TLB array supports indirect entries, the following applies.

- The IND of the TLB entry is loaded from $MAS1_{IND}$.
- If $MAS1_{IND}$ is 1, the SPSIZE field of the TLB entry is loaded from $MAS3_{SPSIZE}$.
- If $MAS1_{IND}$ is 0, the UX, SX, UW, SW, UR, and SR bits of the TLB entry are loaded from $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$.

- If the Embedded.Page Table category is not supported or the TLB array does not support indirect entries, the UX, SX, UW, SW, UR, and SR bits of the TLB entry are loaded from $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$.
- If the Embedded.Hypervisor category is implemented, the TGS, VF, and TLPID fields of the TLB entry are loaded from $MAS8_{TGS\ VF\ TLPID}$.

If the LRAT array is specified ($MAS0_{ATSEL}$ = 0 or MSR_{GS} = 1) for a ***tlbwe***, $MAS0_{WQ}$ must be 0b00 and $MAS0_{HES}$ must be 0. If the LRAT array is specified ($MAS0_{ATSEL}$ = 0 or MSR_{GS} = 1), $MAS0_{WQ}$ is 0b00, $MAS0_{HES}$ is 0, and the ***tlbwe*** instruction is executed in hypervisor state, the following applies.

- An RPN is formed by concatenating $MAS7_{RPNU}$ with $MAS3_{RPNL}$ (RPN = $MAS7_{RPNU}$ || $MAS3_{RPNL}$)
- The contents of the $MAS1_{V\ TSIZE}$, $MAS2_{EPN}$, and the RPN described above are written to the selected LRAT $entry_{V\ LSIZE\ LPN\ RPN}$.
- If the LPID field in the LRAT is supported ($LRATCFG_{LPID}$ = 1), $MAS8_{TLPID}$ is written to the LPID field of the selected entry.

If no exception occurs, and either $MAS0_{WQ}$ is 0b10 or a TLB array was selected by the ***tlbwe*** ($MAS0_{ATSEL}$=0 or MSR_{GS}=1), the TLB-reservation is cleared.

If $MAS0_{WQ}$ is 0b10, no TLB entry is written.

If $MAS0_{WQ}$ is 0b11, the instruction is treated as if the instruction form is invalid.

If the page size specified by $MAS1_{TSIZE}$ is not supported by the specified array, the ***tlbwe*** may be performed as if TSIZE were some implementation-dependent value, or an Illegal Instruction exception occurs.

If a TLB entry is to be written per the preceding description, $MAS1_{IND}$=1, and values of I,M,G, and E to be written to the TLB entry are inconsistent with storage that is not Caching Inhibited, Memory Coherence Required, not Guarded, and Big-Endian, the ***tlbwe*** may be performed as described or an Illegal Instruction exception occurs. Also, if a TLB entry is to be written per the preceding description, $MAS1_{IND}$=1, and values of ACM and U0:U3 to be written to the TLB entry are inconsistent with the requirements that an implementation has for storage control attributes for a Page Table, the ***tlbwe*** may be performed as described or an Illegal Instruction exception occurs.

If an invalid value is specified for $MAS0_{TLBSEL}$ $MAS0_{ESEL}$ or $MAS2_{EPN}$, either no TLB entry is written by the ***tlbwe***, or the ***tlbwe*** is performed as if some implementation-dependent, valid value were substituted for the invalid value, or an Illegal Instruction exception occurs.

A context synchronizing instruction is required after a ***tlbwe*** instruction to ensure any subsequent instructions that will use the updated TLB or LRAT values execute in the new context.

If $TLBnCFG_{HES}$=1 for the selected TLB array, a TLB write does not necessarily invalidate implementation-specific TLB lookaside information. See Section 6.11.4.4.

This instruction is hypervisor privileged if the Embedded.Hypervisor category is supported and any of the following is true.

- The Embedded.Hypervisor.LRAT category is not supported.
- MSR_{GS} = 1 and, for the TLB array selected by $MAS0_{TLBSEL}$, $TLBnCFG_{GTWE}$ = 0.
- Guest execution of *TLB Management* instructions is disabled ($EPCR_{DGTMI}$=1).

Otherwise, this instruction is privileged.

Special Registers Altered:

None

Programming Note

Care must be taken not to invalidate any TLB entry that contains the mapping for any interrupt vector.

Chapter 7. Interrupts and Exceptions

7.1 Overview

An *interrupt* is the action in which the thread saves its old context (MSR and next instruction address) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. *Exceptions* are the events that will, if enabled, cause the thread to take an interrupt.

Exceptions are generated by signals from internal and external peripherals, instructions, the internal timer Interrupts are divided into 4 classes, as described in Section 7.4.3, such that only one interrupt of each class is reported, and when it is processed no program state is lost. Since Save/Restore register pairs GSRR0/ GSRR1 <E.HV> SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: E.ED], and MCSSR0/ MCSSR1 are serially reusable resources used by guest <E.HV>, base, critical, debug [Category: E.ED], Machine Check interrupts, respectively, program state may be lost when an unordered interrupt is taken. (See Section 7.8, “Interrupt Ordering and Masking”.

7.2 Interrupt Registers

7.2.1 Save/Restore Register 0

Save/Restore Register 0 (SRR0) is a 64-bit register. SRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an ***rfi*** is executed. On a non-critical interrupt, SRR0 is set to the current or next instruction address. When ***rfi*** is executed, instruction execution continues at the address in SRR0.

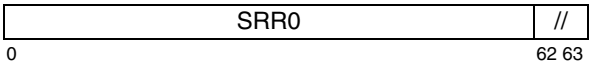


Figure 53. Save/Restore Register 0

In general, SRR0 contains the address of the instruction that caused the non-critical interrupt, or the address of the instruction to return to after a non-critical interrupt is serviced.

The contents of SRR0 when an interrupt is taken are mode dependent, reflecting the computation mode when the interrupt is taken and the computation mode entered for execution of the interrupt (specified by $EPCR_{ICM}$) <E.HV>. When computation mode when the interrupt is taken is 32-bit mode and the computation mode entered for execution of the interrupt is 64-bit mode, the high-order 32 bits of SRR0 are set to 0s. When computation mode when the interrupt is taken is 64-bit mode and the computation mode entered for execution of the interrupt is 32-bit mode, the contents SRR0 are undefined.

The contents of SRR0 upon interrupt can be described as follows (assuming Addr is the address to be put into SRR0):

```
if (MSR_CM = 0) & (EPCR_ICM = 0)
   then SRR0 ← 32undefined || Addr32:63
if (MSR_CM = 0) & (EPCR_ICM = 1)
   then SRR0 ← 320 || Addr32:63
if (MSR_CM = 1)&(EPCR_ICM = 1) then SRR0 ← Addr0:63
if (MSR_CM = 1)&(EPCR_ICM = 0) then SRR0 ← undefined
```

The contents of SRR0 can be read into register RT using ***mfspr*** *RT,SRR0*. The contents of register RS can be written into the SRR0 using ***mtspr*** *SRR0,RS*.

This register is hypervisor privileged.

7.2.2 Save/Restore Register 1

Save/Restore Register 1 (SRR1) is a 32-bit register. SRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on non-critical interrupts, and to restore machine state when an ***rfi*** is executed. When a non-critical interrupt is taken, the contents of the MSR are placed into SRR1. When ***rfi*** is executed, the contents of SRR1 are placed into the MSR.

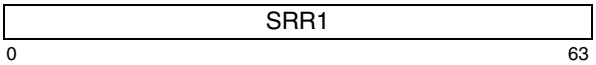


Figure 54. Save/Restore Register 1

Bits of SRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note

A MSR bit that is reserved may be inadvertently modified by ***rfi***/***rfci/rfmci***.

The contents of SRR1 can be read into register RT using ***mfspr*** *RT,SRR1*. The contents of register RS can be written into the SRR1 using ***mtspr*** *SRR1,RS*.

This register is hypervisor privileged.

7.2.3 Guest Save/Restore Register 0 [Category:Embedded.Hypervisor]

Guest Save/Restore Register 0 (GSRR0) is a 64-bit register. GSRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on guest interrupts, and to restore machine state when an ***rfgi*** is executed. On a guest interrupt, GSRR0 is set to the current or next instruction address. When ***rfgi*** is executed, instruction execution continues at the address in GSRR0.

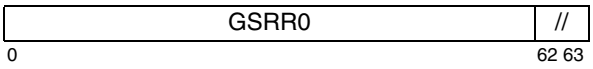


Figure 55. Guest Save/Restore Register 0

In general, GSRR0 contains the address of the instruction that caused the guest interrupt, or the address of the instruction to return to after a guest interrupt is serviced.

The contents of GSRR0 when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the interrupt (specified by $EPCR_{GICM}$). The contents of GSRR0 upon interrupt can be described as follows (assuming Addr is the address to be put into GSRR0):

```
if (MSR_CM = 0) & (EPCR_GICM = 0)
   then GSRR0 ← 32undefined || Addr_32:63
if (MSR_CM = 0) & (EPCR_GICM = 1)
   then GSRR0 ← 320 || Addr_32:63
if (MSR_CM = 1)&(EPCR_GICM = 1) then GSRR0 ← Addr_0:63
if (MSR_CM=1)&(EPCR_GICM=0) then GSRR0 ← undefined
```

The contents of GSRR0 can be read into register RT using ***mfspr*** *RT,GSRR0*. The contents of register RS can be written into the GSRR0 using ***mtspr*** *GSRR0,RS*.

This register is privileged.

Programming Note

mfspr *RT,SRR0* should be used to read GSRR0 in guest supervisor state. ***mtspr*** *SRR0,RS* should be used to write GSRR0 in guest supervisor state. See Section 2.2.1, “Register Mapping”.

7.2.4 Guest Save/Restore Register 1 [Category:Embedded.Hypervisor]

Guest Save/Restore Register 1 (GSRR1) is a 32-bit register. GSRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on guest interrupts, and to restore machine state when an ***rfgi*** is executed. When a guest interrupt is taken, the contents of the MSR are placed into GSRR1. When ***rfgi*** is executed, the contents of GSRR1 are placed into the MSR.

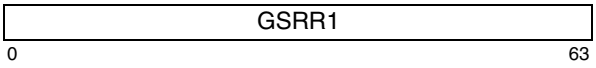


Figure 56. Guest Save/Restore Register 1

Bits of GSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note

A MSR bit that is reserved may be inadvertently modified by ***rfi/rfgi***/***rfci/rfdi/rfmci***.

The contents of GSRR1 can be read into register RT using ***mfspr*** *RT,GSRR1*. The contents of register RS can be written into the GSRR1 using ***mtspr*** *GSRR1,RS*.

This register is privileged.

Programming Note

mfspr *RT,SRR1* should be used to read GSRR1 in guest supervisor state. ***mtspr*** *SRR1,RS* should be used to write GSRR1 in guest supervisor state. See Section 2.2.1, “Register Mapping”.

7.2.5 Critical Save/Restore Register 0

Critical Save/Restore Register 0 (CSRR0) is a 64-bit register. CSRR0 bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an ***rfci*** is executed. When a critical interrupt is taken, the CSRR0 is set to the current or next instruction address. When ***rfci*** is executed,

instruction execution continues at the address in CSRR0.

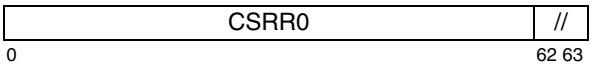


Figure 57. Critical Save/Restore Register 0

In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.

The contents of CSRR0 when an interrupt is taken are mode dependent, reflecting the computation mode when the interrupt is taken and the computation mode entered for execution of the interrupt (specified by $EPCR_{ICM}$) [Category:Embedded.Hypervisor]. If computation mode when the interrupt is taken is 32-bit mode and the computation mode entered for execution of the interrupt is 64-bit mode, the high-order 32 bits of CSRR0 are set to 0s. When computation mode when the interrupt is taken is 64-bit mode and the computation mode entered for execution of the interrupt is 32-bit mode, the contents CSRR0 are undefined.

The contents of CSRR0 upon critical interrupt can be described as follows (assuming Addr is the address to be put into CSRR0):

```
if (MSR_CM = 0) & (EPCR_ICM = 0)
   then CSRR0 ← 32undefined || Addr32:63
if (MSR_CM = 0) & (EPCR_ICM = 1)
   then CSRR0 ← 320 || Addr32:63
if (MSR_CM = 1) & (EPCR_ICM = 1) then CSRR0 ← Addr0:63
if (MSR_CM = 1)&(EPCR_ICM = 0) then CSRR0 ← undefined
```

The contents of CSRR0 can be read into register RT using ***mfspr*** *RT,CSRR0*. The contents of register RS can be written into CSRR0 using ***mtspr*** *CSRR0,RS*.

This register is hypervisor privileged.

7.2.6 Critical Save/Restore Register 1

Critical Save/Restore Register 1 (CSRR1) is a 32-bit register. CSRR1 bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The register is used to save machine state on critical interrupts, and to restore machine state when an ***rfci*** is executed. When a critical interrupt is taken, the contents of the MSR are placed into CSRR1. When ***rfci*** is executed, the contents of CSRR1 are placed into the MSR.

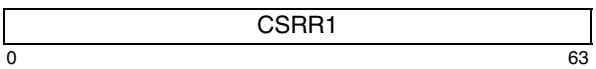


Figure 58. Critical Save/Restore Register 1

Bits of CSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note

A MSR bit that is reserved may be inadvertently modified by ***rfi***/***rfci/rfmci***.

The contents of CSRR1 can be read into bits 32:63 of register RT using ***mfspr*** *RT,CSRR1*, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the CSRR1 using ***mtspr*** *CSRR1,RS*.

This register is hypervisor privileged.

7.2.7 Debug Save/Restore Register 0 [Category: Embedded.Enhanced Debug]

Debug Save/Restore Register 0 (DSRR0) is a 64-bit register used to save machine state on Debug interrupts, and to restore machine state when an ***rfdi*** is executed. When a Debug interrupt is taken, the DSRR0 is set to the current or next instruction address. When ***rfdi*** is executed, instruction execution continues at the address in DSRR0.

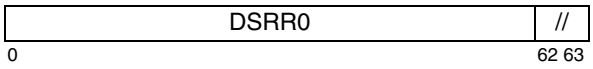


Figure 59. Debug Save/Restore Register 0

In general, DSRR0 contains the address of an instruction that was executing or just finished execution when the Debug exception occurred.

The contents of DSRR0 when an interrupt is taken are mode dependent, reflecting the computation mode when the interrupt is taken and the computation mode entered for execution of the interrupt (specified by $EPCR_{ICM}$) [Category:Embedded.Hypervisor]. If computation mode when the interrupt is taken is 32-bit mode and the computation mode entered for execution of the interrupt is 64-bit mode, the high-order 32 bits of DSRR0 are set to 0s. When computation mode when the interrupt is taken is 64-bit mode and the computation mode entered for execution of the interrupt is 32-bit mode, the contents DSRR0 are undefined.

The contents of DSRR0 upon Debug interrupt can be described as follows (assuming Addr is the address to be put into DSRR0):

```
if (MSR_CM = 0) & (EPCR_ICM = 0) then DSRR0 ← 32undefined ||
Addr32:63
if (MSR_CM = 0) & (EPCR_ICM = 1) then DSRR0 ← 320 ||Addr32:63
if (MSR_CM = 1) & (EPCR_ICM = 1) then DSRR0 ← Addr0:63
if (MSR_CM = 1) & (EPCR_ICM = 0) then DSRR0 ← undefined
```

The contents of DSRR0 can be read into register RT using ***mfspr*** RT,DSRR0. The contents of register RS can be written into DSRR0 using ***mtspr*** DSRR0,RS.

This register is hypervisor privileged.

7.2.8 Debug Save/Restore Register 1 [Category: Embedded.Enhanced Debug]

Debug Save/Restore Register 1 (DSRR1) is a 32-bit register used to save machine state on Debug interrupts, and to restore machine state when an ***rfdi*** is executed. When a Debug interrupt is taken, the contents of the Machine State Register are placed into DSRR1. When ***rfdi*** is executed, the contents of DSRR1 are placed into the Machine State Register.

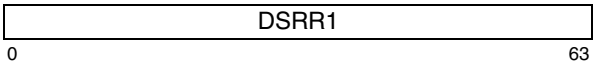


Figure 60. Debug Save/Restore Register 1

Bits of DSRR1 that correspond to reserved bits in the Machine State Register are also reserved.

The contents of DSRR1 can be read into bits 32:63 of register RT using ***mfspr*** RT,DSRR1, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the DSSR1 using ***mtspr*** DSRR1,RS.

This register is hypervisor privileged.

7.2.9 Data Exception Address Register

The Data Exception Address Register (DEAR) is a 64-bit register. DEAR bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The DEAR contains the address that was referenced by a *Load*, *Store* or *Cache Management* instruction that caused an LRAT Error interrupt <E.PT> or that caused an Alignment, Data TLB Miss, Data Storage interrupt if either the Embedded.Hypervisor category is not supported or the interrupt is directed to the hypervisor.

The contents of the DEAR when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the critical interrupt (specified by $EPCR_{ICM}$). The contents of the DEAR upon interrupt can be described as follows (assuming Addr is the address to be put into DEAR):

```
if (MSR_CM = 0) & (EPCR_ICM = 0)
   then DEAR ← 32undefined || Addr_32:63
if (MSR_CM = 0) & (EPCR_ICM = 1)
   then DEAR ← 320 || Addr_32:63
if (MSR_CM = 1) & (EPCR_ICM = 1) then DEAR ← Addr_0:63
if (MSR_CM = 1) & (EPCR_ICM = 0) then DEAR ← undefined
```

The contents of DEAR can be read into register RT using ***mfspr*** *RT,DEAR*. The contents of register RS can be written into the DEAR using ***mtspr*** *DEAR,RS*.

This register is hypervisor privileged.

7.2.10 Guest Data Exception Address Register [Category: Embedded.Hypervisor]

The Guest Data Exception Address Register (GDEAR) is a 64-bit register. GDEAR bits are numbered 0 (most-significant bit) to 63 (least-significant bit). The GDEAR contains the address that was referenced by a *Load*, *Store* or *Cache Management* instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt that was directed to the guest supervisor state. The GDEAR is identical in form and function to DEAR

The contents of the GDEAR when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the interrupt (specified by $EPCR_{GICM}$). The contents of the GDEAR upon interrupt can be described as follows (assuming Addr is the address to be put into GDEAR):

```
if (MSR_CM = 0) & (EPCR_GICM = 0)
   then GDEAR ← 32undefined || Addr_32:63
if (MSR_CM = 0) & (EPCR_GICM = 1)
   then GDEAR ← 320 || Addr_32:63
if (MSR_CM = 1) & (EPCR_GICM = 1)
   then GDEAR ← Addr_0:63
if (MSR_CM = 1)&(EPCR_GICM = 0)
   then GDEAR ← undefined
```

The contents of GDEAR can be read into register RT using ***mtspr*** *RT,GDEAR*. The contents of register RS can be written into the GDEAR using ***mtspr*** *GDEAR,RS*.

This register is privileged.

Programming Note

mfspr *RT,DEAR* should be used to read GDEAR in guest supervisor state. ***mtspr*** *DEAR,RS* should be used to write GDEAR in guest supervisor state. See Section 2.2.1, “Register Mapping”.

7.2.11 Interrupt Vector Prefix Register

The Interrupt Vector Prefix Register (IVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit).

The IVPR is used for Machine Check interrupt if the MCIVPR is not supported. The IVPR is used for other interrupts if Category E.HV is not supported or if the interrupt is directed to the hypervisor state. For these interrupts, the IVPR is used in one of the following ways.

- If Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported, the following applies. Bits 48:63 are reserved. Bits 0:47 of

the Interrupt Vector Prefix Register provide the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets from the appropriate IVOR (provided in Section 7.6.1, “Interrupt Fixed Offsets [Category: Embedded.Phased-In]”) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, the following applies. $IVPR_{52:63}$ are reserved. Bits 0:51 of the Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the exception processing routines. The 12-bit exception vector offsets (provided in Section 7.6.1, “Interrupt Fixed Offsets [Category: Embedded.Phased-In]”) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

The contents of Interrupt Vector Prefix Register can be read into register RT using ***mfspr*** *RT,IVPR*. The contents of register RS can be written into Interrupt Vector Prefix Register using ***mtspr*** *IVPR,RS*.

This register is hypervisor privileged.

7.2.12 Guest Interrupt Vector Prefix Register [Category: Embedded.Hypervisor]

The Guest Interrupt Vector Prefix Register (GIVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit).

If Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported, the following applies. $GIVPR_{48:63}$ are reserved. For interrupts directed to guest state, bits 0:47 of the Guest Interrupt Vector Prefix Register provides the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets (provided in Section 7.6.1, “Interrupt Fixed Offsets [Category: Embedded.Phased-In]”) are concatenated to the right of bits 0:47 of the Guest Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, the following applies. $GIVPR_{52:63}$ are reserved. For interrupts directed to guest state, bits 0:51 of the Guest Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the exception processing routines. The 12-bit exception vector offsets (provided in Section 7.6.1, “Interrupt Fixed Offsets [Category: Embedded.Phased-In]”) are concatenated to the right of bits 0:47 of the Guest Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

The contents of Guest Interrupt Vector Prefix Register can be read into register RT using ***mfspr*** *RT,GIVPR*. The contents of register RS can be written into Interrupt Vector Prefix Register using ***mtspr*** *GIVPR,RS*.

Write access to this register is hypervisor privileged. Read access to this register is privileged.

Programming Note

mfspr *RT,IVPR* should be used to read GIVPR in guest supervisor state. ***mtspr*** *IVPR,RS* should be used to write GIVPR in guest supervisor state. Hypervisor software should emulate the accesses for the guest.

7.2.13 Exception Syndrome Register

The Exception Syndrome Register (ESR) is a 32-bit register. ESR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The ESR provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type. Upon the generation of one of these types of interrupts, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other ESR bits are cleared. Other interrupt types do not affect the contents of the ESR. The ESR does not need to be cleared by software. Figure 61 shows the bit definitions for the ESR.

Bit(s)	Name	Meaning	Associated Interrupt Type
32:35	Implementation-dependent		(Implementation-dependent)
36	PIL	Illegal Instruction exception	Program
37	PPR	Privileged Instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operation	Alignment Data Storage Data TLB LRAT Error Program
40	ST	Store operation	Alignment Data Storage Data TLB LRAT Error
41	Reserved		
42	DLK_0	(Implementation-dependent)	(Implementation-dependent)
43	DLK_1	(implementation-dependent)	(Implementation-dependent)
44	AP	Auxiliary Processor operation	Alignment Data Storage Data TLB LRAT Error Program
45	PUO	Unimplemented Operation exception	Program
46	BO	Byte Ordering exception	Data Storage Instruction Storage
47	PIE	Imprecise exception	Program
48:52	Reserved		
53	DATA	Data Access [Category: Embedded.Page Table]	LRAT Error
54	TLBI	TLB Ineligible [Category: Embedded.Page Table]	Data Storage Instruction Storage LRAT Error
55	PT	Page Table [Category: Embedded.Page Table]	Data Storage Instruction Storage LRAT Error
56	SPV	Signal Processing operation [Category: Signal Processing Engine] Vector operation [Category: Vector]	Alignment Data Storage Data TLB LRAT Error Embedded Floating-point Data Embedded Floating-point Round SPE/Embedded Floating-point/Vector Unavailable
57	EPID	External Process ID operation [Category: Embedded.External Process ID]	Alignment Data Storage Data TLB LRAT Error

Bit(s)	Name	Meaning	Associated Interrupt Type
58	VLEMI	VLE operation [Category: VLE]	Alignment Data Storage Data TLB SPE/Embedded Floating-point/Vector Unavailable Embedded Floating-point Data Embedded Floating-point Round Instruction Storage LRAT Error Program System Call
59:61	Implementation-dependent		(Implementation-dependent)
62	MIF	Misaligned Instruction [Category: VLE]	Instruction TLB Instruction Storage

Figure 61. Exception Syndrome Register Definitions

Programming Note

The information provided by the ESR is not complete. System software may also need to identify the type of instruction that caused the interrupt, examine the TLB entry accessed by a data or instruction storage access, as well as examine the ESR to fully determine what exception or exceptions caused the interrupt. For example, a Data Storage interrupt may be caused by both a Protection Violation exception as well as a Byte Ordering exception. System software would have to look beyond ESR_{BO}, such as the state of MSR_{PR} in SRR1 and the page protection bits in the TLB entry accessed by the storage access, to determine whether or not a Protection Violation also occurred.

The contents of the ESR can be read into bits 32:63 of register RT using *mfspr RT,ESR*, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the ESR using *mtspr ESR,RS*.

This register is hypervisor privileged.

7.2.14 Guest Exception Syndrome Register [Category: Embedded.Hypervisor]

The Guest Exception Syndrome Register (GESR) is a 32-bit register. GESR bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The GESR is identical in form and function to the ESR, but is updated in place of the ESR when an interrupt is directed to the guest. For a description of bit settings and meanings see Section 7.2.13, “Exception Syndrome Register”.

The contents of the GESR can be read into bits 32:63 of register RT using ***mfspr*** *RT,GESR*, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the GESR using ***mtspr*** *GESR,RS*.

This register is privileged.

Programming Note

mfspr *RT,ESR* should be used to read GESR in guest supervisor state. ***mtspr*** *ESR,RS* should be used to write GESR in guest supervisor state. See Section 2.2.1, “Register Mapping”

7.2.15 Interrupt Vector Offset Registers [Category: Embedded.Phased-Out]

The Interrupt Vector Prefix Register (IVPR) is a 64-bit register. Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit).

The IVPR is used for Machine Check interrupt if the MCIVPR is not supported. The IVPR is used for other interrupts if Category E.HV is not supported or if the interrupt is directed to the hypervisor state. For these interrupts, the IVPR is used in one of the following ways.

- If Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported, the following applies. Bits 48:63 are reserved. Bits 0:47 of the Interrupt Vector Prefix Register provide the high-order 48 bits of the address of the exception processing routines. The 16-bit exception vector offsets from the appropriate IVOR (provided in Section 7.2.15) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.
- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, the following applies. $IVPR_{52:63}$ are reserved. Bits 0:51 of the Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the exception processing routines. The 12-bit exception vector offsets (provided in Section 7.2.15) are concatenated to the right of bits 0:47 of the Interrupt Vector Prefix Register to form the 64-bit address of the exception processing routine.

IVORi	Interrupt
IVOR0	Critical Input
IVOR1	Machine Check
IVOR2	Data Storage
IVOR3	Instruction Storage
IVOR4	External Input
IVOR5	Alignment
IVOR6	Program
IVOR7	Floating-Point Unavailable
IVOR8	System Call
IVOR9	Auxiliary Processor Unavailable
IVOR10	Decrementer
GIVOR10	Guest Decrementer [Category: Embedded.Hypervisor]
IVOR11	Fixed-Interval Timer Interrupt
GIVOR11	Guest Fixed-Interval Timer Interrupt [Category: Embedded.Hypervisor]
IVOR12	Watchdog Timer Interrupt
GIVOR12	Guest Watchdog Timer Interrupt [Category: Embedded.Hypervisor]
IVOR13	Data TLB Error
IVOR14	Instruction TLB Error
IVOR15	Debug
IVOR16 : IVOR31	Reserved
[Category: Signal Processing Engine] [Category: Vector]	
IVOR32	SPE/Embedded Floating-Point/Vector Unavailable Interrupt
[Category: SP.Embedded Float_*] (IVORs 33 & 34 are required if any SP.Float_ dependent category is supported.)	
IVOR33	Embedded Floating-Point Data Interrupt
IVOR34	Embedded Floating-Point Round Interrupt
[Category: Embedded Performance Monitor]	
IVOR35	Embedded Performance Monitor Interrupt
[Category: Embedded.Processor Control]	
IVOR36	Processor Doorbell Interrupt
IVOR37	Processor Doorbell Critical Interrupt
[Category: Embedded.Hypervisor, Embedded.Processor Control]	
IVOR38	Guest Processor Doorbell Interrupt
IVOR39	Guest Processor Doorbell Critical/ Machine Check Interrupt
[Category: Embedded.Hypervisor]	
IVOR40	Embedded Hypervisor System Call Interrupt
IVOR41	Embedded Hypervisor Privilege Interrupt
[Category: Embedded.Hypervisor.LRAT]	
IVOR 42	LRAT Error Interrupt
IVOR43.. IVOR63	Implementation-dependent

Figure 62. Interrupt Vector Offset Register Assignments

Bits 48:59 of the contents of IVORi can be read into bits 48:59 of register RT using *mfspr RT,IVORi*, setting bits 0:47 and bits 60:63 of GPR(RT) to zero. Bits 48:59 of the contents of register RS can be written into bits 48:59 of IVORi using *mtspr IVORi,RS*.

These registers are hypervisor privileged.

7.2.16 Guest Interrupt Vector Offset Register [Category: Embedded.Hypervisor.Phased-Out]

The Guest Interrupt Vector Offset Registers (GIVORs) are 32-bit registers. Guest Interrupt Vector Offset Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). Bits 32:47 and bits 60:63 are reserved. A Guest Interrupt Vector Offset Register provides the quadword index from the base address provided by the GIVPR (see Section 7.2.12) for its respective guest state interrupt. Guest Interrupt Vector Offset Registers are analogous to Interrupt Vector Offset Registers except that they are used when an interrupt is directed to the guest supervisor state. Figure 63 provides the assignments of specific Guest Interrupt Vector Offset Registers to specific interrupts.

IVORi	Interrupt
GIVOR2	Data Storage
GIVOR3	Instruction Storage
GIVOR4	External Input
GIVOR8	System Call
GIVOR13	Data TLB Error
GIVOR14	Instruction TLB Error
[Category: Embedded.Performance Monitor]]	
GIVOR 35	Embedded Performance Monitor Interrupt

Figure 63. Guest Interrupt Vector Offset Register Assignments

Bits 48:59 of the contents of GIVORi can be read into bits 48:59 of register RT using ***mfspr*** *RT,GIVORi*, setting bits 0:47 and bits 60:63 of GPR(RT) to zero. Bits 48:59 of the contents of register RS can be written into bits 48:59 of GIVORi using ***mtspr*** *GIVORi,RS*.

Write access to these registers is hypervisor privileged. Read access to these registers is privileged.

Programming Note

mfspr *RT,IVORi* should be used to read GIVORi in guest supervisor state. ***mtspr*** *IVORi,RS* should be used to write GIVOR in guest supervisor state. Hypervisor software should emulate the accesses for the guest.

Programming Note

The architecture only provides a few GIVORs that are implemented in hardware that are performance critical. Hypervisor software should emulate access to IVORs that do not have corresponding GIVORs.

7.2.17 Logical Page Exception Register [Category: Embedded.Hypervisor and Embedded.Page Table]

The Logical Page Exception Register (LPER) is a 64-bit register that is required when both the Embedded.Hypervisor and Embedded.Page Table categories are supported. LPER bits are numbered 0 (most-significant bit) to 63 (least-significant bit).

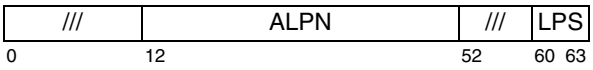


Figure 64. Logical Page Exception Register

The LPER fields are described below.

Bit	Definition
12:52	***Abbreviated Logical Page Number*** (ALPN) This field contains the Abbreviated Real Page Number from the PTE which caused the LRAT Error interrupt. Only bits corresponding to the PTE_{ARPN} bits supported by the implementation need be implemented.
60:63	***Logical Page Size*** (LPS) This field contains the Page Size from the PTE that caused the LRAT Error interrupt.

All other fields are reserved.

The LPER contains the values of the ARPN and PS fields from the PTE that was used to translate a virtual address for an instruction fetch, *Load*, *Store* or *Cache Management* instruction that caused an LRAT Error interrupt as a result of an LRAT Miss exception. The contents of LPER are unchanged by an interrupt for any other type of exception.

The LPER is a hypervisor resource.

The contents of the Logical Page Exception Register can be read into register RT using ***mfspr*** *RT,LPER*. On both a 32-bit and a 64-bit implementation, the contents of $LPER_{0:31}$ can be read into register $RT_{32:63}$ using ***mfspr*** *RT,LPERU*. The contents of register RS can be written into the LPER using ***mtspr*** *LPER,RS*. On both a 32-bit and a 64-bit implementation, the contents of register $RS_{32:63}$ can be written into the $LPER_{0:31}$ using ***mtspr*** *LPERU,RS*.

On a 32-bit implementation that supports fewer than 33 bits of real address, it is implementation-dependent whether the SPR number for LPERU is treated as a reserved value for ***mfspr*** and ***mtspr***.

The LPER is a hypervisor resource.

7.2.18 Machine Check Registers

A set of Special Purpose Registers are provided to support Machine Check interrupts.

7.2.18.1 Machine Check Save/Restore Register 0

Machine Check Save/Restore Register 0 (MCSRR0) is a 64-bit register used to save machine state on Machine Check interrupts, and to restore machine state when an ***rfmci*** is executed. When a Machine Check interrupt is taken, the MCSRR0 is set to the current or next instruction address. When ***rfmci*** is executed, instruction execution continues at the address in MCSRR0.

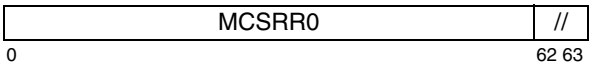


Figure 65. Machine Check Save/Restore Register 0

In general, MCSRR0 contains the address of an instruction that was executing or about to be executed when the Machine Check exception occurred.

The contents of MCSRR0 when a Machine Check interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by MSR_{CM}) and the computation mode entered for execution of the Machine Check interrupt (specified by $EPCR_{ICM}$) [Category:Embedded.Hypervisor]. The contents of MCSRR0 upon Machine Check interrupt can be described as follows (assuming Addr is the address to be put into MCSRR0):

```
if (MSR_CM = 0) & (EPCR_ICM = 0)
   then MCSRR0 ← 32undefined || Addr_32:63
if (MSR_CM = 0) & (EPCR_ICM = 1)
   then MCSRR0 ← 320 || Addr_32:63
if (MSR_CM = 1)&(EPCR_ICM = 1) then MCSRR0 ← Addr_0:63
if (MSR_CM=1)&(EPCR_ICM=0) then MCSRR0 ← undefined
```

The contents of MCSRR0 can be read into register RT using *mfspr RT,MCSRR0*. The contents of register RS can be written into MCSRR0 using *mtspr MCSRR0,RS*.

This register is hypervisor privileged.

7.2.18.2 Machine Check Save/Restore Register 1

Machine Check Save/Restore Register 1 (MCSRR1) is a 32-bit register used to save machine state on Machine Check interrupts, and to restore machine state when an ***rfmci*** is executed. When a Machine Check interrupt is taken, the contents of the MSR are placed into MCSRR1. When ***rfmci*** is executed, the contents of MCSRR1 are placed into the MSR.

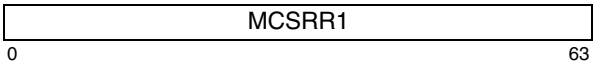


Figure 66. Machine Check Save/Restore Register 1

Bits of MCSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note

A MSR bit that is reserved may be inadvertently modified by ***rfi***/***rfci/rfmci***.

The contents of MCSRR1 can be read into register RT using *mfspr RT,MCSRR1*. The contents of register RS can be written into the MCSRR1 using *mtspr MCSRR1,RS*.

This register is hypervisor privileged.

7.2.18.3 Machine Check Syndrome Register

MCSR (MCSR) is a 64-bit register that is used to record the cause of the Machine Check interrupt. The specific definition of the contents of this register are implementation-dependent (see the User Manual of the implementation).

The contents of MCSR can be read into register RT using *mfspr RT,MCSR*. The contents of register RS can be written into the MCSR using *mtspr MCSR,RS*.

This register is hypervisor privileged.

7.2.18.4 Machine Check Interrupt Vector Prefix Register

The Machine Check Interrupt Vector Prefix Register (MCIVPR) is a 64-bit register. MCIVPR is supported only if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported. Whether the MCIVPR is supported is implementation-dependent.

Machine Check Interrupt Vector Prefix Register bits are numbered 0 (most-significant bit) to 63 (least-significant bit). $MCIVPR_{52:63}$ are reserved. Bits 0:51 of the Machine Check Interrupt Vector Prefix Register provide the high-order 52 bits of the address of the Machine Check exception processing routine. The 12-bit Machine Check exception vector offset (provided in Section 7.2.15) is concatenated to the right of bits 0:47 of the Machine Check Interrupt Vector Prefix Register to form the 64-bit address of the Machine Check exception processing routine.

The contents of Machine Check Interrupt Vector Prefix Register can be read into register RT using ***mfspr*** *RT,IVPR*. The contents of register RS can be written into Machine Check Interrupt Vector Prefix Register using ***mtspr*** *IVPR,RS*.

Programming Note

In some implementations that support Interrupt Fixed Offsets, certain instruction cache errors result in a Machine Check exception. The Machine Check interrupt handler needs to be in Caching Inhibited storage in order for the interrupt handler to operate despite an instruction cache error.

7.2.19 External Proxy Register [Category: External Proxy]

The External Proxy Register (EPR) contains implementation-dependent information related to an External Input interrupt when an External Input interrupt occurs. The EPR is only considered valid from the time that the External Input Interrupt occurs until MSR_{EE} is set to 1 as the result of a ***mtmsr*** or a return from interrupt instruction.

The format of the EPR is shown below.

Figure 67. External Proxy Register

When the External Input interrupt is taken, the contents of the EPR provide information related to the External Input Interrupt.

This register is hypervisor privileged.

Programming Note

The EPR is provided for faster interrupt processing as well as situations where an interrupt must be taken, but software must delay the resultant processing for later.

The EPR contains the vector from the interrupt controller. The process of receiving the interrupt into the EPR acknowledges the interrupt to the interrupt controller. The method for enabling or disabling the acknowledgment of the interrupt by placing the interrupt-related information in the EPR is implementation-dependent. If this acknowledgement is disabled, then the EPR is set to 0 when the External Input interrupt occurs.

7.2.20 Guest External Proxy Register [Category: Embedded Hypervisor, External Proxy]

The Guest External Proxy Register (GEPR) contains implementation-dependent information related to an External Input interrupt when an External Input interrupt directed to the guest occurs. The GEPR is only considered valid from the time that the External Input Interrupt occurs until MSR_{EE} is set to 1 as the result of a ***mtmsr*** or a return from interrupt instruction.

The format of the GEPR is shown below.

Figure 68. Guest External Proxy Register

When the External Input interrupt is taken in the guest supervisor state, the contents of the GEPR provide information related to the External Input Interrupt.

The contents of the GEPR can be read into bits 32:63 of register RT using ***mfspr*** *RT,GEPR*, setting bits 0:31 of RT to zero. The contents of bits 32:63 of register RS can be written into the GEPR using ***mtspr*** *GEPR,RS*.

The GEPR is identical in form and function to the EPR.

This register is privileged.

Programming Note

The GEPR is provided for faster interrupt processing as well as situations where an interrupt must be taken, but software must delay the resultant processing for later.

The GEPR contains the vector from the interrupt controller. The process of receiving the interrupt into the GEPR acknowledges the interrupt to the interrupt controller. The method for enabling or disabling the acknowledgment of the interrupt by placing the interrupt-related information in the GEPR is implementation-dependent. If this acknowledgement is disabled, then the GEPR is set to 0 when the External Input interrupt occurs.

Programming Note

mfspr *RT,EPR* should be used to read GEPR in guest supervisor state. Hypervisor software should emulate the accesses for the guest. This keeps the programming model consistent for an operating system running as a guest and running directly in hypervisor state.

Programming Note

Writing the GEPR register is allowed from both guest supervisor state and hypervisor state. Hypervisor must be able to write GEPR to virtualize External Input interrupt handling for the guest if the guest is using External Proxy. Writing to EPR from the guest is not mapped and results in the same behavior as any undefined supervisor level SPR.

7.3 Exceptions

There are two kinds of exceptions, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several types of interrupts to be invoked.

Examples of exceptions that can be caused directly by the execution of an instruction include but are not limited to the following:

- an attempt to execute a reserved-illegal instruction (Illegal Instruction exception type Program interrupt)
- an attempt by an application program to execute a 'privileged' instruction (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a 'privileged' Special Purpose Register (Privileged Instruction exception type Program interrupt)
- an attempt by an application program to access a Special Purpose Register that does not exist (Unimplemented Operation Instruction exception type Program interrupt)
- an attempt by a system program to access a Special Purpose Register that does not exist (boundedly undefined results)
- the execution of a defined instruction using an invalid form (Illegal Instruction exception type Program interrupt, Unimplemented Operation exception type Program interrupt, or Privileged Instruction exception type Program interrupt)
- an attempt to access a storage location that is either unavailable (Instruction TLB Error interrupt or Data TLB Error interrupt) or not permitted (Instruction Storage interrupt or Data Storage interrupt)
- an attempt to access storage with an effective address alignment not supported by the implementation (Alignment interrupt)
- the execution of a *System Call* instruction (System Call interrupt)
- the execution of a *Trap* instruction whose trap condition is met (Trap type Program interrupt)
- the execution of a floating-point instruction when floating-point instructions are unavailable (Floating-point Unavailable interrupt)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (Enabled exception type Program interrupt)
- the execution of a defined instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an instruction that is not implemented by the implementation (Illegal Instruction exception or Unimplemented Operation exception type of Program interrupt)
- the execution of an auxiliary processor instruction when the auxiliary processor instruction is unavailable (Auxiliary Processor Unavailable interrupt)
- the execution of an instruction that causes an auxiliary processor enabled exception (Enabled exception type Program interrupt)

The invocation of an interrupt is precise, except that if one of the imprecise modes for invoking the Floating-point Enabled Exception type Program interrupt is in effect then the invocation of the Floating-point Enabled Exception type Program interrupt may be imprecise. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the interrupt, has not yet occurred).

7.4 Interrupt Classification

All interrupts, except for Machine Check, can be classified as either Asynchronous or Synchronous. Independent from this classification, all interrupts, including Machine Check, can be classified into one of the following classes:

- Guest [Category:Embedded.Hypervisor]
- Base
- Critical
- Machine Check
- Debug[Category:Embedded.Enhanced Debug].

7.4.1 Asynchronous Interrupts

Asynchronous interrupts are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported to the exception handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

7.4.2 Synchronous Interrupts

Synchronous interrupts are those that are caused directly by the execution (or attempted execution) of instructions, and are further divided into two classes, *precise* and *imprecise*.

Synchronous, precise interrupts are those that *precisely* indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction.

Synchronous, imprecise interrupts are those that may indicate the address of the instruction causing the

exception that generated the interrupt, or some instruction after the instruction causing the exception.

7.4.2.1 Synchronous, Precise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point.

- GSRR0 [Category: Embedded.Hypervisor], SRR0, CSRR0, or DSRR0 [Category: Embedded.Enhanced Debug] addresses either the instruction causing the exception or the instruction immediately following the instruction causing the exception. Which instruction is addressed can be determined from the interrupt type and status bits.
- An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing thread. However, some storage accesses associated with these preceding instructions may not have been performed with respect to other threads and mechanisms.
- The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type. See Section 7.7 on page 1186.
- Architecturally, no subsequent instruction has executed beyond the instruction causing the exception.

7.4.2.2 Synchronous, Imprecise Interrupts

When the execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point.

When the execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point.

- GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSRR0 addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.
- An interrupt is generated such that all instructions preceding the instruction addressed by GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSRR0 appear to have completed with respect to the executing thread.
- If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another exception that generates an interrupt (e.g., Alignment, Data Storage), then GSRR0 [Category: Embedded.Hypervisor] or SRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see Section 7.7 on page 1186).
- If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than ***sync*** or ***isync***, then GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by an ***sync*** or ***isync*** instruction, then GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSRR0 may address either the ***sync*** or ***isync*** instruction, or the following instruction.
- If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSRR0 may have been partially executed (see Section 7.7 on page 1186).
- No instruction following the instruction addressed by GSRR0 [Category: Embedded.Hypervisor], SRR0, or CSRR0 has executed.

7.4.3 Interrupt Classes

Interrupts can also be classified as guest [Category: Embedded.Hypervisor], base, critical, Machine Check, and Debug [Category: Embedded.Enhanced Debug].

Interrupt classes other than the guest [Category: Embedded.Hypervisor] or base class may demand immediate attention even if another class of interrupt is currently being processed and software has not yet had the opportunity to save the state of the machine (i.e. return address and captured state of the MSR). For this reason, the interrupts are organized into a hierarchy (see Section 7.8). To enable taking a critical, Machine Check, or Debug [Category: Embedded.Enhanced Debug] interrupt immediately after a guest [Category: Embedded.Hypervisor] or base class interrupt occurs (i.e. before software has saved the state of the machine), these interrupts use the Save/Restore Register pair CSRR0/CSRR1, MCSRR0/MCSRR1, or DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], and guest [Category: Embedded.Hypervisor] and base class interrupts use Save/Restore Register pairs GSRR0/GSRR1 and SRR0/SRR1.respectively.

rupts use Save/Restore Register pair SRR0/SRR1.

7.4.4 Machine Check Interrupts

Machine Check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure, or by an attempt to access an invalid address. A Machine Check may be caused indirectly by the execution of an instruction, but not be recognized and/or reported until long after the thread has executed

past the instruction that caused the Machine Check. As such, Machine Check interrupts cannot properly be thought of as synchronous or asynchronous, nor as precise or imprecise. The following general rules apply to Machine Check interrupts:

1. No instruction after the one whose address is reported to the Machine Check interrupt handler in MCSRR0 has begun execution.
2. The instruction whose address is reported to the Machine Check interrupt handler in MCSRR0, and all prior instructions, may or may not have completed successfully. All those instructions that are ever going to complete appear to have done so already, and have done so within the context existing prior to the Machine Check interrupt. No further interrupt (other than possible additional Machine Check interrupts) will occur as a result of those instructions.

7.5 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, that is the address of the initial instruction that is executed when the corresponding interrupt occurs.

When Category: Embedded.Hypervisor is implemented, interrupts are directed (see Section 2.3.1, “Directed Interrupts”) to the guest supervisor state or the hypervisor state, which effects how some MSR bits are set. The conditions under which a given interrupt is directed to the guest supervisor state or hypervisor state is more fully described in the interrupt definitions for each interrupt in Section 7.6, “Interrupt Definitions”.

Interrupt processing consists of saving a small part of the thread’s state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt can be taken, the following actions are performed, in order:

1. GSRR0 [Category: Embedded.Hypervisor], SRR0, DSRR0 [Category: Embedded.Enhanced Debug], MCSRR0, or CSRR0 is loaded with an instruction address that depends on the interrupt; see the specific interrupt description for details.
2. The GESR [Category: Embedded.Hypervisor] or ESR is loaded with information specific to the exception. Note that many interrupts can only be caused by a single kind of exception event, and thus do not need nor use an ESR setting to indicate to the cause of the interrupt was.
3. GSRR1 [Category: Embedded.Hypervisor], SRR1, DSRR1 [Category: Embedded.Enhanced Debug], or MCSRR1, or CSRR1 is loaded with a copy of the contents of the MSR.
4. The MSR is updated as described below. The new values take effect beginning with the first instruction following the interrupt. MSR bits of particular interest are the following.
   - $MSR_{EE,PR,FP,FE0,FE1,IS,DS,SPV}$ are set to 0 by all interrupts.
   - If Category E.HV is supported, MSR_{GS} is left unchanged when an interrupt is directed to the guest supervisor state, otherwise they are set to 0 by all interrupts.
   - If Category E.HV is supported, MSR_{PMM} is left unchanged when an interrupt is directed to the guest supervisor state and $MSRP_{PMMP}$ = 1, otherwise MSR_{PMM} is set to 0 by all interrupts.
   - If Category E.HV is supported, MSR_{UCLE} is left unchanged when an interrupt is directed to the guest supervisor state and $MSRP_{UCLEP}$ = 1, otherwise MSR_{UCLE} is set to 0 by all interrupts.
   - MSR_{ME} is set to 0 by Machine Check interrupts and left unchanged by all other interrupts.
   - MSR_{CE} is set to 0 by critical class interrupts, Debug interrupts, and Machine Check interrupts, and is left unchanged by all other interrupts.
   - MSR_{DE} is set to 0 by critical class interrupts unless Category E.ED is supported, by Debug interrupts, and by Machine Check interrupts, and is left unchanged by all other interrupts.
   - If Category E.HV is supported and the interrupt is directed to the guest supervisor state, MSR_{CM} is set to $EPCR_{GICM}$, otherwise MSR_{CM} is set to $EPCR_{ICM}$.
   - Other supported MSR bits are left unchanged by all interrupts.

   See Section 4.2.1 for more detail on the definition of the MSR.
5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt. If Category E.HV is supported, and the interrupt is directed to the guest state, the location is one of the following, where IVORi (GIVORi) is the (Guest) Interrupt Vector Offset Register for that interrupt (see Figure 63 on page 1152):
   - $GIVPR_{0:47}$ ||$GIVORi_{48:59}$ || 0b0000 if Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported
   - $GIVPR_{0:51}$ || fixed offset shown in Figure Figure 62 on page 1152 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported

   Otherwise, the location is one of the following:
   - $IVPR_{0:47}$ || $IVORi_{48:59}$ || 0b0000 if Interrupt Vector Offset Registers [Category: Embedded.Phased-Out] are supported

- $IVPR_{0:51}$ || fixed offset shown in Figure Figure 62 on page 1152 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported and either MCIVPR is not supported or the interrupt is not a Machine Check
- $MCIVPR_{0:51}$ || fixed offset shown in Figure Figure 70 on page 1165 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, MCIVPR is supported, and the interrupt is a Machine Check

The contents of the (Guest) Interrupt Vector Prefix Register, Machine Check Interrupt Vector Prefix Register, and (Guest) Interrupt Vector Offset Registers are indeterminate upon power-on reset, and must be initialized by system software using the ***mtspr*** instruction.

It is implementation-dependent whether interrupts clear reservations obtained with ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, or ***lqarx***.

Interrupts might not clear reservations obtained with *Load and Reserve* instructions. The operating system or hypervisor should do so at appropriate points, such as at process switch or a partition switch.

At the end of an interrupt handling routine, execution of an ***rfgi*** [Category: Embedded.Hypervsior], ***rfi, rfdi*** [Category: Embedded.Enhanced Debug], ***rfmci***, or ***rfci*** causes the MSR to be restored from the contents of GSRR1 [Category: Embedded.Hypervisor], SRR1, DSRR1 [Category: Embedded.Enhanced Debug], MCSRR1, or CSRR1, and instruction execution to resume at the address contained in GSRR0 [Category: Embedded.Hypervisor], SRR0, DSRR0 [Category: Embedded.Enhanced Debug], MCSRR0, or CSRR0, respectively.

Programming Note

In general, at process switch (partition switch), due to possible process interlocks and possible data availability requirements, the operating system (hypervisor) needs to consider executing the following.

- ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.***, to clear the reservation if one is outstanding, to ensure that a ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, or ***lqarx*** in the “old” process (partition) is not paired with a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** in the “new” process (partition).
- ***sync***, to ensure that all storage operations of an interrupted process are complete with respect to other threads before that process begins executing on another thread.
- ***isync***, ***rfgi*** <E.HV>, ***rfi***, ***rfdi*** [Category: Embedded.Enhanced Debug], ***rfmci***, or ***rfci*** to ensure that the instructions in the “new” process execute in the “new” context.

Programming Note

For instruction-caused interrupts, in some cases it may be desirable for the operating system to emulate the instruction that caused the interrupt, while in other cases it may be desirable for the operating system not to emulate the instruction. The following list, while not complete, illustrates criteria by which decisions regarding emulation should be made. The list applies to general execution environments; it does not necessarily apply to special environments such as program debugging, bring-up, etc.

In general, the instruction should be emulated if:

- The interrupt is caused by a condition for which the instruction description (including related material such as the introduction to the section describing the instruction) implies that the instruction works correctly. Example: Alignment interrupt caused by ***lmw*** for which the storage operand is not aligned, or by ***dcbz*** or ***dcbzep*** for which the storage operand is in storage that is Write Through Required or Caching Inhibited.
- The instruction is an illegal instruction that should appear, to the program executing it, as if it were supported by the implementation. Example: Illegal Instruction type Program interrupt caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports, or by an instruction that is in a category that the implementation does not support but is used by some programs that the operating system supports.

In general, the instruction should not be emulated if:

- The purpose of the instruction is to cause an interrupt. Example: System Call interrupt caused by ***sc***.
- The interrupt is caused by a condition that is stated, in the instruction description, potentially to cause the interrupt. Example: Alignment interrupt caused by ***lwarx*** for which the storage operand is not aligned.
- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should be permitted to access but for which there currently is no TLB entry.)

7.6 Interrupt Definitions

Table 69 provides a summary of each interrupt type, the various exception types that may cause that interrupt type, the classification of the interrupt, which ESR (GESR) bits can be set, if any, which MSR bits can mask the interrupt type and which Interrupt Vector Offset Register is used to specify that interrupt type's vector address.

IVOR	Interrupt	Exception	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (GESR) (See Note 5)	MSR Mask Bit(s)[1]	DBCR0/TCR Mask Bit	Category (Section 1.3.5 of Book I)	Notes (see page 1164	Page
IVOR0	Critical Input	Critical Input	x			x		CE or GS		E	1	1165
IVOR1	Machine Check	Machine Check						ME or GS		E	2,4	1165
IVOR2 GIVOR2 [E.HV]	Data Storage	Access		x			[ST],[FP,AP,SPV], [PT], [VLEMI], [EPID]			E	9	1166
		Load and Reserve or *Store Conditional* to 'write-thru required' storage (W=1)		x			[ST], [VLEMI]			E	9	
		Cache Locking		x			{DLK_0,DLK_1},[ST] [VLEMI]			E	8	
		Byte Ordering		x			BO, [ST], [FP,AP,SPV], [VLEMI], [EPID]			E		
		Virtualization Fault					[ST], [FP,AP,SPV], [VLEMI], [EPID]			E.PT		
		Page Table Fault					PT, [ST], [FP,AP,SPV], [VLEMI], [EPID]			E.PT		
		TLB Ineligible					TLBI,[ST], [FP,AP,SPV], [VLEMI], [EPID]			E.PT		
IVOR3 GIVOR3 [E.HV]	Inst Storage	Access		x			[PT]			E		1168
		Byte Ordering		x			BO, [VLEMI]			E		
		Mismatched Instruction Storage (See Book VLE.))		x			BO,VLEMI			E, VLE	1	
		Misaligned Instruction Storage (See Book VLE.)		x			MIF			E, VLE	1	
		Page Table Fault					PT			E.PT		
		TLB Ineligible					TLBI			E.PT		

IVOR	Interrupt	Exception	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (GESR) (See Note 5)	MSR Mask Bit(s)[1]	DBCR0/TCR Mask Bit	Category (Section 1.3.5 of Book I)	Notes (see page 1164	Page
IVOR4	External Input	External Input	x					EE or GS		E	1	1170
GIVOR4 <E.HV>	External Input	External Input	x					EE and GS		E.HV	1	1170
IVOR5	Alignment	Alignment		x			[ST],[FP,AP,SPV] [EPID],[VLEMI]			E		1171
IVOR6	Program	Illegal		x			PIL, [VLEMI]			E		1172
		Privileged		x			PPR,[AP], [VLEMI]			E		
		Trap		x			PTR,[VLEMI]			E		
		FP Enabled		x	x		FP, [PIE]	FE0, FE1		E	6,7	
		AP Enabled		x	x		AP			E		
		Unimplemented Op		x			PUO, [VLEMI] [FP,AP,SPV]			E	7	
IVOR7	FP Unavailable	FP Unavailable		x						E		1173
IVOR8 GIVOR8 {E.HV]	System Call	System Call		x			[VLEMI]			E, E.HV		1173
IVOR9	AP Unavailable	AP Unavailable		x						E		1173
IVOR10	Decrementer		x					EE or GS	DIE	E		1174
GIVOR10 <E.HV>	Guest Decrementer		x					EE and GS	DIE	E.HV		1174
IVOR11	FIT		x					EE or GS	FIE	E		1175
GIVOR11 <E.HV>	Guest FIT		x					EE and GS	FIE	E.HV		1175
IVOR12	Watchdog		x			x		CE or GS	WIE	E	10	1176
GIVOR12 <E.HV>	Guest Watchdog		x			x		CE and GS	WIE	E.HV	10	1176
IVOR13 GIVOR13 <E.HV>	Data TLB Error	Data TLB Miss		x			[ST],[FP,AP,SPV] [VLEMI],[EPID]			E, E.HV		1177
IVOR14 GIVOR14 <E.HV>	Inst TLB Error	Inst TLB Miss		x			[MIF]			E, E.HV		1178

IVOR	Interrupt	Exception	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (GESR) (See Note 5)	MSR Mask Bit(s)[1]	DBCR0/TCR Mask Bit	Category (Section 1.3.5 of Book I)	Notes (see page 1164	Page
IVOR15	Debug	Trap		x		x		DE	IDM	E	10	1178
		Inst Addr Compare		x		x		DE	IDM	E	10	
		Data Addr Compare		x		x		DE	IDM	E	10	
		Instruction Complete		x		x		DE	IDM	E	3,10	
		Branch Taken		x		x		DE	IDM	E	3,10	
		Return From Interrupt		x		x		DE	IDM	E	10	
		Interrupt Taken	x			x		DE	IDM	E	10	
		Uncond Debug Event	x			x		DE	IDM	E.ED	10	
		Critical Interrupt Taken	x					DE	IDM	E.ED		
		Critical Interrupt Return		x				DE	IDM	E.ED		
IVOR32	SPE/Embedded Floating-Point/Vector Unavailable	SPE Unavailable		x			SPV, [VLEMI]			SPE		1179
		Vector Unavailable						SPV		V		
IVOR33	Embedded Floating-Point Data	Embedded Floating-Point Data		x			SPV, [VLEMI]			SP.F*		1180
IVOR34	Embedded Floating-Point Round	Embedded Floating-Point Round		x			SPV, [VLEMI]			SP.F*		1180
IVOR35	Embedded Performance Monitor	Embedded Performance Monitor	x					EE or GS		E.PM		1181
GIVOR35 <E.HV>	Embedded Performance Monitor	Embedded Performance Monitor	x					EE and GS		E.PM, E.HV		1181
IVOR36	Processor Doorbell	Processor Doorbell	x					EE or GS		E.PC		1181
IVOR37	Processor Doorbell Critical	Processor Doorbell Critical	x			x		CE or GS		E.PC		1183
IVOR38	Guest Processor Doorbell	Guest Processor Doorbell	x					EE and GS		E.PC, E.HV		1181
IVOR39	Guest Processor Doorbell Critical	Guest Processor Doorbell Critical	x			x		CE and GS		E.PC, E.HV		1181
	Guest Processor Doorbell Machine Check	Guest Processor Doorbell Machine Check	x			x		ME and GS		E.PC, E.HV		1183
IVOR40	Embedded Hypervisor System Call	Embedded Hypervisor System Call		x			[VLEMI]			E.HV		1183
IVOR41	Embedded Hypervisor Privilege	Embedded Hypervisor Privilege		x			[VLEMI]			E.HV		1184
IVOR42	LRAT Error	LRAT Miss		x			[ST],[FP,AP,SPV] [DATA],[PT] [VLEMI], [EPID]			E.HV. LRAT		1184

1. If an expression of MSR bits is provided, the interrupt is masked if the expression evaluates to 0 and is enabled if the expression evaluates to 1.

Figure 69. Interrupt and Exception Types

Figure 69 Notes

1. Although it is not specified, it is common for system implementations to provide, as part of the interrupt controller, independent mask and status bits for the various sources of Critical Input and External Input interrupts.
2. Machine Check interrupts are a special case and are not classified as asynchronous nor synchronous. See Section 7.4.4 on page 1157.
3. The Instruction Complete and Branch Taken debug events are only defined for MSR_{DE}=1 when in Internal Debug Mode ($DBCR0_{IDM}$=1). In other words, when in Internal Debug Mode with MSR_{DE}=0, then Instruction Complete and Branch Taken debug events cannot occur, and no DBSR status bits are set and no subsequent imprecise Debug interrupt will occur (see Section 10.4 on page 1213).
4. Machine Check status information is commonly provided as part of the system implementation, but is implementation-dependent.
5. In general, when an interrupt causes a particular ESR (GESR) bit or bits to be set (or cleared) as indicated in the table, it also causes all other ESR (GESR) bits to be cleared. There may be special rules regarding the handling of implementation-specific ESR (GESR) bits.

   Legend:

   [xxx] means $ESR(GESR)_{xxx}$ could be set

   [xxx,yyy] means either $ESR(GESR)_{xxx}$ or $ESR(GESR)_{yyy}$ may be set, but never both

   (xxx,yyy) means either $ESR(GESR)_{xxx}$ or $ESR(GESR)_{yyy}$ will be set, but never both

   {xxx,yyy} means either $ESR(GESR)_{xxx}$ or $ESR(GESR)_{yyy}$ will be set, or possibly both

   xxx means $ESR(GESR)_{xxx}$ is set
6. The precision of the Floating-point Enabled Exception type Program interrupt is controlled by the $MSR_{FE0,FE1}$ bits. When $MSR_{FE0,FE1}$=0b01 or 0b10, the interrupt may be imprecise. When such a Program interrupt is taken, if the address saved in SRR0 is not the address of the instruction that caused the exception (i.e. the instruction that caused $FPSCR_{FEX}$ to be set to 1), ESR_{PIE} is set to 1. When $MSR_{FE0,FE1}$=0b11, the interrupt is precise. When $MSR_{FE0,FE1}$=0b00, the interrupt is masked, and the interrupt will subsequently occur imprecisely if and when Floating-point Enabled Exception type Program interrupts are enabled by setting either or both of $MSR_{FE0,FE1}$, and will also cause ESR_{PIE} to be set to 1. See Section 7.6.8. Also, exception status on the exact cause is available in the Floating-Point Status and Control Register (see Section 4.2.2 and Section 4.4 of Book I).

   The precision of the Auxiliary Processor Enabled Exception type Program interrupt is implementation-dependent.
7. Auxiliary Processor exception status is commonly provided as part of the implementation.
8. Cache locking and cache locking exceptions are implementation-dependent.
9. Software must examine the instruction and the subject TLB entry to determine the exact cause of the interrupt.
10. If the Embedded.Enhanced Debug category is enabled, this interrupt is not a critical interrupt. DSRR0 and DSRR1 are used instead of CSRR0 and CSRR1.

7.6.1 Interrupt Fixed Offsets [Category: Embedded.Phased-In]

Figure 62 on page 1152 shows the 12-bit low-order effective address offset for each interrupt type. This value is the offset from the base address provided by either the IVPR (see Section 7.2.11) or the GIVPR (see Section 7.2.12).

offset	Interrupt
0x000	Machine Check
0x020	Critical Input
0x040	Debug
0x060	Data Storage
0x080	Instruction Storage
0x0A0	External Input
0x0C0	Alignment
0x0E0	Program
0x100	Floating-Point Unavailable
0x120	System Call
0x140	Auxiliary Processor Unavailable
0x160	Decrementer, Guest Decrementer [Category: Embedded.Hypervisor]
0x180	Fixed-Interval Timer Interrupt, Guest Fixed-Interval Timer Interrupt [Category: Embedded.Hypervisor]
0x1A0	Watchdog Timer Interrupt, Guest Watchdog Timer Interrupt [Category: Embedded.Hypervisor]
0x1C0	Data TLB Error
0x1E0	Instruction TLB Error
[Category: Signal Processing Engine] [Category: Vector]	
0x200	SPE/Embedded Floating-Point/Vector Unavailable Interrupt
[Category: SP.Embedded Float_*] (The following vector offsets are required if any SP.Float_ dependent category is supported.)	
0x220	Embedded Floating-Point Data Interrupt
0x240	Embedded Floating-Point Round Interrupt
[Category: Embedded Performance Monitor]	
0x260	Embedded Performance Monitor Interrupt
[Category: Embedded.Processor Control]	
0x280	Processor Doorbell Interrupt
0x2A0	Processor Doorbell Critical Interrupt
[Category: Embedded.Hypervisor]	
0x2C0	Guest Processor Doorbell
0x2E0	Guest Processor Doorbell Critical; Guest Processor Doorbell Machine Check
0x300	Embedded Hypervisor System Call
0x320	Embedded Hypervisor Privilege
[Category: Embedded.Hypervisor.LRAT]	
0x340	LRAT Error interrupt
0x360 ... 0x7FF	Reserved
0x800 ... 0xFFF	Implementation-dependent

Figure 70. Interrupt Vector Offsets

7.6.2 Critical Input Interrupt

A Critical Input interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Critical Input exception is presented to the interrupt mechanism, and MSR_{CE}=1. If Category: Embedded.Hypervisor is supported, Critical Input interrupts with the exception of Guest Processor Doorbell Critical are enabled regardless of the state of MSR_{CE} when MSR_{GS}=1. While the specific definition of a Critical Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the system. Also, implementations may provide an alternative means (in addition to MSR_{CE}) for masking the Critical Input interrupt.

CSRR0, CSRR1, and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

ME Unchanged.

DE Unchanged if category E.ED is supported; otherwise set to 0

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x020. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR0_{48:59}$|| 0b0000.

Programming Note

Software is responsible for taking any action(s) that are required by the implementation in order to clear any Critical Input exception status prior to re-enabling MSR_{CE} in order to avoid another, redundant Critical Input interrupt.

7.6.3 Machine Check Interrupt

A Machine Check interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Machine Check exception is presented to the interrupt mechanism, and MSR_{ME}=1. If Category: Embedded.Hypervisor is supported, Machine Check interrupts with the exception of Guest Processor Doorbell Machine Check are enabled regardless of the state of MSR_{ME} when MSR_{GS}=1. The specific cause or causes of Machine Check exceptions are implementation-dependent, as are the details of the actions taken on a Machine Check interrupt.

If the Machine Check Extension is implemented, MCSRR0, MCSRR1, and MCSR are set, otherwise CSRR0, CSRR1, and ESR are set. The registers are updated as follows:

CSRR0/MCSRR0

Set to an instruction address. As closely as possible, set to the effective address of an instruction that was executing or about to be executed when the Machine Check exception occurred.

CSRR1/MCSRR1

Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

All other defined MSR bits set to 0.

ESR/MCSR

Implementation-dependent.

Instruction execution resumes at address $IVPR_{0:47}$ || $IVOR1_{48:59}$||0b0000.

If the Embedded.Hypervisor category is supported, a Machine Check interrupt caused by the existence of multiple direct TLB entries or multiple indirect TLB entries (or similar entries in implementation-specific translation caches) which translate a given virtual address (see Section 6.7.3) must occur while still in the context of the partition or hypervisor state that caused it. In these cases, the interrupt must be presented in a way that permits continuing execution. Treating the exception as instruction-caused allows these requirements to be achieved. Also, if the Embedded.Hypervisor category is supported, a Machine Check interrupt resulting from the following situations must be precise.

- Execution of an *External Process ID* instruction that has an operand that can be translated by multiple TLB entries.
- Execution of a ***tlbivax*** instruction that isn't a TLB invalidate all and there are multiple entries in a single thread's TLB array(s) that match the complete VPN.
- Execution of a ***tlbilx*** instruction with T=3 and there are multiple entries in the TLB array(s) that match the complete VPN.
- Execution of a ***tlbsx*** or ***tlbsrx.*** instruction and there are multiple matching TLB entries.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported and Machine Check Interrupt Vector Prefix Register is supported, instruction execution resumes at address $MCIVPR_{0:51}$ || 0x000. If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported and Machine Check Interrupt Vector Prefix Register is not implemented, instruction execution resumes at address $IVPR_{0:51}$ || 0x000. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR1_{48:59}$|| 0b0000.

Programming Note

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, that may be placed into registers and/or on-chip caches.

Programming Note

On implementations on which a Machine Check interrupt can be caused by referring to an invalid real address, executing a ***dcbz***, ***dcbzep***, or ***dcba*** instruction can cause a delayed Machine Check interrupt by establishing in the data cache a block that is associated with an invalid real address. See Section 4.3 of Book II. A Machine Check interrupt can eventually occur if and when a subsequent attempt is made to write that block to main storage, for example as the result of executing an instruction that causes a cache miss for which the block is the target for replacement or as the result of executing a ***dcbst, dcbstep***, ***dcbf,*** or ***dcbfep*** instruction.

7.6.4 Data Storage Interrupt

A Data Storage interrupt may occur when no higher priority exception exists (see Section 7.9 on page 1190) and a Data Storage exception is presented to the interrupt mechanism. A Data Storage exception is caused when any of the following exceptions arises during execution of an instruction:

Read Access Control exception

A Read Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR_{PR}=1), a *Load* or 'load-class' *Cache Management* instruction attempts to access a location in storage that is not user mode read enabled (i.e. page access control bit UR=0).
- While in supervisor mode (MSR_{PR}=0), a *Load* or 'load-class' *Cache Management* instruction attempts to access a location in storage that is not supervisor mode read enabled (i.e. page access control bit SR=0).

Write Access Control exception

A Write Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR_{PR}=1), a *Store* or 'store-class' *Cache Management* instruction attempts to access a location in storage that is not user mode write enabled (i.e. page access control bit UW=0).
- While in supervisor mode (MSR_{PR}=0), a *Store* or 'store-class' *Cache Management* instruction attempts to access a location in storage that is not

supervisor mode write enabled (i.e. page access control bit SW=0).

Byte Ordering exception

A Byte Ordering exception may occur when the implementation cannot perform the data storage access in the byte order specified by the Endian storage attribute of the page being accessed.

Cache Locking exception

A Cache Locking exception may occur when the locked state of one or more cache lines has the potential to be altered. This exception is implementation-dependent.

Storage Synchronization exception

A Storage Synchronization exception will occur when an attempt is made to execute a *Load and Reserve* or *Store Conditional* instruction from or to a location that is Write Through Required or Caching Inhibited (if the interrupt does not occur then the instruction executes correctly: see Section 4.4.2 of Book II).

If a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

Page Table Fault exception

A Page Table Fault exception is caused when a Page Table translation occurs for a data access due to a *Load*, *Store* or *Cache Management* instruction and the Page Table Entry that is accessed is invalid (PTE Valid bit = 0).

TLB Ineligible exception

A TLB Ineligible exception is caused when a Page Table translation occurs for a data access due to a *Load*, *Store* or *Cache Management* instruction and any of the following conditions are true.

- The only TLB entries that can be used to hold the translation for the virtual address have IPROT=1
- No TLB array can be loaded from the Page Table for the page size specified by the PTE.
- The PTE_{ARPN} is treated as an LPN (The Embedded.Hypervisor category is supported) and there is no TLB array that meets all the following conditions.
  - The TLB array supports the page size specified by the PTE.
  - The TLB array can be loaded from the Page Table ($TLBnCFG_{PT}$ = 1).

If the Embedded.Hypervisor category is supported, an Data Storage interrupt resulting from a TLB Ineligible exception is always directed to hypervisor state regardless of the setting of $EPCR_{DSIGS}$.

Virtualization Fault exception [Category: Embedded.Hypervisor]

A Virtualization Fault exception will occur when a Load, Store, or Cache Management Instruction attempts to access a location in storage that has the Virtualization Fault (VF) bit set. A Data Storage interrupt resulting from a Virtualization Fault exception is always directed to hypervisor state regardless of the setting of $EPCR_{DSIGS}$

Instructions ***lswx*** or ***stswx*** with a length of zero, ***icbt***, ***dcbt***, ***dcbtep***, ***dcbtst***, ***dcbtstep***, or ***dcba*** cannot cause a Data Storage interrupt, regardless of the effective address.

Programming Note

The ***icbi***, ***icbiep***, ***icbt***, ***icbtls*** and ***icblc*** instructions are treated as *Loads* from the addressed byte with respect to address translation and protection. These *Instruction Cache Management* instructions use MSR_{DS}, not MSR_{IS}, to determine translation for their operands. Instruction Storage exceptions and Instruction TLB Miss exceptions are associated with the 'fetching' of instructions not with the 'execution' of instructions. Data Storage exceptions and Data TLB Miss exceptions are associated with the 'execution' of *Instruction Cache Management* instructions. One exception to the above is that ***icbtls*** and ***icblc*** only cause a Data Storage exception if they have neither execute access nor read access.

When a Data Storage interrupt occurs, the thread suppresses the execution of the instruction causing the Data Storage exception.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Data Storage interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM — MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME, DE — Unchanged.

All other defined MSR bits set to 0.

DEAR Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the

page whose access caused the Data Storage exception.

ESR

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.

ST Set to 1 if the instruction causing the interrupt is a *Store* or 'store-class' *Cache Management* instruction; otherwise set to 0.

$DLK_{0:1}$ Set to an implementation-dependent value due to a Cache Locking exception causing the interrupt.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

BO Set to 1 if the instruction caused a Byte Ordering exception; otherwise set to 0.

TLBI Set to 1 if a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt; otherwise set to 0.

PT If a Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, then PT is set to 1 if no TLB entry was created from the Page Table and is set to an implementation-dependent value if a TLB entry was created. See Section 6.7.4 for rules about TLB updates. If no Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, set to 0.

SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state GSRR0, GSRR1, GDEAR, and GESR are set in place of SRR0, SRR1, DEAR, and ESR, respectively. The MSR is set as follows:

MSR

CM MSR_{CM} is set to $EPCR_{GICM}$.

CE, ME,GS,DE
Unchanged.

Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

The following is a prioritized listing of the various exceptions which cause a Data Storage interrupt and the corresponding ESR bit, if applicable. Even though multiple of these exceptions may occur, at most one of the following exceptions is reported in the ESR.

1. Cache Locking <ECL>: $DLK_{0:1}$
2. Page Table Fault <E.PT>: PT
3. Virtualization Fault <E.HV>
4. TLB Ineligible <E.PT>: TLBI
5. Byte Ordering: BO
6. Read Access or Write Access: If the exception occurred during a Page Table translation, PT <E.PT>

Programming Note

Since some Data Storage exceptions are not mutually exclusive, system software may need to examine the TLB entry or the Page Table entry accessed by the data storage access in order to determine whether additional exceptions may have also occurred.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.

- $GIVPR_{0:47}$ || $GIVOR2_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x060 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.

- $IVPR_{0:47}$ || $IVOR2_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x060 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an Instruction Storage exception is presented to the interrupt mechanism. An Instruction Storage exception is caused when any of the following exceptions arises during execution of an instruction:

Execute Access Control exception

An Execute Access Control exception is caused when one of the following conditions exist.

- While in user mode (MSR_{PR}=1), an instruction fetch attempts to access a location in storage that is not user mode execute enabled (i.e. page access control bit UX=0).
- While in supervisor mode (MSR_{PR}=0), an instruction fetch attempts to access a location in storage

that is not supervisor mode execute enabled (i.e. page access control bit SX=0).

Byte Ordering exception

A Byte Ordering exception may occur when the implementation cannot perform the instruction fetch in the byte order specified by the Endian storage attribute of the page being accessed.

Page Table Fault exception

A Page Table Fault exception is caused when a Page Table translation occurs for an instruction fetch and the Page Table Entry that is accessed is invalid (Valid bit = 0).

TLB Ineligible exception

A TLB Ineligible exception is caused when a Page Table translation occurs for an instruction fetch and any of the following conditions are true.

- The only TLB entries that can be used to hold the translation for the virtual address have IPROT=1
- No TLB array can be loaded from the Page Table for the page size specified by the PTE.
- The PTE_{ARPN} is treated as an LPN (The Embedded.Hypervisor category is supported) and there is no TLB array that meets all the following conditions.
  - The TLB array supports the page size specified by the PTE.
  - The TLB array can be loaded from the Page Table ($TLBnCFG_{PT}$ = 1).

If the Embedded.Hypervisor category is supported, an Instruction Storage interrupt resulting from a TLB Ineligible exception is always directed to hypervisor state regardless of the setting of $EPCR_{ISIGS}$.

When an Instruction Storage interrupt occurs, the thread suppresses the execution of the instruction causing the Instruction Storage exception.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Instruction Storage interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

ESR

BO Set to 1 if the instruction fetch caused a Byte Ordering exception; otherwise set to 0.

TLBI Set to 1 if a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt; otherwise set to 0.

PT If a Page Table Fault or an Execute Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, then PT is set to 1 if no TLB entry was created from the Page Table and is set to an implementation-dependent value if a TLB entry was created. See Section 6.7.4 for rules about TLB updates. If no Page Table Fault or Execute Access Control exception occurred during a Page Table translation for the instruction causing the interrupt, set to 0.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state, GSRR0, GSRR1, and GESR are set in place of SRR0, SRR1, and ESR, respectively. The MSR is set as follows:

MSR

CM MSR_{CM} is set to $EPCR_{GICM}$.

CE, ME,GS,DE
Unchanged.

Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

The following is a prioritized listing of the various exceptions which cause a Instruction Storage interrupt and the corresponding ESR bit, if applicable. Even though multiple of these exceptions may occur, at most one of the following exceptions is reported in the ESR.

1. Page Table Fault <E.PT>: PT
2. TLB Ineligible <E.PT>: TLBI
3. Byte Ordering exception: BO
4. Execute Access: If the exception occurred during a Page Table translation, PT <E.PT>

Programming Note

Since some Instruction Storage exceptions are not mutually exclusive, system software may need to examine the TLB entry or the Page Table entry accessed by the data storage access in order to determine whether additional exceptions may have also occurred.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.

- $GIVPR_{0:47}$ || $GIVOR3_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x080 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.

- $IVPR_{0:47}$ || $IVOR3_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x080 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.6 External Input Interrupt

An External Input interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), an External Input exception is presented to the interrupt mechanism, and the interrupt is enabled. While the specific definition of an External Input exception is implementation-dependent, it would typically be caused by the activation of an asynchronous signal that is part of the processing system. Also, implementations may provide an alternative means (in addition to the enabled criteria) for masking the External Input interrupt.

If Category: Embedded.Hypervisor is supported, External Input interrupts are enabled if:

$(EPCR_{EXTGS} = 0)$ & $((MSR_{GS} =1)$ | $(MSR_{EE}=1))$

or

$(EPCR_{EXTGS} = 1)$ & $(MSR_{GS} =1)$ & $(MSR_{EE}=1)$

Otherwise, External Input interrupts are enabled if $MSR_{EE}=1$.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the next instruction to be executed.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EHSR_{ICM}$.

CE, ME,DE, Unchanged.

All other defined MSR bits set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to the guest supervisor state, GSRR0 and GSRR1 are set in place of SRR0 and SRR1, respectively. The MSR is set as follows:

MSR

CM MSR_{CM} is set to $EPCR_{GICM}$.

CE, ME,GS,DE Unchanged.

Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.

- $GIVPR_{0:47}$ || $GIVOR4_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x0A0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.

- $IVPR_{0:47}$ || $IVOR4_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.

  $IVPR_{0:51}$||0x0A0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Programming Note

Software is responsible for taking whatever action(s) are required by the implementation in order to clear any External Input exception status prior to re-enabling MSR_{EE} in order to avoid another, redundant External Input interrupt.

7.6.7 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an Alignment exception is presented to the interrupt mechanism. An Alignment exception may be caused when the implementation cannot perform a data storage access for one of the following reasons:

- The operand of a single-register *Load* or *Store* is not aligned.
- The instruction is a *Load Multiple* or *Store Multiple*, or a *Move Assist* for which the length of the storage operand is not zero.
- The operand of ***dcbz*** or ***dcbzep*** is in storage that is Write Through Required or Caching Inhibited, or one of these instructions is executed in an implementation that has either no data cache or a Write Through data cache or the line addressed by the instruction cannot be established in the cache because the cache is disabled or locked.
- The operand of a *Store*, except *Store Conditional*, or *Store String* for which the length of the storage operand is zero, is in storage that is Write-Through Required.

For ***lmw*** and ***stmw*** with an operand that is not word-aligned, and for *Load and Reserve* and *Store*

Conditional instructions with an operand that is not aligned, an implementation may yield boundedly undefined results instead of causing an Alignment interrupt. A *Store Conditional* to Write Through Required storage may either cause a Data Storage interrupt, cause an Alignment interrupt, or correctly execute the instruction. For all other cases listed above, an implementation may execute the instruction correctly instead of causing an Alignment interrupt. (For ***dcbz*** or ***dcbzep***, 'correct' execution means setting each byte of the block in main storage to 0x00.)

Programming Note

The architecture does not support the use of an unaligned effective address by *Load and Reserve* and *Store Conditional* instructions. If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

When an Alignment interrupt occurs, the thread suppresses the execution of the instruction causing the Alignment exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Alignment interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE Unchanged

CE, ME, DE, ICM Unchanged.

All other defined MSR bits set to 0.

DEAR Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Alignment exception.

ESR

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.

ST Set to 1 if the instruction causing the interrupt is a *Store*; otherwise set to 0.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.

SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x0C0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR5_{48:59}$|| 0b0000.

7.6.8 Program Interrupt

A Program interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Program exception is presented to the interrupt mechanism, and, for Floating-point Enabled exception, $MSR_{FE0,FE1}$ are non-zero. A Program exception is caused when any of the following exceptions arises during execution of an instruction:

Floating-point Enabled exception

A Floating-point Enabled exception is caused when $FPSCR_{FEX}$ is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a *Move To FPSCR* instruction that causes an exception bit and the corresponding enable bit both to be 1. Note that in this context, the term 'enabled exception' refers to the enabling provided by control bits in the Floating-Point Status and Control Register. See Section 4.2.2 of Book I.

Auxiliary Processor Enabled exception

The cause of an Auxiliary Processor Enabled exception is implementation-dependent.

Illegal Instruction exception

An Illegal Instruction exception *does* occur when execution is attempted of any of the following kinds of instructions.

- a reserved-illegal instruction
- when MSR_{PR}=1 (user mode), an ***mtspr*** or ***mfspr*** that specifies an spr value with spr_5=0 (user-mode accessible) that represents an unimplemented Special Purpose Register

An Illegal Instruction exception *may* occur when execution is attempted of any of the following kinds of instructions. If the exception does not occur, the alternative is shown in parentheses.

- an instruction that is in invalid form (boundedly undefined results)

- an ***lswx*** instruction for which register RA or register RB is in the range of registers to be loaded (boundedly undefined results)
- a defined instruction that is not implemented by the implementation (Unimplemented Operation exception)

Privileged Instruction exception

A Privileged Instruction exception occurs when MSR_{PR}=1 and execution is attempted of any of the following kinds of instructions.

- a privileged instruction
- an ***mtspr*** or ***mfspr*** instruction that specifies an spr value with spr_5=1

Trap exception

A Trap exception occurs when any of the conditions specified in a *Trap* instruction are met and the exception is not also enabled as a Debug interrupt. If enabled as a Debug interrupt (i.e. $DBCR0_{TRAP}$=1, $DBCR0_{IDM}$=1, and MSR_{DE}=1), then a Debug interrupt will be taken instead of the Program interrupt.

Unimplemented Operation exception

An Unimplemented Operation exception *may* occur when execution is attempted of a defined instruction that is not implemented by the implementation. Otherwise an Illegal Instruction exception occurs.

An Unimplemented Operation exception *may* also occur when the thread is in 32-bit mode and execution is attempted of an instruction that is part of the 64-Bit category. Otherwise the instruction executes normally.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 For all Program interrupts except an Enabled exception when in one of the imprecise modes (see Section 4.2.1 on page 1035) or when a disabled exception is subsequently enabled, set to the effective address of the instruction that caused the Program interrupt.

For an imprecise Enabled exception, set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If it points to a subsequent instruction, that instruction has not been executed, and ESR_{PIE} is set to 1. If a subsequent instruction is an ***sync*** or ***isync***, SRR0 will point at the ***sync*** or ***isync*** instruction, or at the following instruction.

If $FPSCR_{FEX}$=1 but both MSR_{FE0}=0 and MSR_{FE1}=0, an Enabled exception type Program interrupt will occur imprecisely prior to or at the next synchronizing event if these MSR bits are altered by any instruction that can set the MSR so that the expression

(MSR_{FE0} | MSR_{FE1}) & $FPSCR_{FEX}$

is 1. When this occurs, SRR0 is loaded with the address of the instruction that would have executed next, not with the address of the instruction that modified the MSR causing the interrupt, and ESR_{PIE} is set to 1.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE
Unchanged

All other defined MSR bits set to 0.

ESR

PIL Set to 1 if an Illegal Instruction exception type Program interrupt; otherwise set to 0

PPR Set to 1 if a Privileged Instruction exception type Program interrupt; otherwise set to 0

PTR Set to 1 if a Trap exception type Program interrupt; otherwise set to 0

PUO Set to 1 if an Unimplemented Operation exception type Program interrupt; otherwise set to 0

FP Set to 1 if the instruction causing the interrupt is a floating-point instruction; otherwise set to 0.

PIE Set to 1 if a Floating-point Enabled exception type Program interrupt, and the address saved in SRR0 is not the address of the instruction causing the exception (i.e. the instruction that caused $FPSCR_{FEX}$ to be set); otherwise set to 0.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor instruction; otherwise set to 0.

SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x0E0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR6_{48:59}$|| 0b0000.

7.6.9 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), an attempt is made to execute a floating-point instruction (i.e. any instruction listed in Section 4.6 of Book I), and MSR_{FP}=0.

When a Floating-Point Unavailable interrupt occurs, the hardware suppresses the execution of the instruction causing the Floating-Point Unavailable interrupt.

SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EPCR_{ICM}$.
CE, ME,DE
Unchanged

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x100. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR7_{48:59}$|| 0b0000.

7.6.10 System Call Interrupt

A System Call interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and a *System Call* (***sc***) instruction is executed.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction *after* the ***sc*** instruction.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EPCR_{ICM}$.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state (MSR_{GS} = 1), GSRR0 and GSRR1 are set in place of SRR0 and SRR1, respectively. The MSR is set as follows:

MSR
CM MSR_{CM} is set to $EPCR_{GICM}$.
CE, ME,GS,DE
Unchanged.

Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.

- $GIVPR_{0:47}$ || $GIVOR8_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x120 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.

- $IVPR_{0:47}$ || $IVOR8_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x120 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.11 Auxiliary Processor Unavailable Interrupt

An Auxiliary Processor Unavailable interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), an attempt is made to execute an Auxiliary Processor instruction (including Auxiliary Processor loads, stores, and moves), the target Auxiliary Processor is present on the implementation, and the Auxiliary Processor is configured as unavailable. Details of the Auxiliary Processor, its instruction set, and its configuration are implementation-dependent. See User's Manual for the implementation.

When an Auxiliary Processor Unavailable interrupt occurs, the hardware suppresses the execution of the instruction causing the Auxiliary Processor Unavailable interrupt.

Registers SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EPCR_{ICM}$.
CE, ME,DE
Unchanged

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x140. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR9_{48:59}$|| 0b0000.

7.6.12 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Decrementer exception exists (TSR_{DIS}=1), and the excep-

tion is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR[DIE]=1 and (MSR_{EE}=1 or MSR_{GS}=1). Otherwise, the interrupt is enabled by TCR_{DIE}=1 and MSR_{EE}=1. See Section 9.3 on page 1199.

Programming Note

MSR_{EE} also enables the External Input and Fixed-Interval Timer interrupts.

SRR0, SRR1, MSR, and TSR are updated as follows:

SRR0 Set to the effective address of the next instruction to be executed.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE Unchanged

All other defined MSR bits set to 0.

TSR (See Section 9.7.1 on page 1204.)

DIS Set to 1.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x160. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR10_{48:59}$|| 0b0000.

Programming Note

Software is responsible for clearing the Decrementer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Decrementer interrupt. To clear the Decrementer exception, the interrupt handling routine must clear TSR_{DIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

7.6.13 Guest Decrementer Interrupt

A Guest Decrementer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Decrementer exception exists ($GTSR_{DIS}$=1), and the exception is enabled. The interrupt is enabled by GTCR[DIE]=1 and MSR_{EE}=1 and MSR_{GS}=1. See Section 9.7 on page 1203.

Programming Note

MSR_{EE} also enables the External Input and Guest Fixed-Interval Timer interrupts.

GSRR0, GSRR1, MSR, and GTSR are updated as follows:

GSRR0 Set to the effective address of the next instruction to be executed.

GSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{GICM}$.

CE, ME, DE, GS Unchanged

All other defined MSR bits set to 0.

Guest TSR (See Section 9.8.1 on page 1207.)

DIS Set to 1.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $GIVPR_{0:51}$ || 0x160. Otherwise, instruction execution resumes at address $GIVPR_{0:47}$ || $GIVOR10_{48:59}$|| 0b0000.

Programming Note

Software is responsible for clearing the Guest Decrementer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Guest Decrementer interrupt. To clear the Guest Decrementer exception, the interrupt handling routine must clear TSR_{DIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Hypervisor software can modify the value of the GTSR by writing the desired value to the GTSRWR. Bits specified in the GTSRWR directly set or clear the corresponding implemented bits in the GTSR.

7.6.14 Fixed-Interval Timer Interrupt

A Fixed-Interval Timer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Fixed-Interval Timer exception exists (TSR_{FIS}=1), and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR_{FIE}=1 and (MSR_{EE}=1 or MSR_{GS} = 1). Otherwise, the interrupt is enabled by TCR_{FIE}=1 and MSR_{EE}=1. See Section 9.9 on page 1208.

Programming Note

MSR_{EE} also enables the External Input and Decrementer interrupts.

SRR0, SRR1, MSR, and TSR are updated as follows:

SRR0 Set to the effective address of the next instruction to be executed.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EPCR_{ICM}$.
CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

TSR (See Section 9.7.1 on page 1204.)
FIS Set to 1

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x180. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR11_{48:59}$|| 0b0000.

Programming Note

Software is responsible for clearing the Fixed-Interval Timer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Fixed-Interval Timer interrupt. To clear the Fixed-Interval Timer exception, the interrupt handling routine must clear TSR_{FIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

7.6.15 Guest Fixed Interval Timer Interrupt

A Guest Decrementer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Decrementer exception exists ($GTSR_{DIS}$=1), and the exception is enabled. The interrupt is enabled by GTCR[DIE]=1 and MSR_{EE}=1 and MSR_{GS}=1. See Section 9.8 on page 1206.

Programming Note

MSR_{EE} also enables the External Input and Guest Fixed-Interval Timer interrupts.

GSRR0, GSRR1, MSR, and GTSR are updated as follows:

GSRR0 Set to the effective address of the next instruction to be executed.

GSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EPCR_{GICM}$.
CE, ME, DE, GS
Unchanged

All other defined MSR bits set to 0.

Guest TSR (See Section 9.8.1 on page 1207.)
DIS Set to 1.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $GIVPR_{0:51}$ || 0x160. Otherwise, instruction execution resumes at address $GIVPR_{0:47}$ || $GIVOR10_{48:59}$|| 0b0000.

Programming Note

Software is responsible for clearing the Guest Decrementer exception status prior to re-enabling the MSR_{EE} bit in order to avoid another redundant Guest Decrementer interrupt. To clear the Guest Decrementer exception, the interrupt handling routine must clear TSR_{DIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Hypervisor software can modify the value of the GTSR by writing the desired value to the GTSRWR. Bits specified in the GTSRWR directly set or clear the corresponding implemented bits in the GTSR.

7.6.16 Watchdog Timer Interrupt

A Watchdog Timer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Watchdog Timer exception exists (TSR_{WIS}=1), and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR_{WIE}=1 and (MSR_{CE} = 1 or MSR_{GS}=1).Otherwise, the interrupt is enabled by TCR_{WIE}=1 and MSR_{CE}=1. See Section 9.11 on page 1208.

Programming Note

MSR_{CE} also enables the Critical Input interrupt.

CSRR0, CSRR1, MSR, and TSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
ME Unchanged.
DE Unchanged if category E.ED is supported; otherwise set to 0.

All other defined MSR bits set to 0.

TSR (See Section 9.7.1 on page 1204.)
WIS Set to 1.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x1A0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR12_{48:59}$|| 0b0000.

Programming Note

Software is responsible for clearing the Watchdog Timer exception status prior to re-enabling the MSR_{CE} bit in order to avoid another redundant Watchdog Timer interrupt. To clear the Watchdog Timer exception, the interrupt handling routine must clear TSR_{WIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

7.6.17 Guest Watchdog Timer Interrupt

A Guest Watchdog Timer interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Guest Watchdog Timer exception exists ($GTSR_{WIS}$=1), and the exception is enabled. The interrupt is enabled by $GTCR_{WIE}$=1 and MSR_{CE} = 1 and MSR_{GS}=1. See Section 9.8 on page 1206.

Programming Note

MSR_{CE} also enables the Critical Input interrupt.

CSRR0, CSRR1, MSR, and GTSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EPCR_{ICM}$.
ME Unchanged.
DE Unchanged if category E.ED is supported; otherwise set to 0.

All other defined MSR bits set to 0.

Guest TSR (See Section 9.8.1 on page 1207.)
WIS Set to 1.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x1A0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR12_{48:59}$|| 0b0000.

Programming Note

Software is responsible for clearing the Guest Watchdog Timer exception status prior to re-enabling the MSR_{CE} bit in order to avoid another redundant Guest Watchdog Timer interrupt. To clear the Guest Watchdog Timer exception, the interrupt handling routine must clear TSR_{WIS}. Clearing is done by writing a word to TSR using ***mtspr*** with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Hypervisor software can modify the value of the GTSR by writing the desired value to the GTSRWR. Bits specified in the GTSRWR directly set or clear the corresponding implemented bits in the GTSR.

Programming Note

As all Watchdog Timer interrupts are directed to the hypervisor, it is the responsibility of hypervisor software to reflect interrupts generated by the Guest Watchdog Timer to the guest supervisor.

7.6.18 Data TLB Error Interrupt

A Data TLB Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and any of the following Data TLB Error exceptions is presented to the interrupt mechanism.

TLB Miss exception

Caused when the virtual address associated with a data storage access does not match any valid entry in the TLB as specified in Section 6.7.2 on page 1081.

If a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** would not perform its store in the absence of a Data Storage interrupt, and a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation dependent whether a Data Storage interrupt occurs.

When a Data TLB Error interrupt occurs, the hardware suppresses the execution of the instruction causing the Data TLB Error interrupt.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, MSR, DEAR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Data TLB Error interrupt

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EHSR_{ICM}$.
CE, ME, DE
Unchanged.

All other defined MSR bits set to 0.

DEAR Set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the Data TLB Error exception.

ESR
ST Set to 1 if the instruction causing the interrupt is a *Store*, ***dcbi***, ***dcbz***, or ***dcbzep*** instruction; otherwise set to 0.
FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise set to 0.
AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store; otherwise set to 0.
SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation; otherwise set to 0.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction; otherwise set to 0.

All other defined ESR bits are set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state, GSRR0, GSRR1, GDEAR, and GESR are set in place of SRR0, SRR1, DEAR, and ESR, respectively. The MSR is set as follows:

MSR
CM MSR_{CM} is set to $EPCR_{GICM}$.
CE, ME,GS,DE
Unchanged.

Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.
- $GIVPR_{0:47}$ || $GIVOR13_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x1C0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.
- $IVPR_{0:47}$ || $IVOR13_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x1C0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.19 Instruction TLB Error Interrupt

An Instruction TLB Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and any of the following Instruction TLB Error exceptions is presented to the interrupt mechanism.

TLB Miss exception

Caused when the virtual address associated with an instruction fetch does not match any valid entry in the TLB as specified in Section 6.7.2 on page 1081.

When an Instruction TLB Error interrupt occurs, the hardware suppresses the execution of the instruction causing the Instruction TLB Miss exception.

If Category: Embedded.Hypervisor is not supported or if Category: Embedded.Hypervisor is supported and the interrupt is directed to hypervisor state, SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Instruction TLB Error interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR
CM MSR_{CM} is set to $EPCR_{ICM}$.
CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

If Category: Embedded.Hypervisor is supported and the interrupt is directed to guest supervisor state, GSRR0 and GSRR1 are set in place of SRR0 and SRR1, respectively. The MSR is set as follows:

MSR
CM MSR_{CM} is set to $EPCR_{GICM}$.
CE, ME,GS,DE
Unchanged.

Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

All other defined MSR bits set to 0.

If Category Embedded.Hypervisor is supported and the interrupt is directed to the guest state, instruction execution resumes at the address given by one of the following.

- $GIVPR_{0:47}$ || $GIVOR14_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x1E0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

Otherwise, instruction execution resumes at the address given by one of the following.

- $IVPR_{0:47}$ || $IVOR14_{48:59}$||0b0000 if IVORs [Category: Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x1E0 if Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported.

7.6.20 Debug Interrupt

A Debug interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), a Debug exception exists in the DBSR, and Debug interrupts are enabled ($DBCR0_{IDM}$=1 and MSR_{DE}=1). A Debug exception occurs when a Debug Event causes a corresponding bit in the DBSR to be set. See Section 10.5.

If the Embedded.Enhanced Debug category is not supported or is supported and is not enabled, CSRR0, CSRR1, MSR, and DBSR are updated as follows. If the Embedded.Enhanced Debug category is supported and is enabled, DSRR0 and DSRR1 are updated as specified below and CSRR0 and CSRR1 are not changed. The means by which the Embedded.Enhanced Debug category is enabled is implementation-dependent.

CSRR0 or **DSRR0** [Category: Embedded.Enhanced Debug]

For Debug exceptions that occur while Debug interrupts are enabled ($DBCR0_{IDM}$=1 and MSR_{DE}=1), CSRR0 is set as follows:

- For Instruction Address Compare (IAC1, IAC2, IAC3, IAC4), Data Address Compare (DAC1R, DAC1W, DAC2R, DAC2W), Trap (TRAP), or Branch Taken (BRT) debug exceptions, set to the address of the instruction causing the Debug interrupt.
- For Instruction Complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the Debug interrupt.
- For Unconditional Debug Event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the Debug interrupt had not occurred.
- For Interrupt Taken (IRPT) debug exceptions, set to the interrupt vector value of the interrupt that caused the Interrupt Taken debug event.
- For Return From Interrupt (RET) debug exceptions, set to the address of the ***rfi*** instruction that caused the Debug interrupt.
- For Critical Interrupt Taken (CRPT) debug exceptions, DSRR0 is set to the address of the first instruction of the critical interrupt handler. CSRR0 is unaffected.
- For Critical Interrupt Return (CRET) debug exceptions, DSRR0 is set to the address of the ***rfci*** instruction that caused the Debug interrupt. See

Section 10.4.10, “Critical Interrupt Return Debug Event [Category: Embedded.Enhanced Debug]”.

For Debug exceptions that occur while Debug interrupts are disabled ($DBCR0_{IDM}$=0 or MSR_{DE}=0), a Debug interrupt will occur at the next synchronizing event if $DBCR0_{IDM}$ and MSR_{DE} are modified such that they are both 1 and if the Debug exception Status is still set in the DBSR. When this occurs, CSRR0 or DSRR0 [Category:Embedded.Enhanced Debug] is set to the address of the instruction that would have executed next, not with the address of the instruction that modified the Debug Control Register 0 or MSR and thus caused the interrupt.

CSRR1 or **DSRR1** [Category: Embedded.Enhanced Debug]

Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
ME Unchanged

All other supported MSR bits set to 0.

DBSR Set to indicate type of Debug Event (see Section 10.5.2)

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x040. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR15_{48:59}$|| 0b0000.

7.6.21 SPE/Embedded Floating-Point/Vector Unavailable Interrupt [Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Vector, Vector]

The SPE/Embedded Floating-Point/Vector Unavailable interrupt occurs when no higher priority exception exists, and an attempt is made to execute an SPE, SPE.Embedded Float Scalar Double, SPE.Embedded Float Vector, or Vector instruction and MSR_{SPV} = 0.

When an Embedded Floating-Point Unavailable interrupt occurs, the hardware suppresses the execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Embedded Floating-Point Unavailable interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
CE, ME,DE
Unchanged

ESR

SPV Set to 1.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x200. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR32_{48:59}$|| 0b0000.

Programming Note

This interrupt is also used by the Signal Processing Engine in the same manner. It should be used by software to determine if the application is using the upper 32 bits of the GPRs in a 32-bit implementation and thus be required to save and restore them on context switch.

7.6.22 Embedded Floating-Point Data Interrupt [Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, SPE.Embedded Float Vector]

The Embedded Floating-Point Data interrupt occurs when no higher priority exception exists (see Section 7.9) and an Embedded Floating-Point Data exception is presented to the interrupt mechanism. The Embedded Floating-Point Data exception causing the interrupt is indicated in the SPEFSCR; these exceptions include Embedded Floating-Point Invalid Operation/Input Error (FINV, FINVH), Embedded Floating-Point Divide By Zero (FDBZ, FDBZH), Embedded Floating-Point Overflow (FOV, FOVH), and Embedded Floating-Point Underflow (FUNF, FUNFH)

When an Embedded Floating-Point Data interrupt occurs, the hardware suppresses the execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Embedded Floating-Point Data interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE
Unchanged

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined MSR bits set to 0.

ESR

SPV Set to 1.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x220. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR33_{48:59}$|| 0b0000.

7.6.23 Embedded Floating-Point Round Interrupt [Categories: SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, SPE.Embedded Float Vector]

The Embedded Floating-Point Round interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190), $SPEFSCR_{FINXE}$ is set to 1, and any of the following occurs:

- the unrounded result of an *Embedded Floating-Point* operation is not exact
- an overflow occurs and overflow exceptions are disabled (FOVF or FOVFH is set to 1 and FOVFE is set to 0)
- an underflow occurs and underflow exceptions are disabled (FUNF is set to 1 and FUNFE is set to 0).

The value of $SPEFSCR_{FINXS}$ is 1, indicating that one of the above exceptions has occurred, and additional information about the exception is found in $SPEFSCR_{FGH\ FG\ FXH\ FX}$.

When an Embedded Floating-Point Round interrupt occurs, the hardware completes the execution of the instruction causing the exception and writes the result to the destination register prior to taking the interrupt.

SRR0, SRR1, MSR, and ESR are updated as follows:

SRR0 Set to the effective address of the instruction following the instruction causing the Embedded Floating-Point Round interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE
Unchanged

All other defined MSR bits set to 0.

ESR

SPV Set to 1.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

All other defined ESR bits are set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x240. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR34_{48:59}$|| 0b0000.

Programming Note

If an implementation does not support ±Infinity rounding modes and the rounding mode is set to be +Infinity or -Infinity, an Embedded Floating-Point Round interrupt occurs after every *Embedded Floating-Point* instruction for which rounding might occur regardless of the value of FINXE, provided no higher priority exception exists.

When an Embedded Floating-Point Round interrupt occurs, the unrounded (truncated) result of an inexact high or low element is placed in the target register. If only a single element is inexact, the other exact element is updated with the correctly rounded result, and the FG and FX bits corresponding to the other exact element will both be 0.

The bits FG (FGH) and FX (FXH) are provided so that an interrupt handler can round the result as it desires. FG (FGH) is the value of the bit immediately to the right of the least significant bit of the destination format mantissa from the infinitely precise intermediate calculation before rounding. FX (FXH) is the value of the 'or' of all the bits to the right of the FG (FGH) of the destination format mantissa from the infinitely precise intermediate calculation before rounding.

7.6.24 Performance Monitor Interrupt [Category: Embedded.Performance Monitor]

The Performance Monitor interrupt is part of the optional Performance Monitor facility; see Appendix D.

7.6.25 Processor Doorbell Interrupt [Category: Embedded.Processor Control]

A Processor Doorbell Interrupt occurs when no higher priority exception exists, a Processor Doorbell exception is present, and the interrupt is enabled (MSR_{EE}=1). Processor Doorbell exceptions are generated when DBELL messages (see Section 11) are received and accepted by the thread.

If Category: Embedded.Hypervisor is supported, the interrupt is enabled if MSR_{GS} = 1 or MSR_{EE}=1.

SRR0, SRR1 and MSR are updated as follows:

SRR0 Set to the effective address of the next instruction to be executed.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x280. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR36_{48:59}$|| 0b0000.

7.6.26 Processor Doorbell Critical Interrupt [Category: Embedded.Processor Control]

A Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Processor Doorbell Critical exception is present, and the interrupt is enabled (MSR_{CE}=1). Processor Doorbell Critical exceptions are generated when DBELL_CRIT messages (see Section 11) are received and accepted by the thread.

If Category: Embedded.Hypervisor is supported, the interrupt is enabled if MSR_{GS} = 1 or MSR_{CE}=1.

CSRR0, CSRR1 and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
ME Unchanged.
DE Unchanged if category E.ED is supported, otherwise set to 0

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x2A0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR37_{48:59}$|| 0b0000.

7.6.27 Guest Processor Doorbell Interrupt [Category: Embedded.Hypervisor,Embedded.Processor Control]

A Guest Processor Doorbell Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell exception is present, and the interrupt is enabled (MSR_{GS}=1 and MSR_{EE}=1). Guest Processor Doorbell exceptions are generated when G_DBELL messages (see Section 11) are received and accepted by the thread.

GSRR0, GSRR1 and MSR are updated as follows:

GSRR0 Set to the effective address of the next instruction to be executed.

GSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x2C0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR38_{48:59}$|| 0b0000.

Programming Note

Guest Processor Doorbell interrupts are used by the hypervisor to be notified when the guest operating system has set MSR_{EE} to 1. This allows the hypervisor to reflect base class interrupts to the guest at a time when the guest is ready to accept them (MSR_{GS}=1 and MSR_{EE}=1).

Programming Note

Some guest operating systems running on a hypervisor may use *lazy* interrupt blocking. That is, when the operating system wants to block interrupts at the interrupt controller, it does not actually perform the blocking operation, but instead sets a value in memory that represents the level at which interrupts are to be blocked. When an actual interrupt occurs, this value is consulted to determine if the interrupt should have been blocked. If so, the current interrupt level that is to be blocked is set in the interrupt controller and the interrupt handling code returns without acknowledging the interrupt. When interrupts are unblocked at a later time, the interrupt will be reasserted by the interrupt controller. When a hypervisor is taking external interrupts and then reflecting them to a guest, the hypervisor must acknowledge the interrupt before reflecting it to the guest since the external interrupt will occur again once MSR_{GS} = 1 regardless of the state of MSR_{EE}.

To emulate the behavior required for lazy interrupt blocking by the guest, the hypervisor should execute another ***msgsnd*** instruction specifying a Guest Processor Doorbell at the time that it is reflecting the interrupt to the guest. When the guest performs its interrupt acknowledge (a hypercall or writing to an interrupt controller register emulated by the hypervisor), the hypervisor can execute a ***msgclr*** to clear a pending message if there are no other interrupts to be reflected to the guest.

7.6.28 Guest Processor Doorbell Critical Interrupt [Category: Embedded.Hypervisor,Embedded.Processor Control]

A Guest Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Critical exception is present, and the interrupt is enabled (MSR_{GS} = 1 and MSR_{CE}=1). Guest Processor Doorbell Critical exceptions are generated when G_DBELL_CRIT messages (see Section 11) are received and accepted by the thread.

CSRR0, CSRR1 and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
ME Unchanged.
DE Unchanged if category E.ED is supported, otherwise set to 0

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x2E0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR39_{48:59}$|| 0b0000.

Programming Note

Guest Processor Doorbell Critical interrupts are used by the hypervisor to be notified when the guest operating system has set MSR_{CE} to 1. This allows the hypervisor to reflect critical class interrupts to the guest at a time when the guest is ready to accept them (MSR_{GS}=1 and MSR_{CE}=1).

7.6.29 Guest Processor Doorbell Machine Check Interrupt [Category: Embedded.Hypervisor,Embedded.Processor Control]

A Guest Processor Doorbell Machine Check Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Machine Check exception is present, and the interrupt is enabled (MSR_{GS} = 1 and MSR_{ME}=1). Guest Processor Doorbell Machine Check exceptions are generated when G_DBELL_MC messages (see Section 11) are received and accepted by the thread.

CSRR0, CSRR1 and MSR are updated as follows:

CSRR0 Set to the effective address of the next instruction to be executed.

CSRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
ME Unchanged.
DE Unchanged if category E.ED is supported, otherwise set to 0

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x2E0. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR39_{48:59}$|| 0b0000.

Programming Note

Guest Processor Doorbell Machine Check interrupts are used by the hypervisor to be notified when the guest operating system has set MSR_{ME} to 1. This allows the hypervisor to reflect machine check class interrupts to the guest at a time when the guest is ready to accept them (MSR_{GS}=1 and MSR_{ME}=1).

Programming Note

Guest Processor Doorbell Critical interrupts and Guest Processor Doorbell Machine Check interrupts share the same IVOR. Hypervisor software can differentiate between the two interrupts by comparing whether CE or ME is set in CSRR1 and which interrupt class is to be reflected.

7.6.30 Embedded Hypervisor System Call Interrupt [Category: Embedded.Hypervisor]

An Embedded Hypervisor System Call interrupt occurs when no higher priority exception exists (see Section 7.9) and a *System Call* (***sc***) instruction with LEV = 1 is executed.

SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction *after* the ***sc*** instruction.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.
CE, ME,DE

Unchanged.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x300. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR40_{48:59}$|| 0b0000.

7.6.31 Embedded Hypervisor Privilege Interrupt [Category: Embedded.Hypervisor]

An Embedded Hypervisor Privilege interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an Embedded Hypervisor Privilege exception is presented to the exception mechanism.

An Embedded Hypervisor Privilege exception occurs when MSR_{GS} = 1 and MSR_{PR} = 0 and execution is attempted of any of the following:

- a hypervisor-privileged instruction
- an ***mtspr*** or ***mfspr*** instruction that specifies an SPR that is hypervisor privileged
- a ***tlbwe*** instruction and Category: Embedded.Hypervisor.LRAT is not implemented.
- a ***tlbwe***, ***tlbsrx.***, or ***tlbilx*** instruction and $EPCR_{DGTMI}$=1
- a ***tlbwe*** instruction that attempts to write a TLB entry for which TLB_{V}=1 and TLB_{IPROT}=1 when $MAS0_{WQ}$=0b00
- a ***tlbwe*** instruction that attempts to write a TLB entry when $MAS1_{V}$=1, $MAS1_{IPROT}$=1, and $MAS0_{WQ}$=0b00
- a ***mtpmr*** or ***mfpmr*** instruction and $MSRP_{PMMP}$ = 1
- a *Cache Locking* instruction and $MSRP_{UCLEP}$ = 1

An Embedded Hypervisor Privilege exception may occur for the following implementation dependent reasons when MSR_{GS} = 1 and MSR_{PR} = 0 and execution is attempted of any of the following:

- a ***tlbwe*** instruction that attempts to write a TLB entry for which TLB_{V}=0 and TLB_{IPROT}=1 when $MAS0_{WQ}$=0b00
- a ***tlbwe*** instruction that attempts to write a TLB entry when $MAS1_{V}$=0, $MAS1_{IPROT}$=1, and $MAS0_{WQ}$=0b00
- a ***tlbwe*** instruction that attempts to write a TLB entry for which TLB_{IPROT}=1 when $MAS0_{WQ}$=0b01
- a ***tlbwe*** instruction that attempts to write a TLB entry when $MAS1_{IPROT}$=1, and $MAS0_{WQ}$=0b01
- a ***tlbwe*** instruction that attempts to write a TLB entry when $MAS0_{HES}$=0
- a ***tlbwe*** instruction that attempts to write a TLB entry to an array that is disallowed by the implementation
- an implementation dependent instruction or SPR which is hypervisor privileged

An Embedded Hypervisor Privilege exception also occurs when execution is attempted of an ***ehpriv*** instruction, regardless of the state of the thread.

Execution of the instruction causing the interrupt is suppressed and SRR0, SRR1, and MSR are updated as follows:

SRR0 Set to the effective address of the instruction causing the Embedded Hypervisor Privilege interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage.

CE, ME,DE
Unchanged.

All other defined MSR bits set to 0.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:51}$ || 0x320. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR41_{48:59}$|| 0b0000.

7.6.32 LRAT Error Interrupt [Category: Embedded.Hypervisor.LRAT]

An LRAT Error interrupt occurs when no higher priority exception exists (see Section 7.9 on page 1190) and an LRAT Miss exception is presented to the interrupt mechanism.

An LRAT Miss exception is caused by either of the following.

- A ***tlbwe*** instruction is executed in guest supervisor state and the logical page number (RPN specified by MAS7 and MAS3 and page size specified by $MAS1_{TSIZE}$) does not match any valid entry in the LRAT.
- A Page Table translation is performed and the associated PTE_{ARPN} is to be treated as an LPN (the Embedded.Hypervisor category is supported) and the logical page number (RPN based on PTE_{ARPN} and page size specified by PTE_{PS}) does not match any valid entry in the LRAT.

When an LRAT Error interrupt occurs, the hardware suppresses the execution of the instruction causing the LRAT Error interrupt.

SRR0, SRR1, MSR, ESR, and LPER are updated as follows:

SRR0 Set to the effective address of the instruction causing the LRAT Error interrupt.

SRR1 Set to the contents of the MSR at the time of the interrupt.

MSR

CM MSR_{CM} is set to $EPCR_{ICM}$.
CE, ME, DE Unchanged.

All other defined MSR bits are set to 0.

DEAR If the LRAT Error interrupt occurred for a Page Table translation, set to the effective address of a byte that is both within the range of the bytes being accessed by the *Storage Access* or *Cache Management* instruction, and within the page whose access caused the LRAT Miss exception. Otherwise, undefined.

ESR

FP Set to 1 if the instruction causing the interrupt is a floating-point load or store and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.

ST Set to 1 if the instruction causing the interrupt is a *Store* or 'store-class' *Cache Management* instruction and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.

AP Set to 1 if the instruction causing the interrupt is an Auxiliary Processor load or store and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.

SPV Set to 1 if the instruction causing the interrupt is a SPE operation or a Vector operation, the instruction is a *Load* or *Store*, and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.

DATA Set to 1 if the interrupt is due to an LRAT miss resulting from a Page Table translation of a Lo*ad, Store* or *Cache Management* operand address; otherwise set to 0.

TLBI Set to 1 if a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt; otherwise set to 0.

PT Set to 1 if the cause of the interrupt is an LRAT miss exception on a Page Table translation. Set to 0 if the cause of the interrupt is an LRAT miss exception on a ***tlbwe***.

VLEMI Set to 1 if the instruction causing the interrupt resides in VLE storage, the instruction is a *Load*, *Store*, or *Cache Management* instruction, and the translation of the operand address causes the LRAT Miss exception.

EPID Set to 1 if the instruction causing the interrupt is an External Process ID instruction, the instruction is a *Load*, *Store*, or *Cache Management* instruction, and the translation of the operand address causes the LRAT Miss exception; otherwise set to 0.

All other defined ESR bits are set to 0.

LPER

Set to the values of the ARPN and PS fields from the PTE that was used to translate a virtual address for an instruction fetch, Load, Store or Cache Management instruction that caused an LRAT Error interrupt as a result of an LRAT Miss exception.

If Interrupt Fixed Offsets [Category: Embedded.Phased-In] are supported, instruction execution resumes at address $IVPR_{0:47}$ || 0x0340. Otherwise, instruction execution resumes at address $IVPR_{0:47}$ || $IVOR42_{48:59}$|| 0b0000.

7.7 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then to be restarted from the beginning upon return from the interrupt. Unaligned *Load* and *Store* instructions, or *Load Multiple*, *Store Multiple*, *Load String*, and *Store String* instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. In order to guarantee that a particular load or store instruction will complete without being interrupted and restarted, software must mark the storage being referred to as Guarded, and must use an elementary (non-string or non-multiple) load or store that is aligned on an operand-sized boundary.

In order to guarantee that *Load* and *Store* instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an execution is partially executed and then interrupted:

- For an elementary *Load*, no part of the target register RT or FRT, will have been altered.
- For *'with update'* forms of *Load* or *Store*, the update register, register RA, will not have been altered.

On the other hand, the following effects are permissible when certain instructions are partially executed and then restarted:

- For any *Store*, some of the bytes at the target storage location may have been altered (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for *Store Conditional* instructions, CR0 has been set to an undefined value, and it is undefined whether the reservation has been cleared.
- For any *Load*, some of the bytes at the addressed storage location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism).
- For *Load Multiple* or *Load String*, some of the registers in the range to be loaded may have been altered. Including the addressing registers (RA, and possibly RB) in the range to be loaded is a programming error, and thus the rules for partial execution do not protect against overwriting of these registers.

In no case will access control be violated.

As previously stated, the only load or store instructions that are guaranteed to not be interrupted after being partially executed are elementary, aligned, guarded loads and stores. All others may be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types that could cause the interruption:

1. Any *Load* or *Store* (except elementary, aligned, guarded):
   Any asynchronous interrupt
   Machine Check
   Program (Imprecise Mode Floating-Point Enabled)
   Program (Imprecise Mode Auxiliary Processor Enabled)
2. Unaligned elementary *Load* or *Store*, or any multiple or string:
   All of the above listed under item 1, plus the following:
   Data Storage (if the access crosses a protection boundary)
   Debug (Data Address Compare)
3. ***mtcrf*** may also be partially executed due to the occurrence of any of the interrupts listed under item 1 at the time the ***mtcrf*** was executing.
   - All instructions prior to the ***mtcrf*** have completed execution. (Some storage accesses generated by these preceding instructions may not have completed.)
   - No subsequent instruction has begun execution.
   - The ***mtcrf*** instruction (the address of which was saved in SRR0/CSRR0/MCSRR0/DSRR0 [Category: Embedded.Enhanced Debug] at the occurrence of the interrupt), may appear not to have begun or may have partially executed.

7.8 Interrupt Ordering and Masking

It is possible for multiple exceptions to exist simultaneously, each of which could cause the generation of an interrupt. Furthermore, for interrupts classes other than the Machine Check interrupt and critical interrupts, the architecture does not provide for reporting more than one interrupt of the same class (unless the Embedded.Enhanced Debug category is supported). Therefore, the architecture defines that interrupts are ordered with respect to each other, and provides a masking mechanism for certain persistent interrupt types.

When an interrupt is masked (disabled), and an event causes an exception that would normally generate the interrupt, the exception *persists* as a *status* bit in a register (which register depends upon the exception type). However, no interrupt is generated. Later, if the interrupt is enabled (unmasked), and the exception status has not been cleared by software, the interrupt due to the original exception event will then finally be generated.

All asynchronous interrupts can be masked. In addition, certain synchronous interrupts can be masked. An example of such an interrupt is the Floating-Point Enabled exception type Program interrupt. The execution of a floating-point instruction that causes the $FPSCR_{FEX}$ bit to be set to 1 is considered an exception event, regardless of the setting of $MSR_{FE0,FE1}$. If $MSR_{FE0,FE1}$ are both 0, then the Floating-Point Enabled exception type of Program interrupt is masked, but the exception persists in the $FPSCR_{FEX}$ bit. Later, if the $MSR_{FE0,FE1}$ bits are enabled, the interrupt will finally be generated.

The architecture enables implementations to avoid situations in which an interrupt would cause the state information (saved in Save/Restore Registers) from a previous interrupt to be overwritten and lost. In order to do this, the architecture defines interrupt classes in a hierarchical manner. At each interrupt class, hardware automatically disables any further interrupts associated with the interrupt class by masking the interrupt enable in the MSR when the interrupt is taken. In addition, each interrupt class masks the interrupt enable in the MSR for each lower class in the hierarchy. The hierarchy of interrupt classes is as follows from highest to lowest:

Interrupt Class	MSR Enables Cleared	Save/Restore Registers
Machine Check	ME,DE, CE, EE	MSRR0/1
Debug[1]	DE,CE,EE	DSRR0/1
Critical	CE,EE	CSRR0/1
Base	EE	SRR0/1
Guest <E.HV>	EE	GSRR0/1

[1] The Debug interrupt class is Category: E.ED. Note: MSR_{DE} may be cleared when a critical interrupt occurs if Category: E.ED is not supported.

Figure 71. Interrupt Hierarchy

[Category: Embedded.Hypervisor]
The masking of interrupts is affected by MSR_{GS} and whether the interrupt is directed to the guest supervisor state or the hypervisor state. In general, interrupts directed to the hypervisor state (with the exception of Guest Processor Doorbell type interrupts), are enabled if MSR_{GS} = 1 regardless of the value of other MSR enables. Interrupts directed to the guest supervisor state are enabled if the associated MSR enables are set and MSR_{GS} = 1.

If the Embedded.Enhanced Debug category is not supported (or is supported and is not enabled), then the Debug interrupt becomes a Critical class interrupt and all critical class interrupts will clear DE, CE, and EE in the MSR.

Base Class interrupts that occur as a result of precise exceptions are not masked by the EE bit in the MSR and any such exception that occurs prior to software saving the state of SRR0/1 in a base class exception handler will result in a situation that could result in the loss of state information.

This first step of the hardware clearing the MSR enable bits lower in the hierarchy shown in Figure 71 prevents any subsequent asynchronous interrupts from overwriting the Save/Restore Registers (SRR0/SRR1, CSRR0/CSRR1, MCSRR0/MCSRR1, or DSRR0/DSRR1 [Category: Embedded.Enhanced Debug]), prior to software being able to save their contents. Hardware also automatically clears, on any interrupt, $MSR_{PR,FP,FE0,FE1,IS,DS}$. The clearing of these bits assists in the avoidance of subsequent interrupts of certain other types. However, *guaranteeing* that interrupt classes lower in the hierarchy do not occur and thus do not overwrite the Save/Restore Registers (SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], or MCSRR0/MCSRR1) also requires the cooperation of system software. Specifically, system software must avoid the exe-

cution of instructions that could cause (or enable) a subsequent interrupt, if the contents of the Save/Restore Registers (SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1 [Category: Embedded.Enhanced Debug]), or MCSRR0/MCSRR1) have not yet been saved.

7.8.1 Guidelines for System Software

The following list identifies the actions that system software must avoid, prior to having saved the Save/Restore Registers' contents:

- Re-enabling an interrupt class that is at the same or a lower level in the interrupt hierarchy. This includes the following actions:
  - Re-enabling of MSR_{EE}
  - Re-enabling of $MSR_{CE,EE}$ in critical class interrupt handlers, and if the Embedded.Enhanced Debug category is not supported, re-enabling of MSR_{DE}.
  - [Category: Embedded.Enhanced Debug] Re-enabling of $MSR_{CE,EE,DE}$ in Debug class interrupt handlers
  - Re-enabling of $MSR_{EE,CE,DE,ME}$ in Machine Check interrupt handlers.
- Branching (or sequential execution) to addresses for which any of the following conditions are true.
  - The address is not mapped by the TLB or the Page Table or is mapped without UX=1 or SX=1 permission.
  - Both the Embedded.Hypervisor.LRAT and the Embedded Page.Table category are supported, MSR_{GS}=1, and the effective address is mapped by the Page Table but the LPN is not mapped by the LRAT.

  This prevents Instruction Storage, LRAT Error, and Instruction TLB Error interrupts.
- *Load*, *Store* or *Cache Management* instructions to addresses for which any of the following conditions are true.
  - The address is not mapped by the TLB or the Page Table or is mapped without the required access permissions.
  - Both the Embedded.Hypervisor.LRAT and the Embedded Page.Table category are supported, MSR_{GS}=1, and the effective address is mapped by the Page Table but the LPN is not mapped by the LRAT.

  This prevents Data Storage, LRAT Error, and Data TLB Error interrupts.
- Execution of any floating-point instruction

  This prevents Floating-Point Unavailable interrupts. Note that this interrupt would occur upon the execution of any floating-point instruction, due to the automatic clearing of MSR_{FP}. However, even if software were to re-enable MSR_{FP}, floating-point instructions must still be avoided in order to prevent Program interrupts due to various possible Program interrupt exceptions (Floating-Point Enabled, Unimplemented Operation).
- Re-enabling of MSR_{PR}

  This prevents Privileged Instruction exception type Program interrupts. Alternatively, software could re-enable MSR_{PR}, but avoid the execution of any privileged instructions.
- Execution of any Auxiliary Processor instruction

  This prevents Auxiliary Processor Unavailable interrupts, and Auxiliary Processor Enabled type and Unimplemented Operation type Program interrupts.
- Execution of any Illegal instructions

  This prevents Illegal Instruction exception type Program interrupts.
- Execution of any instruction that could cause an Alignment interrupt

  This prevents Alignment interrupts. Included in this category are any string or multiple instructions, and any unaligned elementary load or store instructions. See Section 7.6.7 on page 1170 for a complete list of instructions that may cause Alignment interrupts.

It is not necessary for hardware or software to avoid interrupts higher in the interrupt hierarchy (see Figure 71 on page 1187) from within interrupt handlers (and hence, for example, hardware does not automatically clear $MSR_{CE,ME,DE}$ upon a base class interrupt), since interrupts at each level of the hierarchy use different pairs of Save/Restore Registers to save the instruction address and MSR (i.e. SRR0/SRR1 for base class interrupts, and MCSRR0/MCSRR1,DSRR0/DSRR1 [Category: Embedded.Enhanced Debug], or CSRR0/CSRR1 for non-base class interrupts). The converse, however, is not true. That is, hardware and software must cooperate in the avoidance of interrupts lower in the hierarchy from occurring within interrupt handlers, even though the these interrupts use different Save/Restore Register pairs. This is because the interrupt higher in the hierarchy may have occurred from within a interrupt handler for an interrupt lower in the hierarchy prior to the interrupt handler having saved the Save/Restore Registers. Therefore, within an interrupt handler, Save/Restore Registers for all interrupts lower in the hierarchy may contain data that is necessary to the system software.

7.8.2 Interrupt Order

The following is a prioritized listing of the various enabled interrupts for which exceptions might exist simultaneously:

1. Synchronous (Non-Debug) Interrupts:
   - Data Storage
   - Instruction Storage
   - Alignment
   - Program
   - Embedded Hypervisor Privilege [Category: Embedded.Hypervisor]
   - Floating-Point Unit Unavailable
   - Auxiliary Processor Unavailable
   - Embedded Floating-Point Unavailable [Category: SP.Embedded Float_*]
   - SPE/Embedded Floating-Point/Vector Unavailable [Category: SP.Embedded Float_*]
   - Embedded Floating-Point Data [Category: SP.Embedded Float_*]
   - Embedded Floating-Point Round [Category: SP.Embedded Float_*]
   - System Call
   - Embedded Hypervisor System Call [Category: Embedded.Hypervisor]
   - Data TLB Error
   - Instruction TLB Error
   - LRAT Error [Category: Embedded.Hypervisor.LRAT]

   Only one of the above types of synchronous interrupts may have an existing exception generating it at any given time. This is guaranteed by the exception priority mechanism (see Section 7.9 on page 1190) and the requirements of the Sequential Execution Model.
2. Machine Check
3. Guest Processor Doorbell Machine Check [Category: Embedded.Hypervisor]
4. Debug
5. Critical Input
6. Watchdog Timer
7. Guest Watchdog Timer [Category: Embedded.Hypervisor]
8. Processor Doorbell Critical [Category: Embedded.Processor Control]
9. Guest Processor Doorbell Critical [Category: Embedded.Hypervisor]
10. External Input Category: Embedded.Processor Control
11. Fixed-Interval Timer Category: Embedded.Processor Control
12. Guest Fixed-Interval Timer Category: Embedded.Processor Control, Embedded.Hypervisor
13. Decrementer Category: Embedded.Processor Control
14. Guest Decrementer Category: Embedded.Processor Control, Embedded.Hypervisor
15. Processor Doorbell [Category: Embedded.Processor Control]
16. Guest Processor Doorbell [Category: Embedded.Hypervisor]
17. Embedded Performance Monitor

Even though, as indicated above, the base, synchronous exception types listed under item 1 are generated with higher priority than the non-base interrupt classes listed in items 2-6, the fact is that these base class interrupts will immediately be followed by the highest priority existing interrupt in items 2-5, without executing any instructions at the base class interrupt handler. This is because the base interrupt classes do not automatically disable the MSR mask bits for the interrupts listed in 2-5. In all other cases, a particular interrupt class from the above list will automatically disable any subsequent interrupts of the same class, as well as all other interrupt classes that are listed below it in the priority order.

7.9 Exception Priorities

All synchronous (precise and imprecise) interrupts are reported in program order, as required by the Sequential Execution Model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt will then be generated with all of those exception types reported cumulatively, in both the ESR, and any status registers associated with the particular exception type (e.g. the Floating-Point Status and Control Register).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction will be permitted to cause a *single* enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time, there exists for consideration only one of the synchronous interrupt types listed in item 1 of Section 7.8.2 on page 1189. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

Because unaligned *Load* and *Store* instructions, or *Load Multiple*, *Store Multiple*, *Load String*, and *Store Sting* instructions may be broken up into multiple, smaller accesses, and these accesses may be performed in any order. The exception priority mechanism applies to each of the multiple storage accesses in the order they are performed by the implementation.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled will have no effect on the generation of other exceptions for which the corresponding interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it will prevent the setting of that other exception, independent of whether that other exception's corresponding interrupt type is enabled or disabled.

Except as specifically noted, only one of the exception types listed for a given instruction type will be permitted to be generated at any given time. The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

Programming Note

Some exception types may even be mutually exclusive of each other and could otherwise be considered the same priority. In these cases, the exceptions are listed in the order suggested by the sequential execution model.

7.9.1 Exception Priorities for Defined Instructions

7.9.1.1 Exception Priorities for Defined Floating-Point Load and Store Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined *Floating-Point Load* and *Store* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error for instruction fetch [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)
7. Floating-Point Unavailable
8. Program (Unimplemented Operation)
9. Data TLB Error
10. Data Storage (all types)
11. Alignment
12. LRAT Error for data access [Categories: E.PT and E.HV.LRAT]
13. Debug (Data Address Compare)
14. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare), and is not causing any of the exceptions listed in items 2-11, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.2 Exception Priorities for Other Defined Load and Store Instructions and Defined Cache Management Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any other defined *Load* or *Store* instruction, or defined *Cache Management* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error for instruction fetch [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)

7. Program (Unimplemented Operation)
8. Embedded Hypervisor Privilege [Category: E.HV]
9. Data TLB Error
10. Data Storage (all types)
11. Alignment
12. LRAT Error for data access [Categories: E.PT and E.HV.LRAT]
13. Debug (Data Address Compare)
14. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Data Address Compare), and is not causing any of the exceptions listed in items 2-11, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.3 Exception Priorities for Other Defined Floating-Point Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined floating-point instruction other than a load or store.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Floating-Point Unavailable
7. Program (Unimplemented Operation)
8. Program (Floating-point Enabled)
9. Debug (Instruction Complete)

7.9.1.4 Exception Priorities for Defined Privileged Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined privileged instruction, except ***dcbi***, ***rfi***, and ***rfci*** instructions.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT] (for hardware tablewalk)
5. Program (Illegal Instruction, except for TLB management instructions with invalid MAS settings, see 9)
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Embedded Hypervisor Privilege [Category: E.HV]
9. Program (Illegal Instruction, special case for TLB management instructions with invalid MAS settings)
10. LRAT Error [Category: E.HV.LRAT] (for ***tlbwe***)
11. Debug (Instruction Complete)

For ***mtmsr***, ***mtspr*** (DBCR0, DBCR1, DBCR2), ***mtspr*** (TCR), and ***mtspr*** (TSR), if they are not causing Debug (Instruction Address Compare) nor Program (Privileged Instruction) exceptions, it is possible that they are simultaneously enabling (via mask bits) multiple existing exceptions (and at the same time possibly causing a Debug (Instruction Complete) exception). When this occurs, the interrupts will be handled in the order defined by Section 7.8.2 on page 1189.

7.9.1.5 Exception Priorities for Defined Trap Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined *Trap* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Unimplemented Operation)
7. Debug (Trap)
8. Program (Trap)
9. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Trap), and is not causing any of the exceptions listed in items 2-6, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.6 Exception Priorities for Defined System Call Instruction

The following prioritized list of exceptions may occur as a result of the attempted execution of a defined *System Call* instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Unimplemented Operation)
7. System Call
8. Embedded Hypervisor System Call [Category: E.HV]
9. Debug (Instruction Complete)

7.9.1.7 Exception Priorities for Defined Branch Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any defined branch instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)

6. Program (Unimplemented Operation)
7. Debug (Branch Taken)
8. Debug (Instruction Complete)

If the instruction is causing both a Debug (Instruction Address Compare) and a Debug (Branch Taken), and is not causing any of the exceptions listed in items 2-6, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.8 Exception Priorities for Defined Return From Interrupt Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of an ***rfi***, ***rfci***, ***rfmci***, ***rfdi*** [Category:Embedded.Enhanced Debug], ***rfgi*** [Category: Embedded.Hypervisor] instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Debug (Return From Interrupt)
9. Debug (Instruction Complete)

If the ***rfi*** or ***rfci***, ***rfmci***, or ***rfdi*** [Category: Embedded.Enhanced Debug] or ***rfgi*** [Category: Embedded.Hypervisor] instruction is causing both a Debug (Instruction Address Compare) and a Debug (Return From Interrupt), and is not causing any of the exceptions listed in items 2-6, it is permissible for both exceptions to be generated and recorded in the DBSR. A single Debug interrupt will result.

7.9.1.9 Exception Priorities for Other Defined Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of all other instructions not listed above.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. LRAT Error instruction fetch [Categories: E.PT and E.HV.LRAT]
5. Program (Illegal Instruction)
6. Program (Privileged Instruction)
7. Program (Unimplemented Operation)
8. Embedded Hypervisor Privilege <E.HV>
9. LRAT Error for data access for ***tlbwe*** [Category: E.HV.LRAT]
10. Debug (Instruction Complete)

7.9.2 Exception Priorities for Reserved Instructions

The following prioritized list of exceptions may occur as a result of the attempted execution of any reserved instruction.

1. Debug (Instruction Address Compare)
2. Instruction TLB Error
3. Instruction Storage Interrupt (all types)
4. Program (Illegal Instruction)

Chapter 8. Reset and Initialization

8.1 Background

This chapter describes the requirements for thread reset. This includes both the means of causing reset, and the specific initialization that is required to be performed automatically by the hardware. This chapter also provides an overview of the operations that should be performed by initialization software.

In general, the specific actions taken by a thread upon reset are implementation-dependent. Also, it is the responsibility of system initialization software to initialize the majority of thread and system resources after reset. Implementations are required to provide a minimum thread initialization such that this system software may be fetched and executed, thereby accomplishing the rest of system initialization.

8.2 Reset Mechanisms

This specification defines two mechanisms for internally invoking a thread reset operation using either the Watchdog Timer (see Section 9.11 on page 1208) or the Debug facilities using $DBCR0_{RST}$ (see Section 10.5.1.1 on page 1221). In addition, implementations will typically provide additional means for invoking a reset operation, using an external mechanism such as a signal pin, which when activated, causes the thread to be reset.

8.3 Thread State after Reset

The initial thread state is controlled by the register contents after reset. In general, the contents of most registers are undefined after reset.

The hardware is only guaranteed to initialize those registers (or specific bits in registers) which must be initialized in order for software to be able to reliably perform the rest of system initialization.

The Thread Enable Register, Machine State Register, Processor Version Register, and a TLB entry are updated as follows.

Thread Enable Register [Category: Embedded Multi-Threading]

The TEN is set to the value 0x0000_0000_0000_0001, indicating that only thread 0 is enabled.

Machine State Register

The state of the MSR for all threads is as shown in Figure 72.

Bit	Setting	Comments
CM	0	Computation Mode (set to 32-bit mode)
GS	0	Hypervisor state <E.HV>
UCLE	0	User Cache Locking Enable
SPV	0	SPE/Embedded Floating-Point/ Vector Unavailable
CE	0	Critical Input interrupts disabled
DE	0	Debug interrupts disabled
EE	0	External Input interrupts disabled
PR	0	Supervisor mode
FP	0	FP unavailable
ME	0	Machine Check interrupts disabled
FE0	0	FP exception type Program interrupts disabled
FE1	0	FP exception type Program interrupts disabled
IS	0	Instruction Address Space 0
DS	0	Data Address Space 0
PMM	0	Performance Monitor Mark

Figure 72. Machine State Register Initial Values

Logical Partition Identification Register [Category: Embedded.Hypervisor]

The Logical Partition Identification Register (LPIDR) is set to 0.

Processor Version Register

Implementation-Dependent. (This register is read-only, and contains a value which identifies the specific implementation)

TLB entry

A TLB entry (which entry is implementation-dependent) is initialized in an implementation-dependent manner that maps the last page in the implemented effective storage address space, with the following field settings:

Field	Setting	Comments
V	1	valid
EPN	see below	Represents the last page in effective address space
RPN	see below	Represents the last page in physical address space
TS	0	translation address space 0
IND <E.PT>	0	direct entry
TLPID <E.HV>	0	translation logical partition ID
TGS <E.HV>	0	translation hypervisor state
SIZE	?	smallest page size supported
W	?	implementation-dependent value
I	?	implementation-dependent value
M	?	implementation-dependent value
G	?	implementation-dependent value
E	?	implementation-dependent value
U0	?	implementation-dependent value
U1	?	implementation-dependent value
U2	?	implementation-dependent value
U3	?	implementation-dependent value
TID	?	implementation-dependent value, but page must be accessible
UX	?	implementation-dependent value
UR	?	implementation-dependent value
UW	?	implementation-dependent value
SX	1	page is execute accessible in supervisor mode
SR	1	page is read accessible in supervisor mode
SW	1	page is write accessible in supervisor mode
VLE	?	implementation-dependent value
ACM	?	implementation-dependent value
IPROT	?	implementation-dependent value
VF <E.HV>	0	no virtualization fault

Figure 73. TLB Initial Values

The initial settings of EPN and RPN are dependent upon the number of bits implemented in the EPN and RPN fields and the minimum page size supported by the implementation. For example, an implementation that supports 64KB pages as the smallest size and 32 bits of effective address would implement a 16 bit EPN and set the initial value of the EPN field of the TLB boot entry to 2^{16}-1 (0xFFFF) while an implementation that supports 4K pages as the smallest size and 32 bits of effective address would implement a 20 bit EPN and

set the initial value of the boot entry to 2^{20}-1 (0xFFFFF).

Instruction execution begins at the last word address of the page mapped by the boot TLB entry. Note that this address is different from the System Reset interrupt vector specified in Book III-S.

An implementation may provide additional methods for initializing the TLB entry used for initial boot by providing an implementation-dependent RPN, or initializing other TLB entries.

If Category: Embedded Multi-threading.Thread Management is not supported, instruction execution for other threads begins at the last word address of the effective address space; otherwise execution begins at the address specified by the NIA register corresponding to the thread.

8.4 Software Initialization Requirements

When reset occurs, the thread is initialized to a minimum configuration to start executing initialization code. Initialization code is necessary to complete the thread and system configuration. The initialization code described in this section is the minimum recommended for configuring the thread to run application code.

Initialization code should configure the following resources:

- Invalidate the instruction cache and data cache (implementation-dependent).
- Initialize system memory as required by the operating system or application code.
- Initialize the Interrupt Vector Prefix Register and Interrupt Vector Offset Register.
- Initialize other registers as needed by the system.
- Initialize off-chip system facilities.
- Dispatch the operating system or application code.

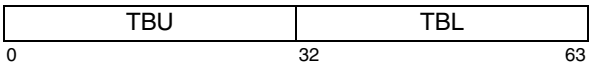
Chapter 9. Timer Facilities

9.1 Overview

The Time Base, Decrementer, Fixed-interval Timer, and Watchdog Timer provide timing functions for the system. The remainder of this section describes these registers and related facilities.

9.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 74) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.



Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 74. Time Base

The Time Base bits 0:59 increment until their value becomes 0xFFF_FFFF_FFFF_FFFF (2^{59} - 1), at the next increment their value becomes 0x000_0000_0000_0000. There is no interrupt or other indication when this occurs.

Time base bits 60:63 may increment at a variable rate. When the value of bit 59 changes, bits 60:63 are set to zero; if bits 60:63 increment to 0xF before the value of bit 59 changes, they remain at 0xF until the value of bit 59 changes.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 1 GHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{1\ \text{GHz}} = 5.90 \times 10^{11}\ \text{seconds}$$

which is approximately 18,700 years.

The Time Base is implemented such that:

1. Loading a GPR from the Time Base has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Time Base replaces the contents of the Time Base with the contents of the GPR.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a Power ISA system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base bits 0:59 changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base bits 0:59 is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in user mode (MSR_{PR}=1). If the means is under software control, it must be privileged. There must be a method for getting all Time Bases in the system to start incrementing with values that are identical or almost identical.

Programming Note

If software initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from 2^{64}-1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

See the description of the Time Base in Book II, for ways to compute time of day in POSIX format from the Time Base.

9.2.1 Writing the Time Base

Writing the Time Base is hypervisor privileged. Reading the Time Base is not privileged, it is discussed in Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The ***mttbl*** and ***mttbu*** extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the ***mtspr*** instruction; see Appendix B, "Assembler Extended Mnemonics" on page 1245.

The Time Base can be written by a sequence such as:

```
lwz    Rx,upper # load 64-bit value for
lwz    Ry,lower # TB into Rx and Ry
li     Rz,0
mttbl  Rz       # set TBL to 0
mttbu  Rx       # set TBU
mttbl  Ry       # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

Virtualized Implementation Note

In virtualized implementations, TBU and TBL are read-only.

Programming Note

The instructions for writing the Time Base are mode-independent. Thus code written to set the Time Base will work correctly in either 64-bit or 32-bit mode.

9.3 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

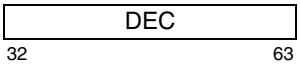


Figure 75. Decrementer

Decrementer bits 32:59 count down until their value becomes 0x000_0000, at the next increment their value becomes 0xFFF_FFFF. Decrementer bits 60:63 may decrement at a variable rate. When the value of bit 59 changes, bits 60:63 are set to 0xF; if bits 60:63 decrement to 0x0 before the value of bit 59 changes, they remain at 0x0 until the value of bit 59 changes.

The Decrementer is driven by the same frequency as the Time Base. The period of the Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 9.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{1\ \text{GHz}} = 137 \text{ seconds.}$$

The Decrementer counts down.

The operation of the Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
3. Copying the contents of a GPR to the Decrementer replaces the contents of the Decrementer with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

9.3.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are hypervisor privileged. When ***mfspr*** and ***mtspr*** are executed in guest supervisor state, the access to the DEC is mapped to the GDEC. Using an extended mnemonic (see Appendix B, “Assembler Extended Mnemonics” on page 1245), the Decrementer can be written from GPR Rx using:

```
mtdec  Rx
```

The Decrementer can be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

9.3.2 Decrementer Events

A Decrementer event occurs when a decrement occurs on a Decrementer value of 0x0000_0001.

Upon the occurrence of a Decrementer event, the Decrementer may be reloaded from a 32-bit Decrementer Auto-Reload Register (DECAR). See Section 9.5. Upon the occurrence of a Decrementer event, the Decrementer has the following basic modes of operation.

Decrement to one and stop on zero

If TCR_{ARE}=0, TSR_{DIS} is set to 1, the value 0x0000_0000 is then placed into the DEC, and the Decrementer stops decrementing.

A Decrementer interrupt occurs when no higher priority interrupt exists, a Decrementer exception exists, and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR_{DIE}=1 and (MSR_{EE}=1 or MSR_{GS}=1). Otherwise, the interrupt is enabled by TCR_{DIE}=1 and MSR_{EE}=1. See Section 7.6.12, “Decrementer Interrupt” on page 1173 for details of register behavior caused by the Decrementer interrupt.

Decrement to one and auto-reload

If TCR_{ARE}=1, TSR_{DIS} is set to 1, the contents of the Decrementer Auto-Reload Register is then placed into the DEC, and the Decrementer continues decrementing from the reloaded value.

A Decrementer interrupt occurs when no higher priority interrupt exists, a Decrementer exception exists, and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR_{DIE}=1 and (MSR_{EE}=1 or

MSR_{GS}=1). Otherwise, the interrupt is enabled by TCR_{DIE}=1 and MSR_{EE}=1. See Section 7.6.12, "Decrementer Interrupt" on page 1173 for details of register behavior caused by the Decrementer interrupt.

Forcing the Decrementer to 0 using the ***mtspr*** instruction will not cause a Decrementer exception; however, decrementing which was in progress at the instant of the ***mtspr*** may cause the exception. To eliminate the Decrementer as a source of exceptions, set TCR_{DIE} to 0 (clear the Decrementer Interrupt Enable bit).

If it is desired to eliminate all Decrementer activity, the procedure is as follows:

1. Write 0 to TCR_{DIE}. This will prevent Decrementer activity from causing exceptions.
2. Write 0 to TCR_{ARE} to disable the Decrementer auto-reload.
3. Write 0 to Decrementer. This will halt Decrementer decrementing. While this action will not cause a Decrementer exception to be set in TSR_{DIS}, a near simultaneous decrement may have done so.
4. Write 1 to TSR_{DIS}. This action will clear TSR_{DIS} to 0 (see Section 9.7.1 on page 1204). This will clear any Decrementer exception which may be pending. Because the Decrementer is frozen at zero, no further Decrementer events are possible.

If the auto-reload feature is disabled (TCR_{ARE}=0), then once the Decrementer decrements to zero, it will stay there until software reloads it using the ***mtspr*** instruction.

On reset, TCR_{ARE} is set to 0. This disables the auto-reload feature.

9.4 Guest Decrementer [Category: Embedded.Hypervisor]

The Guest Decrementer (GDEC) is a 32-bit decrementing counter that provides a mechanism for causing a Guest Decrementer interrupt after a programmable delay. The contents of the Guest Decrementer are treated as a signed integer.

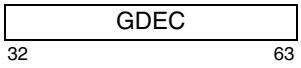


Figure 76. Guest Decrementer

Guest Decrementer bits 32:59 count down until their value becomes 0x000_0000, at the next increment their value becomes 0xFFF_FFFF. Guest Decrementer bits 60:63 may decrement at a variable rate. When the value of bit 59 changes, bits 60:63 are set to 0xF; if bits 60:63 decrement to 0x0 before the value of bit 59 changes, they remain at 0x0 until the value of bit 59 changes.

The Guest Decrementer is driven by the same frequency as the Time Base. The period of the Guest Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 9.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{1\ \text{GHz}} = 137 \text{ seconds.}$$

The Guest Decrementer counts down.

The operation of the Guest Decrementer satisfies the following constraints.

1. The operation of the Time Base and the Guest Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Guest Decrementer has no effect on the accuracy of the Time Base.
3. Copying the contents of a GPR to the Guest Decrementer replaces the contents of the Guest Decrementer with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Guest Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Guest Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

9.4.1 Writing and Reading the Guest Decrementer

The contents of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are supervisor privileged.

Copying the Guest Decrementer to a GPR has no effect on the Guest Decrementer contents or on the interrupt mechanism.

9.4.2 Guest Decrementer Events

A Guest Decrementer event occurs when a decrement occurs on a Guest Decrementer value of 0x0000_0001.

Upon the occurrence of a Guest Decrementer event, the Guest Decrementer may be reloaded from a 32-bit Guest Decrementer Auto-Reload Register (GDECAR). See see Section 9.6. Upon the occurrence of a Guest Decrementer event, the Guest Decrementer has the following basic modes of operation.

Decrement to one and stop on zero

If $GTCR_{ARE}$=0, $GTSR_{DIS}$ is set to 1, the value 0x0000_0000 is then placed into the GDEC, and the Guest Decrementer stops decrementing.

A Guest Decrementer interrupt occurs when no higher priority interrupt exists, a Guest Decrementer exception exists, and the exception is enabled. The interrupt is enabled by $GTCR_{DIE}$=1 and (MSR_{EE}=1 or MSR_{GS}=1). See Section 7.6.12, "Decrementer Interrupt" on page 1173 for details of register behavior caused by the Guest Decrementer interrupt.

Decrement to one and auto-reload

If $GTCR_{ARE}$=1, $GTSR_{DIS}$ is set to 1, the contents of the Guest Decrementer Auto-Reload Register is then placed into the GDEC, and the Guest Decrementer continues decrementing from the reloaded value.

A Guest Decrementer interrupt occurs when no higher priority interrupt exists, a Guest Decrementer exception exists, and the exception is enabled. The interrupt is enabled by $GTCR_{DIE}$=1 and (MSR_{EE}=1 or MSR_{GS}=1). See Section 7.6.12, "Decrementer Interrupt" on page 1173 for details of register behavior caused by the Guest Decrementer interrupt.

Forcing the Guest Decrementer to 0 using the ***mtspr*** instruction will not cause a Guest Decrementer exception; however, decrementing which was in progress at the instant of the ***mtspr*** may cause the exception. To eliminate the Guest Decrementer as a source of exceptions, set $GTCR_{DIE}$ to 0 (clear the Guest Decrementer Interrupt Enable bit).

If it is desired to eliminate all Guest Decrementer activity, the procedure is as follows:

1. Write 0 to $GTCR_{DIE}$. This will prevent Guest Decrementer activity from causing exceptions.
2. Write 0 to $GTCR_{ARE}$ to disable the Guest Decrementer auto-reload.
3. Write 0 to Guest Decrementer. This will halt Guest Decrementer decrementing. While this action will not cause a Guest Decrementer exception to be set in $GTSR_{DIS}$, a near simultaneous decrement may have done so.
4. Write 1 to $GTSR_{DIS}$. This action will clear $GTSR_{DIS}$ to 0 (see Section 9.8.1 on page 1207). This will clear any Guest Decrementer exception which may be pending. Because the Guest Decrementer is frozen at zero, no further Guest Decrementer events are possible.

If the auto-reload feature is disabled ($GTCR_{ARE}$=0), then once the Guest Decrementer decrements to zero, it will stay there until software reloads it using the ***mtspr*** instruction.

On reset, $GTCR_{ARE}$ is set to 0. This disables the auto-reload feature.

Programming Note

mfspr *RT,DEC* should be used to read GDEC in guest supervisor state. ***mtspr*** *DEC,RS* should be used to write GDEC in guest supervisor state.

9.5 Decrementer Auto-Reload Register

The Decrementer Auto-Reload Register is a 32-bit register as shown below.

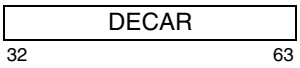


Figure 77. Decrementer Auto-Reload Register

Bits of the Decrementer Auto-Reload register are numbered 32 (most-significant bit) to 63 (least-significant bit). The Decrementer Auto-Reload Register is provided to support the auto-reload feature of the Decrementer. See Section 9.3.2

The contents of the Decrementer Auto-Reload Register cannot be read. The contents of bits 32:63 of register RS can be written to the Decrementer Auto-Reload Register using the ***mtspr*** instruction.

This register is hypervisor privileged.

9.6 Guest Decrementer Auto-Reload Register [Category:Embedded.Hypervisor]

The Guest Decrementer Auto-Reload Register is a 32-bit register as shown below.

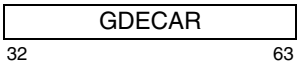


Figure 78. Guest Decrementer Auto-Reload Register

Bits of the Guest Decrementer Auto-Reload Register are numbered 32 (most-significant bit) to 63 (least-significant bit). The Guest Decrementer Auto-Reload Register is provided to support the auto-reload feature of the Guest Decrementer. See Section 9.4.2.

The contents of the Guest Decrementer Auto-Reload Register cannot be read. The contents of bits 32:63 of register RS can be written to the Guest Decrementer Auto-Reload Register using the ***mtspr*** instruction.

This register is hypervisor privileged.

Programming Note

mtspr *DECAR,RS* should be used to write GDECAR in guest supervisor state. Hypervisor software should emulate the accesses for the guest.

9.7 Timer Control Register

The Timer Control Register (TCR) is a 32-bit register. Timer Control Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Control Register controls Decrementer (see Section 9.3), Fixed-Interval Timer (see Section 9.9), and Watchdog Timer (see Section 9.11) options.

The relationship of the Timer facilities to the TCR and TB is shown in the figure below.

This register is hypervisor privileged. In guest supervisor state, the access to the TCR is mapped to the GTCR.

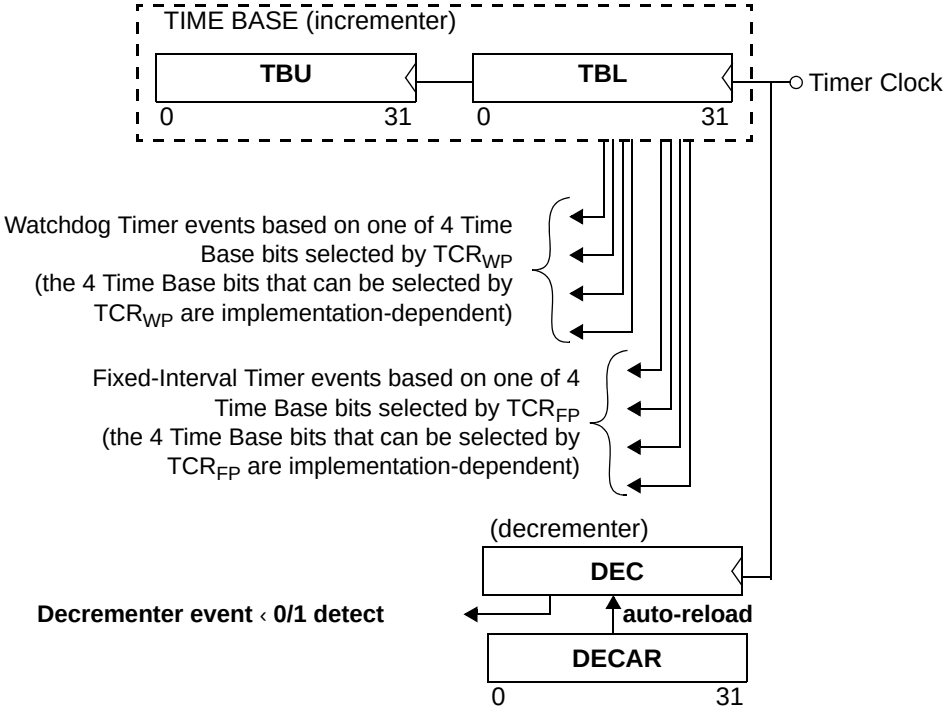


Figure 79. Relationships of the Timer Facilities

The contents of the Timer Control Register can be read using the ***mfspr*** instruction. The contents of bits 32:63 of register RS can be written to the Timer Control Register using the ***mtspr*** instruction.

The contents of the TCR are defined below:

Bit(s) Description

32:33 ***Watchdog Timer Period*** (WP) (see Section 9.11 on page 1208)

Specifies one of 4 bit locations of the Time Base used to signal a Watchdog Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Watchdog Timer period are implementation-dependent.

34:35 ***Watchdog Timer Reset Control*** (WRC) (see Section 9.11 on page 1208)

00 No Watchdog Timer reset will occur.

TCR_{WRC} resets to 0b00.

01-11
Force thread to be reset on second time-out of Watchdog Timer. The exact function of any of these settings is implementation-dependent.

Architecture Note

In previous versions of the architecture, it was not possible for software to clear WRC. That limitation has been removed.

36 ***Watchdog Timer Interrupt Enable*** (WIE) (see Section 9.11 on page 1208)

0 Disable Watchdog Timer interrupt
1 Enable Watchdog Timer interrupt

37 ***Decrementer Interrupt Enable*** (DIE) (see Section 9.3 on page 1199)

0 Disable Decrementer interrupt

1 Enable Decrementer interrupt

38:39 ***Fixed-Interval Timer Period*** (FP) (see Section 9.9 on page 1208)

Specifies one of 4 bit locations of the Time Base used to signal a Fixed-Interval Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Fixed-Interval Timer period are implementation-dependent.

40 ***Fixed-Interval Timer Interrupt Enable*** (FIE) (see Section 9.9 on page 1208)

0 Disable Fixed-Interval Timer interrupt
1 Enable Fixed-Interval Timer interrupt

41 ***Auto-Reload Enable*** (ARE)

0 Disable auto-reload of the Decrementer

Decrementer exception is presented (i.e. TSR_{DIS} is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The next value placed in the Decrementer is the value 0x0000_0000.

If Category: Embedded.Hypervisor is supported, when (MSR_{EE}=1 or MSR_{GS}=1), TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken.

If Category: Embedded.Hypervisor is not supported, when MSR_{EE}=1, TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken. Software must reset TSR_{DIS}.

1 Enable auto-reload of the Decrementer

Decrementer exception is presented (i.e. TSR_{DIS} is set to 1) when the Decrementer is decremented from a value of 0x0000_0001. The contents of the Decrementer Auto-Reload Register is placed in the Decrementer. The Decrementer resumes decrementing.

If Category: Embedded.Hypervisor is supported, when (MSR_{EE}=1 or MSR_{GS}=1), TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSR_{EE}=1, TCR_{DIE}=1, and TSR_{DIS}=1, a Decrementer interrupt is taken. Software must reset TSR_{DIS}.

42 Implementation-dependent

43:63 Reserved

9.7.1 Timer Status Register

The Timer Status Register (TSR) is a 32-bit register. Timer Status Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Timer Status Register contains status on timer events and the most recent Watchdog Timer-initiated thread reset.

The Timer Status Register is set via hardware, and read and cleared via software. The contents of the Timer Status Register can be read using the ***mfspr*** instruction. Bits in the Timer Status Register can be cleared using the ***mtspr*** instruction. Clearing is done by writing bits 32:63 of a General Purpose Register to the Timer Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The contents of the TSR are defined below:

Bit(s) Description

32 ***Enable Next Watchdog Timer*** (ENW) (see Section 9.11 on page 1208)

0 Action on next Watchdog Timer time-out is to set TSR_{ENW}
1 Action on next Watchdog Timer time-out is governed by TSR_{WIS}

33 ***Watchdog Timer Interrupt Status*** (WIS) (see Section 9.11 on page 1208)

0 A Watchdog Timer event has not occurred.
1 A Watchdog Timer event has occurred. If Category: Embedded.Hypervisor is supported, when (MSR_{CE}=1 or MSR_{GS}=1) and TCR_{WIE}=1, a Watchdog Timer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSR_{CE}=1 and TCR_{WIE}=1, a Watchdog Timer interrupt is taken.

34:35 ***Watchdog Timer Reset Status*** (WRS) (see Section 9.11 on page 1208)

These two bits are set to one of three values when a reset is caused by the Watchdog Timer. These bits are undefined at power-up.

00 No Watchdog Timer reset has occurred.
01 Implementation-dependent reset information.
10 Implementation-dependent reset information.
11 Implementation-dependent reset information.

36 ***Decrementer Interrupt Status*** (DIS) (see Section 9.3.2 on page 1199)

0 A Decrementer event has not occurred.
1 A Decrementer event has occurred. If Category: Embedded.Hypervisor is supported, when (MSR_{EE}=1 or MSR_{GS}=1) and TCR_{DIE}=1, a Decrementer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSR_{EE}=1 and TCR_{DIE}=1, a Decrementer interrupt is taken.

37 ***Fixed-Interval Timer Interrupt Status*** (FIS) (see Section 9.9 on page 1208)

- 0 A Fixed-Interval Timer event has not occurred.
- 1 A Fixed-Interval Timer event has occurred. If Category: Embedded.Hypervisor is supported, when (MSR_{EE}=1 or MSR_{GS}=1) and TCR_{FIE}=1, a Fixed-Interval Timer interrupt is taken. If Category: Embedded.Hypervisor is not supported, when MSR_{EE}=1 and TCR_{FIE}=1, a Fixed-Interval Timer interrupt is taken.

38:63 Reserved

This register is hypervisor privileged. In guest supervisor state, the access to the TSR is mapped to the GTSR.

9.8 Guest Timer Control Register [Category: Embedded.Hypervisor]

The Guest Timer Control Register (GTCR) is a 32-bit register. Guest Timer Control Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Guest Timer Control Register controls Guest Decrementer (see Section 9.4), Guest Fixed-Interval Timer (see Section 9.10), and Watchdog Timer (see Section 9.11) options.

The relationship of the Guest Timer facilities to the GTCR and TB is shown in the figure below.

This register is supervisor privileged.

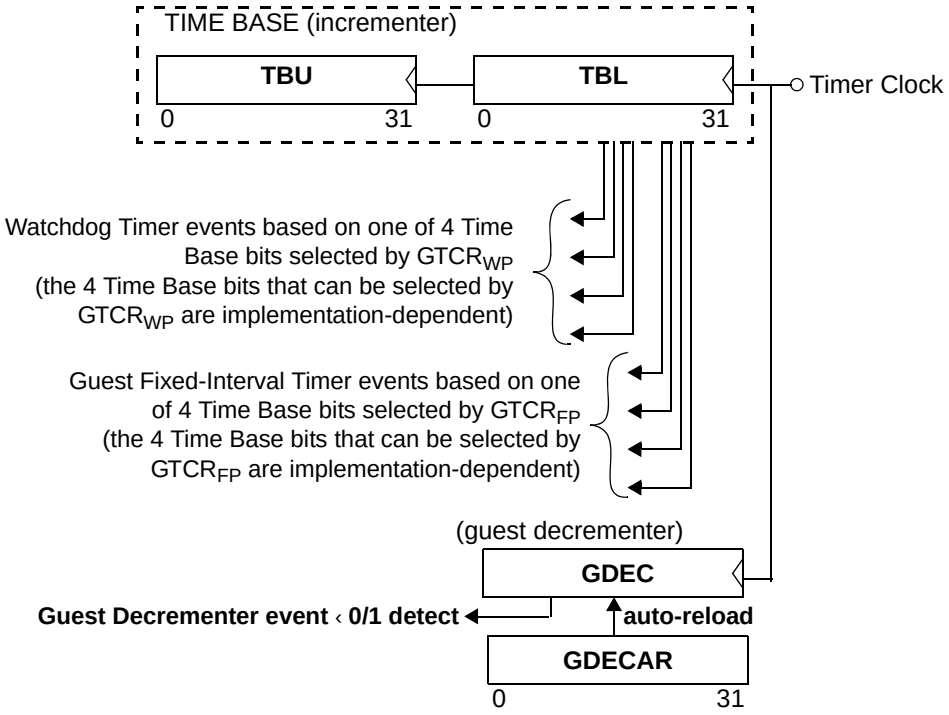


Figure 80. Relationships of the Guest Timer Facilities

The contents of the Guest Timer Control Register can be read using the ***mfspr*** instruction. The contents of bits 32:63 of register RS can be written to the Timer Control Register using the ***mtspr*** instruction.

The contents of the GTCR are defined below:

Bit(s) Description

32:33 ***Guest Watchdog Timer Period*** (WP) (see Section 9.11 on page 1208)

Specifies one of 4 bit locations of the Time Base used to signal a Guest Watchdog Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Guest Watchdog Timer period are implementation-dependent.

34:35 ***Guest Watchdog Timer Reset Control*** (WRC) (see Section 9.11 on page 1208)

00 No Guest Watchdog Timer reset will occur

GTCR$_{WRC}$ resets to 0b00.

01-11
Force thread to signal a Watchdog Timer exception to the hypervisor on second time[illegible] g Timer.

Architecture Note

In previous versions of the architecture, it was not possible for software to clear WRC. That limitation has been removed.

36 ***Guest Watchdog Timer Interrupt Enable*** (WIE) (see Section 9.11 on page 1208)

0 Disable Guest Watchdog Timer interrupt
1 Enable Guest Watchdog Timer interrupt

37 ***Guest Decrementer Interrupt Enable*** (DIE) (see Section 9.3 on page 1199)

0 Disable Guest Decrementer interrupt

1 Enable Guest Decrementer interrupt

38:39 ***Guest Fixed-Interval Timer Period*** (FP) (see Section 9.9 on page 1208)

Specifies one of 4 bit locations of the Time Base used to signal a Guest Fixed-Interval Timer exception on a transition from 0 to 1. The 4 Time Base bits that can be specified to serve as the Guest Fixed-Interval Timer period are implementation-dependent.

40 ***Guest Fixed-Interval Timer Interrupt Enable*** (FIE) (see Section 9.9 on page 1208

0 Disable Guest Fixed-Interval Timer interrupt
1 Enable Guest Fixed-Interval Timer interrupt

41 ***Guest Auto-Reload Enable*** (ARE)

0 Disable auto-reload of the Guest Decrementer

Guest Decrementer exception is presented (i.e. $GTSR_{DIS}$ is set to 1) when the Guest Decrementer is decremented from a value of 0x0000_0001. The next value placed in the Guest Decrementer is the value 0x0000_0000. When (MSR_{EE}=1 and MSR_{GS}=1), $GTCR_{DIE}$=1, and $GTSR_{DIS}$=1, a Guest Decrementer interrupt is taken. Software must reset $GTSR_{DIS}$.

1 Enable auto-reload of the Guest Decrementer

Guest Decrementer exception is presented (i.e. $GTSR_{DIS}$ is set to 1) when the Guest Decrementer is decremented from a value of 0x0000_0001. The contents of the Guest Decrementer Auto-Reload Register is placed in the Guest Decrementer. When (MSR_{EE}=1 and MSR_{GS}=1), $GTCR_{DIE}$=1, and $GTSR_{DIS}$=1, a Guest Decrementer interrupt is taken. Software must reset $GTSR_{DIS}$.

42 Implementation-dependent

43:63 Reserved

Programming Note

mfspr *RT,TCR* should be used to read GTCR in guest supervisor state. ***mtspr*** *TCR,RS* should be used to write GTCR in guest supervisor state.

9.8.1 Guest Timer Status Register [Category: Embedded.Hypervisor]

The Guest Timer Status Register (GTSR) is a 32-bit register. Guest Timer Status Register bits are numbered 32 (most-significant bit) to 63 (least-significant bit). The Guest Timer Status Register contains status on timer events and the most recent Watchdog Timer-initiated thread reset.

The Guest Timer Status Register is set via hardware, and read and cleared via software. The contents of the Guest Timer Status Register can be read using the ***mfspr*** instruction. Bits in the Guest Timer Status Register can be cleared using the ***mtspr*** instruction. Clearing is done by writing bits 32:63 of a General Purpose Register to the Guest Timer Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the Guest Timer Status Register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The contents of the GTSR are defined below:

Bit(s) Description

32 ***Enable Next Guest Watchdog Timer*** (ENW) (see Section 9.11 on page 1208)

0 Action on next Guest Watchdog Timer time-out is to set $GTSR_{ENW}$
1 Action on next Guest Watchdog Timer time-out is governed by $GTSR_{WIS}$

33 ***Guest Watchdog Timer Interrupt Status*** (WIS) (see Section 9.11 on page 1208)

0 A Guest Watchdog Timer event has not occurred.
1 A Guest Watchdog Timer event has occurred. When (MSR_{CE}=1 and MSR_{GS}=1) and $GTCR_{WIE}$=1, a Guest Watchdog Timer interrupt is taken.

34:35 ***Guest Watchdog Timer Reset Status*** (WRS) (see Section 9.11 on page 1208)

These two bits are set to one of three values when a reset is caused by the Guest Watchdog Timer. These bits are undefined at power-up.

00 No Guest Watchdog Timer reset has occurred.
01 Implementation-dependent reset information.
10 Implementation-dependent reset information.
11 Implementation-dependent reset information.

36 ***Guest Decrementer Interrupt Status*** (DIS) (see Section 9.4.2 on page 1200)

0 A Guest Decrementer event has not occurred.
1 A Guest Decrementer event has occurred. When MSR_{EE}=1 and MSR_{GS}=1 and $GTCR_{DIE}$=1, a Guest Decrementer interrupt is taken.

37 ***Guest Fixed-Interval Timer Interrupt Status*** (FIS) (see Section 9.10 on page 1208)

0 A Guest Fixed-Interval Timer event has not occurred.

1 A Guest Fixed-Interval Timer event has occurred. When (MSR_{EE}=1 and MSR_{GS}=1) and $GTCR_{FIE}$=1, a Guest Fixed-Interval Timer interrupt is taken.

38:63 Reserved

This register is supervisor privileged.

Programming Note

mfspr *RT,TSR* should be used to read GTSR in guest supervisor state. ***mtspr*** *TSR,RS* should be used to write GTSR in guest supervisor state.

9.8.2 Guest Timer Status Register Write Register (GTSRWR) [Category: Embedded.Hypervisor]

The Guest Timer Status Register Write Register (GTSRWR) allows a hypervisor state program to write the contents of the Guest Timer Status Register (see Section 9.8.1). The format of the GTSRWR is shown in Figure 81 below..

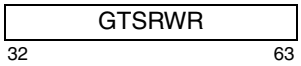


Figure 81. Guest Timer Status Register Write Register

The GTSRWR is provided as a means to restore the contents of the GTSR on a partition switch.

Writing GTSRWR changes the value in the GTSR. Writing non-zero bits may cause a Guest Decrementer or Fixed-Interval Timer exception.

This register is hypervisor privileged.

Programming Note

Hypervisors must ensure that a partition swap does not cause missing timer events to occur in guests. Upon partition restore, the hypervisor must set the appropriate status conditions in the GTSR.

9.9 Fixed-Interval Timer

The Fixed-Interval Timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload Decrementer, except that there are fewer selections of interrupt period available. The Fixed-Interval Timer exception occurs on 0 to 1 transitions of a selected bit from the Time Base (see Section 9.7).

The Fixed-Interval Timer exception is logged by TSR_{FIS}. A Fixed-Interval Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Fixed-Interval Timer exception exists (TSR_{FIS}=1), and the exception is enabled. If category mbedded.Hypervisor is supported, the interrupt is enabled by TCR_{FIE}=1 and (MSR_{EE} = 1 or MSR_{GS}=1). Otherwise, the interrupt is enabled by TCR_{FIE}=1 and MSR_{EE}=1. See Section 7.6.14 on page 1174 for details of register behavior caused by the Fixed-Interval Timer interrupt.

Note that a Fixed-Interval Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an ***mtspr*** instruction that writes a 1 to the bit when its previous value was 0.

9.10 Guest Fixed-Interval Timer [Category: Embedded.Hypervisor]

The Guest Fixed-Interval Timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload Guest Decrementer, except that there are fewer selections of interrupt period available. The Guest Fixed-Interval Timer exception occurs on 0 to 1 transitions of a selected bit from the Time Base (see Section 9.7).

The Guest Fixed-Interval Timer exception is logged by $GTSR_{FIS}$. A Guest Fixed-Interval Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Fixed-Interval Timer exception exists ($GTSR_{FIS}$=1), and the exception is enabled. The interrupt is enabled by $GTCR_{FIE}$=1 and (MSR_{EE} = 1 and MSRGS=1). See Section 7.6.14 on page 1174 for details of register behavior caused by the Fixed-Interval Timer interrupt.

Note that a Guest Fixed-Interval Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an ***mtspr*** instruction that writes a 1 to the bit when its previous value was 0.

9.11 Watchdog Timer

The Watchdog Timer is a facility intended to aid system recovery from faulty software or hardware. Watchdog time-outs occur on 0 to 1 transitions of selected bits from the Time Base (Section 9.7).

When a Watchdog Timer time-out occurs while Watchdog Timer Interrupt Status is clear (TSR_{WIS} = 0) and the next Watchdog Time-out is enabled (TSR_{ENW} = 1), a Watchdog Timer exception is generated and logged by setting TSR_{WIS} to 1. This is referred to as a Watchdog Timer First Time Out. A Watchdog Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Watchdog Timer excep-

tion exists (TSR_{WIS}=1), and the exception is enabled. If Category: Embedded.Hypervisor is supported, the interrupt is enabled by TCR_{WIE}=1 and (MSR_{CE} = 1 or MSR_{GS}=1). Otherwise, the interrupt is enabled by TCR_{WIE}=1 and MSR_{CE}=1. See Section 7.6.16 on page 1175 for details of register behavior caused by the Watchdog Timer Interrupt. The purpose of the Watchdog Timer First time-out is to give an indication that there may be problem and give the system a chance to perform corrective action or capture a failure before a reset occurs from the Watchdog Timer Second time-out as explained further below.

Note that a Watchdog Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an ***mtspr*** instruction that writes a 1 to the bit when its previous value was 0.

When a Watchdog Timer time-out occurs while TSR_{WIS} = 1 and TSR_{ENW} = 1, a thread reset occurs if it is enabled by a non-zero value of the Watchdog Reset Control field in the Timer Control Register (TCR_{WRC}). This is referred to as a Watchdog Timer Second Time Out. The assumption is that TSR_{WIS} was not cleared because the thread was unable to execute the Watchdog Timer interrupt handler, leaving reset as the only available means to restart the system.

A more complete view of Watchdog Timer behavior is afforded by Figure 82 and Figure 83, which describe the Watchdog Timer state machine and Watchdog Timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.

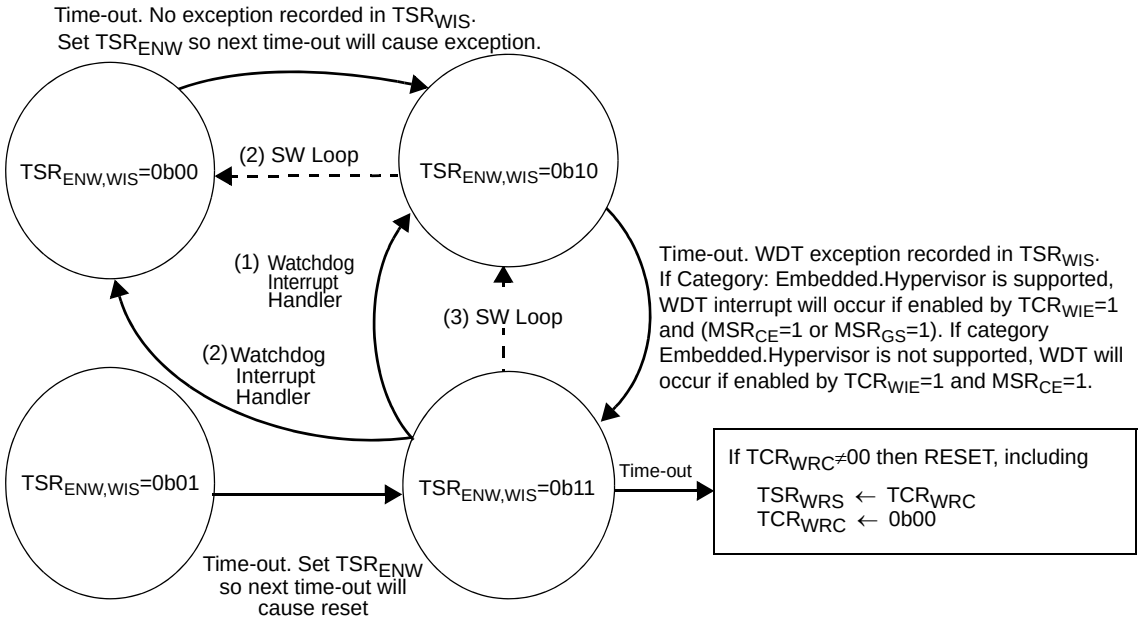


Figure 82. Watchdog State Machine

Enable Next WDT (TSR_{ENW})	WDT Status (TSR_{WIS})	Action when timer interval expires
0	0	Set Enable Next Watchdog Timer (TSR_{ENW}=1).
0	1	Set Enable Next Watchdog Timer (TSR_{ENW}=1).
1	0	Set Watchdog Timer interrupt status bit (TSR_{WIS}=1). If Category: Embedded.Hypervisor is supported and Watchdog Timer interrupt is enabled (TCR_{WIE}=1 and (MSR_{CE}=1 or MSR_{GS}=1)), then interrupt.If Category: Embedded.Hypervisor is not supported and Watchdog Timer interrupt is enabled (TCR_{WIE}=1 and MSR_{CE}=1), then interrupt.
1	1	Cause Watchdog Timer reset action specified by TCR_{WRC}. Reset will copy pre-reset TCR_{WRC} into TSR_{WRS}, then clear TCR_{WRC}.

Figure 83. Watchdog Timer Controls

The controls described in the above table imply three different modes of operation that a programmer might select for the Watchdog Timer. Each of these modes assumes that TCR_{WRC} has been set to allow thread reset by the Watchdog facility:

1. Always take the Watchdog Timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the Watchdog Timer interrupt caused by a first time-out is used to clear TSR_{WIS} so a second time-out never occurs. TSR_{ENW} is not cleared, thereby allowing the next time-out to cause another interrupt.
2. Always take the Watchdog Timer interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) is used to repeatedly clear TSR_{ENW} such that a first time-out exception is avoided, and thus no Watchdog Timer interrupt occurs. Once TSR_{ENW} has been cleared, software has between one and two full Watchdog periods before a Watchdog exception will be posted in TSR_{WIS}. If this occurs before the software is able to clear TSR_{ENW} again, a Watchdog Timer interrupt will occur. In this case, the Watchdog Timer interrupt handler will then clear both TSR_{ENW} and TSR_{WIS}, in order to (hopefully) avoid the next Watchdog Timer interrupt.
3. Never take the Watchdog Timer interrupt. In this mode, Watchdog Timer interrupts are disabled (via TCR_{WIE}=0), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Fixed-Interval Timer interrupt handler) to repeatedly clear TSR_{WIS} such that a second time-out is avoided, and thus no reset occurs. TSR_{ENW} is not cleared, thereby allowing the next time-out to set TSR_{WIS} again. The recurring code loop must have a period which is less than one Watchdog Timer period in order to guarantee that a Watchdog Timer reset will not occur.

9.12 Guest Watchdog Timer [Category: Embedded.Hypervisor]

The Guest Watchdog Timer is a facility intended to aid system recovery from faulty software or hardware. Guest Watchdog time-outs occur on 0 to 1 transitions of selected bits from the Time Base (Section 9.7).

When a Guest Watchdog Timer time-out occurs while Guest Watchdog Timer Interrupt Status is clear ($GTSR_{WIS}$ = 0) and the next Guest Watchdog Time-out is enabled ($GTSR_{ENW}$ = 1), a Guest Watchdog Timer exception is generated and logged by setting $GTSR_{WIS}$ to 1. This is referred to as a Guest Watchdog Timer First Time Out. A Guest Watchdog Timer interrupt occurs when no higher priority interrupt exists (see Section 7.9 on page 1190), a Guest Watchdog Timer exception exists ($GTSR_{WIS}$=1), and the exception is enabled. The interrupt is enabled by $GTCR_{WIE}$=1 and (MSR_{CE} = 1 and MSR_{GS}=1). See Section 7.6.17 on page 1176 for details of register behavior caused by the Guest Watchdog Timer Interrupt. The purpose of the Guest Watchdog Timer First time-out is to give an indication that there may be problem and give the system a chance to perform corrective action or capture a failure before a virtualized reset (a Watchdog Timer exception in the hypervisor) occurs from the Guest Watchdog Timer Second time-out as explained further below.

Note that a Guest Watchdog Timer exception will also occur if the selected Time Base bit transitions from 0 to 1 due to an ***mtspr*** instruction that writes a 1 to the bit when its previous value was 0.

A Guest Watchdog Timer Second Time Out results when a Guest Watchdog Timer time-out occurs while $GTSR_{WIS}$ = 1, $GTSR_{ENW}$ = 1, and a non-zero value is present in the Guest Watchdog Reset Control field in the Guest Timer Control Register ($GTCR_{WRC}$). In this case, a Watchdog Timer Interrupt is directed toward the hypervisor and the value set in $GTSR_{WRS}$ reflects the virtualized reset condition. The assumption is that $GTSR_{WIS}$ was not cleared because the guest was unable to execute the Guest Watchdog Timer interrupt handler, leaving a virtualized reset as the only available means to stop or restart the guest.

A more complete view of Guest Watchdog Timer behavior is afforded by Figure 84 and Figure 85, which describe the Guest Watchdog Timer state machine and Guest Watchdog Timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation which follow the table.

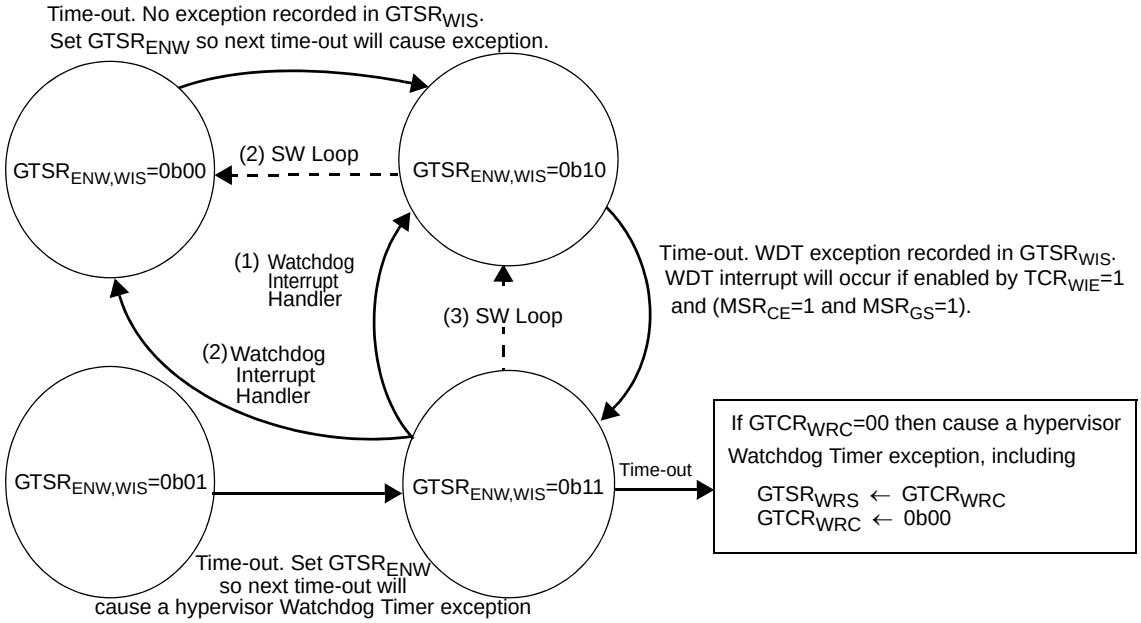


Figure 84. Guest Watchdog State Machine

Enable Next WDT ($GTSR_{ENW}$)	WDT Status ($GTSR_{WIS}$)	Action when timer interval expires
0	0	Set Enable Next Guest Watchdog Timer ($GTSR_{ENW}$=1).
0	1	Set Enable Next Guest Watchdog Timer ($GTSR_{ENW}$=1).
1	0	Set Guest Watchdog Timer interrupt status bit ($GTSR_{WIS}$=1). If Guest Watchdog Timer interrupt is enabled ($GTCR_{WIE}$=1 and (MSR_{CE}=1 and MSR_{GS}=1)), then interrupt.
1	1	Cause Guest Watchdog Timer virtualized reset action specified by $GTCR_{WRC}$. Virtualized reset will copy pre-reset $GTCR_{WRC}$ into $GTSR_{WRS}$, then clear $GTCR_{WRC}$.

Figure 85. Guest Watchdog Timer Controls

The controls described in the above table imply three different modes of operation that a programmer might select for the Guest Watchdog Timer. Each of these modes assumes that $GTCR_{WRC}$ has been set on a thread to allow virtualized reset by the Guest Watchdog facility:

1. Always take the Guest Watchdog Timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the Guest Watchdog Timer interrupt caused by a first time-out is used to clear $GTSR_{WIS}$ so a second time-out never occurs. $GTSR_{ENW}$ is not cleared, thereby allowing the next time-out to cause another interrupt.
2. Always take the Guest Watchdog Timer interrupt when pending, but avoid when possible. In this mode a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Guest Fixed-Interval Timer interrupt handler) is used to repeatedly clear $GTSR_{ENW}$ such that a first time-out exception is avoided, and thus no Guest Watchdog Timer interrupt occurs. Once $GTSR_{ENW}$ has been cleared, software has between one and two full Guest Watchdog periods before a Guest Watchdog exception will be posted in $GTSR_{WIS}$. If this occurs before the software is able to clear $GTSR_{ENW}$ again, a Guest Watchdog Timer interrupt will occur. In this case, the Guest Watchdog Timer interrupt handler will then clear both $GTSR_{ENW}$ and $GTSR_{WIS}$, in order to (hopefully) avoid the next Guest Watchdog Timer interrupt.
3. Never take the Guest Watchdog Timer interrupt. In this mode, Guest Watchdog Timer interrupts are disabled (via $GTCR_{WIE}$=0), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the Guest Fixed-Interval Timer interrupt handler) to repeatedly clear $GTSR_{WIS}$ such that a second time-out is avoided, and thus no virtualized reset occurs. $GTSR_{ENW}$ is not cleared, thereby allowing the next time-out to set $GTSR_{WIS}$ again. The recurring code loop must have a period which is less than one Guest Watchdog Timer period in order to guarantee that a Guest Watchdog Timer virtualized reset will not occur.

9.13 Freezing the Timer Facilities

The debug mechanism provides a means of temporarily freezing the timers upon a debug event. Whenever a debug event is set in the Debug Status Register, all timers will be frozen by preventing the Time Base from incrementing. This allows a debugger to simulate the appearance of ‘real time’, even though the application has been temporarily ‘halted’ to service the debug event. See the description of bit 63 of the Debug Control Register 0 (Freeze Timers on Debug Event or $DBCR0_{FT}$) in Section 10.5.1.1 on page 1221.

Chapter 10. Debug Facilities

10.1 Overview

Debug facilities are provided to enable hardware and software debug functions, such as instruction and data breakpoints and program single stepping. The debug facilities consist of a set of Debug Control Registers (DBCR0, DBCR1, and DBCR2) (see Section 10.5.1 on page 1221), a set of Address Compare Registers (IAC1, IAC2, IAC3, IAC4, DAC1, and DAC2), (see Section 10.4.3, Section 10.4.4, and Section 10.4.5), a Debug Status Register (DBSR) (see Section 10.5.2) for enabling and recording various kinds of debug events, and a special Debug interrupt type built into the interrupt mechanism (see Section 7.6.20). The debug facilities also provide a mechanism for software-controlled thread reset, and for controlling the operation of the timers in a debug environment.

The ***mfspr*** and ***mtspr*** instructions (see Section 5.4.1) provide access to the registers of the debug facilities.

In addition to the facilities described here, implementations will typically include debug facilities, modes, and access mechanisms which are implementation-specific. For example, implementations will typically provide access to the debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

10.2 Internal Debug Mode

Debug events include such things as instruction and data breakpoints. These debug events cause status bits to be set in the Debug Status Register. The existence of a set bit in the Debug Status Register is considered a Debug exception. Debug exceptions, if enabled, will cause Debug interrupts.

There are two different mechanisms that control whether Debug interrupts are enabled. The first is the MSR_{DE} bit, and this bit must be set to 1 to enable Debug interrupts. The second mechanism is an enable bit in the Debug Control Register 0 (DBCR0). This bit is the Internal Debug Mode bit ($DBCR0_{IDM}$), and it must also be set to 1 to enable Debug interrupts.

When $DBCR0_{IDM}$=1, the thread is in Internal Debug Mode. In this mode, debug events will (if also enabled by MSR_{DE}) cause Debug interrupts. Software at the Debug interrupt vector location will thus be given control upon the occurrence of a debug event, and can access (via the normal instructions) all architected resources. In this fashion, debug monitor software can control the thread and gather status, and interact with debugging hardware.

When the thread is not in Internal Debug Mode ($DBCR0_{IDM}$=0), debug events may still occur and be recorded in the Debug Status Register. These exceptions may be monitored via software by reading the Debug Status Register (using ***mfspr***), or may eventually cause a Debug interrupt if later enabled by setting $DBCR0_{IDM}$=1 (and MSR_{DE}=1). Behavior when debug events occur while $DBCR0_{IDM}$=0 is implementation-dependent.

10.3 External Debug Mode [Category: Embedded.Enhanced Debug]

The External Debug Mode is a mode in which external facilities can control execution and access registers and other resources. These facilities are defined as the *external debug* facilities and are not defined here, however some instructions and registers share internal and external debug roles and are briefly described as necessary.

A ***dnh*** instruction is provided to stop instruction fetching and execution and allow the thread to be managed by an external debug facility. After the ***dnh*** instruction is executed, instructions are not fetched, interrupts are not taken, and the thread does not execute instructions.

10.4 Debug Events

Debug events are used to cause Debug exceptions to be recorded in the Debug Status Register (see Section 10.5.2). In order for a debug event to be enabled to set a Debug Status Register bit and thereby

cause a Debug exception, the specific event type must be enabled by a corresponding bit or bits in the Debug Control Register DBCR0 (see Section 10.5.1.1), DBCR1 (see Section 10.5.1.2), or DBCR2 (see Section 10.5.1.3), in most cases; the Unconditional Debug Event (UDE) is an exception to this rule. Once a Debug Status Register bit is set, if Debug interrupts are enabled by MSR_{DE}, a Debug interrupt will be generated.

[Category: Embedded.Hypervisor]
To prevent spurious hypervisor debug events from occurring when a guest has been permitted to use the Debug facilities, if the thread is in hypervisor state (MSR_{GS}=0) and debug events are disabled for hypervisor ($EPCR_{DUVD}$=1), no debug events are allowed to occur except for the Unconditional Debug Event. It is implementation-dependent whether the Unconditional Debug Event is allowed to occur in hypervisor state when $EPCR_{DUVD}$=1.

Certain debug events are not allowed to occur when MSR_{DE}=0. In such situations, no Debug exception occurs and thus no Debug Status Register bit is set. Other debug events may cause Debug exceptions and set Debug Status Register bits regardless of the state of MSR_{DE}. The associated Debug interrupts that result from such Debug exceptions will be delayed until MSR_{DE}=1, provided the exceptions have not been cleared from the Debug Status Register in the meantime.

Any time that a Debug Status Register bit is allowed to be set while MSR_{DE}=0, a special Debug Status Register bit, Imprecise Debug Event ($DBSR_{IDE}$), will also be set. $DBSR_{IDE}$ indicates that the associated Debug exception bit in the Debug Status Register was set while Debug interrupts were disabled via the $MMSR_{DE}$ bit. Debug interrupt handler software can use this bit to determine whether the address recorded in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] should be interpreted as the address associated with the instruction causing the Debug exception, or simply the address of the instruction after the one which set the MSR_{DE} bit, thereby enabling the delayed Debug interrupt.

Debug interrupts are ordered with respect to other interrupt types (see Section 7.8 on page 179). Debug exceptions are prioritized with respect to other exceptions (see Section 7.9 on page 183).

There are eight types of debug events defined:

1. Instruction Address Compare debug events
2. Data Address Compare debug events
3. Trap debug events
4. Branch Taken debug events
5. Instruction Complete debug events
6. Interrupt Taken debug events
7. Return debug events
8. Unconditional debug events
9. Critical Interrupt Taken debug events [Category: Embedded.Enhanced Debug]
10. Critical Interrupt Return debug events [Category: Embedded.Enhanced Debug]

Programming Note

There are two classes of debug exception types:

Type 1: exception before instruction

Type 2: exception after instruction

Almost all debug exceptions fall into the first type. That is, they all take the interrupt upon encountering an instruction having the exception without updating any architectural state (other than DBSR, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug], CSRR1/ DSRR1 [Category: Embedded.Enhanced Debug], MSR) for that instruction.

The CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] for this type of exception points to the instruction that encountered the exception. This includes IAC, DAC, branch taken, etc.

The only exception which fall into the second type is the instruction complete debug exception. This exception is taken upon completing and updating one instruction and then pointing CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] to the next instruction to execute.

To make forward progress for any Type 1 debug exception one does the following:

1. Software sets up Type 1 exceptions (e.g. branch taken debug exceptions) and then returns to normal program operation
2. Hardware takes Debug interrupt upon the first branch taken Debug exception, pointing to the branch with CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].
3. Software, in the debug handler, sees the branch taken exception type, does whatever logging/analysis it wants to, then clears all debug event enables in the DBCR except for the instruction complete debug event enable.
4. Software does an ***rfci*** or ***rfdi*** [Category: Embedded.Enhanced Debug].
5. Hardware would execute and complete one instruction (the branch taken in this case), and then take a Debug interrupt with CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] pointing to the target of the branch.
6. Software would see the instruction complete interrupt type. It clears the instruction complete event enable, then enables the branch taken interrupt event again.
7. Software does an ***rfci*** *or* ***rfdi*** [Category: Embedded.Enhanced Debug].
8. Hardware resumes on the target of the taken branch and continues until another taken branch, in which case we end up at step 2 again.

This, at first, seems like a double tax (i.e. 2 debug interrupts for every instance of a Type 1 exception), but there doesn't seem like any other clean way to make forward progress on Type 1 debug exceptions. The only other way to avoid the double tax is to have the debug handler routine actually emulate the instruction pointed to for the Type 1 exceptions, determine the next instruction that would have been executed by the interrupted program flow and load the CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] with that address and do an ***rfci/rfdi*** [Category: Embedded.Enhanced Debug]; this is probably not faster.

10.4.1 Instruction Address Compare Debug Event

One or more Instruction Address Compare debug events (IAC1, IAC2, IAC3 or IAC4) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in the DBCR0, DBCR1, IAC1, IAC2, IAC3, and IAC4 Registers.

Instruction Address Compare User/ Supervisor Mode

$DBCR1_{IAC1US}$ specifies whether IAC1 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC2US}$ specifies whether IAC2 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC3US}$ specifies whether IAC3 debug events can occur in user mode or supervisor mode, or both.

$DBCR1_{IAC4US}$ specifies whether IAC4 debug events can occur in user mode or supervisor mode, or both.

Effective/Real Address Mode

$DBCR1_{IAC1ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC1 debug events.

$DBCR1_{IAC2ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or

effective addresses and MSR_{IS}=1 are used in determining an address match on IAC2 debug events.

$DBCR1_{IAC3ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC3 debug events.

$DBCR1_{IAC4ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{IS}=0, or effective addresses and MSR_{IS}=1 are used in determining an address match on IAC4 debug events.

Instruction Address Compare Mode

$DBCR1_{IAC12M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC1 or IAC2, whether the address must be inside a specific range specified by the IAC1 and IAC2 or outside a specific range specified by the IAC1 and IAC2 for an IAC1 or IAC2 debug event to occur.

$DBCR1_{IAC34M}$ specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC3 Register or IAC4 Register, whether the address must be inside a specific range specified by the IAC3 Register and IAC4 Register or outside a specific range specified by the IAC3 Register and IAC4 Register for an IAC3 or IAC4 debug event to occur.

There are four instruction address compare modes.

There are four instruction address compare modes.

- *Exact address compare mode*
  If the address of the instruction fetch is equal to the value in the enabled IAC Register, an instruction address match occurs. For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.
- *Address bit match mode*
  For IAC1 and IAC2 debug events, if the address of the instruction fetch access, ANDed with the contents of the IAC2, are equal to the contents of the IAC1, also ANDed with the contents of the IAC2, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the address of the instruction fetch, ANDed with the contents of the IAC4, are equal to the contents of the IAC3, also ANDed with the contents of the IAC4, an instruction address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.
- *Inclusive address range compare mode*
  For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the IAC1 and less than the contents of the IAC2, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is greater than or equal to the contents of the IAC3 and less than the contents of the IAC4, an instruction address match occurs.
- For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.
- *Exclusive address range compare mode*
  For IAC1 and IAC2 debug events, if the 64-bit address of the instruction fetch is less than the contents of the IAC1 or greater than or equal to the contents of the IAC2, an instruction address match occurs.

  For IAC3 and IAC4 debug events, if the 64-bit address of the instruction fetch is less than the contents of the IAC3 or greater than or equal to the contents of the IAC4, an instruction address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

See the detailed description of DBCR0 (see Section 10.5.1.1, “Debug Control Register 0 (DBCR0)” on page 1221) and DBCR1 (see Section 10.5.1.2, “Debug Control Register 1 (DBCR1)” on page 1222) and the modes for detecting IAC1, IAC2, IAC3 and IAC4 debug events. Instruction Address Compare debug events can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When an Instruction Address Compare debug event occurs, the corresponding $DBSR_{IAC1}$, $DBSR_{IAC2}$, $DBSR_{IAC3}$, or $DBSR_{IAC4}$ bit or bits are set to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the excepting instruction.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Instruction Address Compare debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt).

Later, if the debug exception has not been reset by clearing $DBSR_{IAC1}$, $DBSR_{IAC2}$, $DBSR_{IAC3}$, and $DBSR_{IAC4}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug.

10.4.2 Data Address Compare Debug Event

One or more Data Address Compare debug events (DAC1R, DAC1W, DAC2R, DAC2W) occur if they are enabled, execution is attempted of a data storage access instruction, and the type, address, and possibly even the data value of the data storage access meet the criteria specified in the Debug Control Register 0, Debug Control Register 2, and the DAC1 and DAC2 Registers.

Data Address Compare Read/Write Enable

$DBCR0_{DAC1}$ specifies whether DAC1R debug events can occur on read-type data storage accesses and whether DAC1W debug events can occur on write-type data storage accesses.

$DBCR0_{DAC2}$ specifies whether DAC2R debug events can occur on read-type data storage accesses and whether DAC2W debug events can occur on write-type data storage accesses.

Indexed-string instructions (***lswx***, ***stswx***) for which the XER field specifies zero bytes as the length of the string are treated as no-ops, and are not allowed to cause Data Address Compare debug events.

All *Load* instructions are considered reads with respect to debug events, while all *Store* instructions are considered writes with respect to debug events. In addition, the *Cache Management* instructions, and certain special cases, are handled as follows.

- ***dcbt***, ***dcbtls***, ***dcblq.***, ***dcbtep***, ***icbt***, ***icbtls***, ***icbi***, ***icblc***, ***icblq.***, ***dcblc***, and ***icbiep*** are all considered reads with respect to debug events. Note that ***dcbt***, ***dcbtep***, and ***icbt*** are treated as no-operations when they report Data Storage or Data TLB Miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause Debug interrupts, even when they would otherwise have been no-op'ed due to a Data Storage or Data TLB Miss exception.
- ***dcbtst, dcbtstls, dcbtstep, dcbz, dcbzep***, ***dcbi***, ***dcbf***, ***dcbfep***, ***dcba***, ***dcbst***, and ***dcbstep*** are all considered writes with respect to debug events. Note that ***dcbf, dcbfep, dcbst,*** and ***dcbstep*** are considered reads with respect to Data Storage exceptions, since they do not actually change the data at a given address. However, since the execution of these instructions may result in write activity on the data bus, they are treated as writes with respect to debug events. Note also that ***dcbtst*** and ***dcbtstep*** are treated as no-operations when they report Data Storage or Data TLB Miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause Debug interrupts, even when they would otherwise have been no-op'ed due to a Data Storage or Data TLB Miss exception.

Data Address Compare User/Supervisor Mode

$DBCR2_{DAC1US}$ specifies whether DAC1R and DAC1W debug events can occur in user mode or supervisor mode, or both.

$DBCR2_{DAC2US}$ specifies whether DAC2R and DAC2W debug events can occur in user mode or supervisor mode, or both.

Effective/Real Address Mode

$DBCR2_{DAC1ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{DS}=0, or effective addresses and MSR_{DS}=1 are used to in determining an address match on DAC1R and DAC1W debug events.

$DBCR2_{DAC2ER}$ specifies whether effective addresses, real addresses, effective addresses and MSR_{DS}=0, or effective addresses and MSR_{DS}=1 are used to in determining an address match on DAC2R and DAC2W debug events.

Data Address Compare Mode

$DBCR2_{DAC12M}$ specifies whether all or some of the bits of the address of the data storage access must match the contents of the DAC1 or DAC2, whether the address must be inside a specific range specified by the DAC1 and DAC2 or outside a specific range specified by the DAC1 and DAC2 for a DAC1R, DAC1W, DAC2R or DAC2W debug event to occur.

There are four data address compare modes.

- *Exact address compare mode*
  If the 64-bit address of the data storage access is equal to the value in the enabled Data Address Compare Register, a data address match occurs.

For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

- *Address bit match mode*
  If the address of the data storage access, ANDed with the contents of the DAC2, are equal to the contents of the DAC1, also ANDed with the contents of the DAC2, a data address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

- *Inclusive address range compare mode*
  If the 64-bit address of the data storage access is greater than or equal to the contents of the DAC1 and less than the contents of the DAC2, a data address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

- *Exclusive address range compare mode*
  If the 64-bit address of the data storage access is less than the contents of the DAC1 or greater than or equal to the contents of the DAC2, a data address match occurs.

  For 64-bit implementations, the addresses are masked to compare only bits 32:63 when the thread is executing in 32-bit mode.

The description of DBCR0 (see Section 10.5.1.1) and DBCR2 (see Section 10.5.1.3) and the modes for detecting Data Address Compare debug events. Data Address Compare debug events can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When an Data Address Compare debug event occurs, the corresponding $DBSR_{DAC1R}$, $DBSR_{DAC1W}$, $DBSR_{DAC2R}$, or $DBSR_{DAC2W}$ bit or bits are set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Data Address Compare debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), the execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will be set to the address of the excepting instruction. Depending on the type of instruction and/or the alignment of the data access, the instruction causing the exception may have been partially executed (see Section 7.7).

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Data Address Compare debug exception, a Debug interrupt will not occur, and the instruction will complete execution (provided the instruction is not causing some other exception which will generate an enabled interrupt). Also, $DBSR_{IDE}$ is set to indicate that the debug exception occurred while Debug interrupts were disabled by MSR_{DE}=0.

Later, if the debug exception has not been reset by clearing $DBSR_{DAC1R}$, $DBSR_{DAC1W}$, $DBSR_{DAC2R}$, $DBSR_{DAC2W}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

10.4.3 Trap Debug Event

A Trap debug event (TRAP) occurs if $DBCR0_{TRAP}$=1 (i.e. Trap debug events are enabled) and a ***Trap*** instruction (***tw***, ***twi***, ***td***, ***tdi***) is executed and the conditions specified by the instruction for the trap are met. The event can occur regardless of the setting of MSR_{DE} or $DBCR0_{IDM}$.

When a Trap debug event occurs, $DBSR_{TR}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Trap debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the excepting instruction.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Trap debug exception, a Debug interrupt will not occur, and a Trap exception type Program interrupt will occur instead if the trap condition is met.

Later, if the debug exception has not been reset by clearing $DBSR_{TR}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting both MSR_{DE} and $DBCR0_{IDM}$ to 1. Software in the debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

10.4.4 Branch Taken Debug Event

A Branch Taken debug event (BRT) occurs if $DBCR0_{BRT}$=1 (i.e. Branch Taken Debug events are enabled), execution is attempted of a branch instruction whose direction will be taken (that is, either an unconditional branch, or a conditional branch whose branch condition is met), and MSR_{DE}=1.

Branch Taken debug events are not recognized if MSR_{DE}=0 at the time of the execution of the branch instruction and thus $DBSR_{IDE}$ can not be set by a Branch Taken debug event. This is because branch instructions occur very frequently. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled via MSR_{DE} would result in an inordinate number of imprecise Debug interrupts.

When a Branch Taken debug event occurs, the $DBSR_{BRT}$ bit is set to 1 to record the debug exception and a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt). The execution of the instruction causing the exception will be suppressed, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the excepting instruction.

10.4.5 Instruction Complete Debug Event

An Instruction Complete debug event (ICMP) occurs if $DBCR0_{ICMP}$=1 (i.e. Instruction Complete debug events are enabled), execution of any instruction is completed, and MSR_{DE}=1. Note that if execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an Instruction Complete debug event. The ***sc*** instruction does not fall into the type of an instruction whose execution is suppressed, since the instruction actually completes execution and then generates a System Call interrupt. In this case, the Instruction Complete debug exception will also be set.

Instruction Complete debug events are not recognized if MSR_{DE}=0 at the time of the execution of the instruction, $DBSR_{IDE}$ can not be set by an ICMP debug event. This is because allowing the common event of Instruction Completion to be recorded as an exception in the DBSR while Debug interrupts are disabled via MSR_{DE} would mean that the Debug interrupt handler software would receive an inordinate number of imprecise Debug interrupts every time Debug interrupts were re-enabled via MSR_{DE}.

When an Instruction Complete debug event occurs, $DBSR_{ICMP}$ is set to 1 to record the debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the instruction after the one causing the Instruction Complete debug exception.

10.4.6 Interrupt Taken Debug Event

10.4.6.1 Causes of Interrupt Taken Debug Events

Only base class interrupts and guest class interrupts (Category: Embedded.Hypervisor) can cause an Interrupt Taken debug event. If the Embedded.Enhanced Debug category is not supported or is supported and not enabled, all other interrupts automatically clear MSR_{DE}, and thus would always prevent the associated Debug interrupt from occurring precisely. If the Embedded.Enhanced Debug category is supported and enabled, then critical class interrupts do not automatically clear MSR_{DE}, but they cause Critical Interrupt Taken debug events instead of Interrupt Taken debug events.

Also, if the Embedded.Enhanced Debug category is not supported or is supported and not enabled, Debug interrupts themselves are critical class interrupts, and thus any Debug interrupt (for any other debug event) would always end up setting the additional exception of $DBSR_{IRPT}$ upon entry to the Debug interrupt handler. At this point, the Debug interrupt handler would be unable to determine whether or not the Interrupt Taken debug event was related to the original debug event.

10.4.6.2 Interrupt Taken Debug Event Description

An Interrupt Taken debug event (IRPT) occurs if $DBCR0_{IRPT}$=1 (i.e. Interrupt Taken debug events are enabled) and a base class interrupt occurs. Interrupt Taken debug events can occur regardless of the setting of MSR_{DE}.

When an Interrupt Taken debug event occurs, $DBSR_{IRPT}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Interrupt Taken debug event, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and Critical Save/Restore Register 0/Debug Save/Restore Register 0 [Category: Embedded.Enhanced Debug] will be set to the address of the interrupt vector which caused the Interrupt Taken debug event. No instructions at the base interrupt handler will have been executed.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Interrupt Taken debug event, a Debug interrupt will not occur, and the handler for the interrupt which caused the Interrupt Taken debug event will be allowed to execute.

Later, if the debug exception has not been reset by clearing $DBSR_{IRPT}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe the $DBSR_{IDE}$ bit to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug.]

10.4.7 Return Debug Event

A Return debug event (RET) occurs if $DBCR0_{RET}$=1 and an attempt is made to execute an ***rfi*** (and also ***rfgi*** <E.HV>). Return debug events can occur regardless of the setting of MSR_{DE}.

When a Return debug event occurs, $DBSR_{RET}$ is set to 1 to record the debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 at the time of the Return Debug event, a Debug interrupt will occur immediately, and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the ***rfi***.

If MSR_{DE}=0 at the time of the Return Debug event, a Debug interrupt will not occur.

Later, if the Debug exception has not been reset by clearing $DBSR_{RET}$, and MSR_{DE} is set to 1, a delayed imprecise Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. An imprecise Debug interrupt can be caused by executing an ***rfi*** when $DBCR0_{RET}$=1 and MSR_{DE}=0, and the execution of that ***rfi*** happens to cause MSR_{DE} to be set to 1. Software in the Debug interrupt handler can observe the $DBSR_{IDE}$ bit to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

10.4.8 Unconditional Debug Event

An Unconditional debug event (UDE) occurs when the Unconditional Debug Event (UDE) signal is activated by the debug mechanism. The exact definition of the UDE signal and how it is activated is implementation-dependent. The Unconditional debug event is the only debug event which does not have a corresponding enable bit for the event in DBCR0 (hence the name of the event). The Unconditional debug event can occur regardless of the setting of MSR_{DE}.

When an Unconditional debug event occurs, the $DBSR_{UDE}$ bit is set to 1 to record the Debug exception. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 (i.e. Debug interrupts are enabled) at the time of the Unconditional Debug exception, a Debug interrupt will occur immediately (provided there exists no higher priority exception which is enabled to cause an interrupt), and CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will be set to the address of the instruction which would have executed next had the interrupt not occurred.

If MSR_{DE}=0 (i.e. Debug interrupts are disabled) at the time of the Unconditional Debug exception, a Debug interrupt will not occur.

Later, if the Unconditional Debug exception has not been reset by clearing $DBSR_{UDE}$, and MSR_{DE} is set to 1, a delayed Debug interrupt will occur. In this case, CSRR0/DSRR0 [Category: Embedded.Enhanced Debug] will contain the address of the instruction after the one which enabled the Debug interrupt by setting MSR_{DE} to 1. Software in the Debug interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in CSRR0/DSRR0 [Category: Embedded.Enhanced Debug].

10.4.9 Critical Interrupt Taken Debug Event [Category: Embedded.Enhanced Debug]

A Critical Interrupt Taken debug event (CIRPT) occurs if $DBCR0_{CIRPT}$ = 1 (i.e. Critical Interrupt Taken debug events are enabled) and a critical interrupt occurs. A critical interrupt is any interrupt that saves state in CSRR0 and CSRR1 when the interrupt is taken. Critical Interrupt Taken debug events can occur regardless of the setting of MSR_{DE}.

When a Critical Interrupt Taken debug event occurs, $DBSR_{CIRPT}$ is set to 1 to record the debug event. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE} = 1 (i.e. Debug Interrupts are enabled) at the time of the Critical Interrupt Taken debug event, a Debug Interrupt will occur immediately (provided there is no higher priority exception which is enabled to cause an interrupt), and DSRR0 will be set to the address of the first instruction of the critical interrupt handler. No instructions at the critical interrupt handler will have been executed.

If MSR_{DE} = 0 (i.e. Debug Interrupts are disabled) at the time of the Critical Interrupt Taken debug event, a Debug Interrupt will not occur, and the handler for the critical interrupt which caused the debug event will be allowed to execute normally. Later, if the debug exception has not been reset by clearing $DBSR_{CIRPT}$ and MSR_{DE} is set to 1, a delayed Debug Interrupt will occur. In this case DSRR0 will contain the address of the instruction after the one that set MSR_{DE} = 1. Software in the Debug Interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in DSRR0.

10.4.10 Critical Interrupt Return Debug Event [Category: Embedded.Enhanced Debug]

A Critical Interrupt Return debug event (CRET) occurs if $DBCR0_{CRET}$ = 1 (i.e. Critical Interrupt Return debug events are enabled) and an attempt is made to execute an ***rfci*** instruction. Critical Interrupt Return debug events can occur regardless of the setting of MSR_{DE}.

When a Critical Interrupt Return debug event occurs, $DBSR_{CRET}$ is set to 1 to record the debug event. If MSR_{DE}=0, $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE} = 1 (i.e. Debug Interrupts are enabled) at the time of the Critical Interrupt Return debug event, a Debug Interrupt will occur immediately (provided there is no higher priority exception which is enabled to cause an interrupt), and DSRR0 will be set to the address of the ***rfci*** instruction.

If MSR_{DE} = 0 (i.e. Debug Interrupts are disabled) at the time of the Critical Interrupt Return debug event, a Debug Interrupt will not occur. Later, if the debug exception has not been reset by clearing $DBSR_{CRET}$ and MSR_{DE} is set to 1, a delayed Debug Interrupt will occur. In this case DSRR0 will contain the address of the instruction after the one that set MSR_{DE} = 1. An imprecise Debug Interrupt can be caused by executing an ***rfci*** when $DBCR0_{CRET}$ = 1 and MSR_{DE} = 0, and the execution of the ***rfci*** happens to cause MSR_{DE} to be set to 1. Software in the Debug Interrupt handler can observe $DBSR_{IDE}$ to determine how to interpret the value in DSRR0.

10.5 Debug Registers

This section describes debug-related registers that are accessible to software. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code.

10.5.1 Debug Control Registers

Debug Control Register 0 (DBCR0), Debug Control Register 1 (DBCR1), and Debug Control Register 2 (DBCR2) are each 32-bit registers. Bits of DBCR0, DBCR1, and DBCR2 are numbered 32 (most-significant bit) to 63 (least-significant bit). DBCR0, DBCR1, and DBCR2 are used to enable debug events, reset the thread, control timer operation during debug events, and set the debug mode of the thread.

10.5.1.1 Debug Control Register 0 (DBCR0)

The contents of the DBCR0 can be read into bits 32:63 of register RT using *mfspr RT,DBCR0*, setting bits 0:31 of RT to 0. The contents of bits 32:63 of register RS can be written to the DBCR0 using *mtspr DBCR0,RS*. The bit definitions for DBCR0 are shown below.

Bit(s) Description

32 ***External Debug Mode*** (EDM)
The EDM bit is a read-only bit that reflects whether the thread is controlled by an external debug facility. When EDM is set, internal debug mode is suppressed and the taking of debug interrupts does not occur.

0 The thread is not in external debug mode.
1 The thread is in external debug mode.

Virtualized Implementation Note

In a virtualized implementation when EDM=1, the value of MSR_{DE} is not specified and is not modifiable.

33 ***Internal Debug Mode*** (IDM)

0 Debug interrupts are disabled.
1 If MSR_{DE}=1, then the occurrence of a debug event or the recording of an earlier debug event in the Debug Status Register when MSR_{DE}=0 or $DBCR0_{IDM}$=0 will cause a Debug interrupt.

Programming Note

Software must clear debug event status in the Debug Status Register in the Debug interrupt handler when a Debug interrupt is taken before re-enabling interrupts via MSR_{DE}. Otherwise, redundant Debug interrupts will be taken for the same debug event.

34:35 ***Reset*** (RST)

00 No action
01 Implementation-specific
10 Implementation-specific
11 Implementation-specific

Warning: Writing 0b01, 0b10, or 0b11 to these bits may cause a thread reset to occur.

36 ***Instruction Completion Debug Event*** (ICMP)

0 ICMP debug events are disabled
1 ICMP debug events are enabled

Note: Instruction Completion will not cause an ICMP debug event if MSR_{DE}=0.

37 ***Branch Taken Debug Event Enable*** (BRT)

0 BRT debug events are disabled
1 BRT debug events are enabled

Note: Taken branches will not cause a BRT debug event if MSR_{DE}=0.

38 ***Interrupt Taken Debug Event Enable*** (IRPT)

0 IRPT debug events are disabled
1 IRPT debug events are enabled

Note: Critical interrupts will not cause an IRPT Debug event even if MSR_{DE}=0. If the Embedded.Enhanced Debug category is supported, see Section 10.4.9.

39 ***Trap Debug Event Enable*** (TRAP)

0 TRAP debug events cannot occur
1 TRAP debug events can occur

40 ***Instruction Address Compare 1 Debug Event Enable*** (IAC1)

0 IAC1 debug events cannot occur
1 IAC1 debug events can occur

41 ***Instruction Address Compare 2 Debug Event Enable*** (IAC2)

0 IAC2 debug events cannot occur
1 IAC2 debug events can occur

42 ***Instruction Address Compare 3 Debug Event Enable*** (IAC3)

0 IAC3 debug events cannot occur
1 IAC3 debug events can occur

43 ***Instruction Address Compare 4 Debug Event Enable*** (IAC4)

0 IAC4 debug events cannot occur
1 IAC4 debug events can occur

44:45 ***Data Address Compare 1 Debug Event Enable*** (DAC1)

00 DAC1 debug events cannot occur
01 DAC1 debug events can occur only if a store-type data storage access
10 DAC1 debug events can occur only if a load-type data storage access
11 DAC1 debug events can occur on any data storage access

46:47 ***Data Address Compare 2 Debug Event Enable*** (DAC2)

00 DAC2 debug events cannot occur
01 DAC2 debug events can occur only if a store-type data storage access
10 DAC2 debug events can occur only if a load-type data storage access
11 DAC2 debug events can occur on any data storage access

48 ***Return Debug Event Enable*** (RET)

0 RET debug events cannot occur
1 RET debug events can occur

Note: Return From Critical Interrupt will not cause an RET debug event if MSR_{DE}=0. If the Embedded.Enhanced Debug category is supported, see Section 10.4.10

49:56 Reserved

57 ***Critical Interrupt Taken Debug Event*** (CIRPT) **[Category: Embedded.Enhanced Debug]**
A Critical Interrupt Taken Debug Event occurs when $DBCR0_{CIRPT}$ = 1 and a critical interrupt (any interrupt that uses the critical class, i.e. uses CSRR0 and CSRR1) occurs.

0 Critical interrupt taken debug events are disabled.
1 Critical interrupt taken debug events are enabled.

58 ***Critical Interrupt Return Debug Event*** (CRET) **[Category: Embedded.Enhanced Debug]**
A Critical Interrupt Return Debug Event occurs when $DBCR0_{CRET}$= 1 and a return from critical interrupt (an ***rfci*** instruction is executed) occurs.

0 Critical interrupt return debug events are disabled.
1 Critical interrupt return debug events are enabled.

59:62 Implementation-dependent

63 ***Freeze Timers on Debug Event*** (FT)

0 Enable clocking of timers and Time Base
1 Disable clocking of timers and Time Base if any DBSR bit is set (except MRR)

Virtualized Implementation Note

The FT bit may not be supported in virtualized implementations.

This register is hypervisor privileged.

This register is hypervisor privileged.

10.5.1.2 Debug Control Register 1 (DBCR1)

The contents of the DBCR1 can be read into bits 32:63 a register RT using ***mfspr*** *RT,DBCR1*, setting bits 0:31 of RT to 0. The contents of bits 32:63 of register RS can be written to the DBCR1 using ***mtspr*** *DBCR1,RS*. The bit definitions for DBCR1 are shown below.

Bit(s) Description

32:33 ***Instruction Address Compare 1 User/ Supervisor Mode***(IAC1US)

00 IAC1 debug events can occur
01 Reserved

10 IAC1 debug events can occur only if MSR_{PR}=0
11 IAC1 debug events can occur only if MSR_{PR}=1

34:35 ***Instruction Address Compare 1 Effective/ Real Mode*** (IAC1ER)

00 IAC1 debug events are based on effective addresses
01 IAC1 debug events are based on real addresses
10 IAC1 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC1 debug events are based on effective addresses and can occur only if MSR_{IS}=1

36:37 ***Instruction Address Compare 2 User/ Supervisor Mode*** (IAC2US)

00 IAC2 debug events can occur
01 Reserved
10 IAC2 debug events can occur only if MSR_{PR}=0
11 IAC2 debug events can occur only if MSR_{PR}=1

38:39 ***Instruction Address Compare 2 Effective/ Real Mode*** (IAC2ER)

00 IAC2 debug events are based on effective addresses
01 IAC2 debug events are based on real addresses
10 IAC2 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC2 debug events are based on effective addresses and can occur only if MSR_{IS}=1

40:41 ***Instruction Address Compare 1/2 Mode*** (IAC12M)

00 Exact address compare

IAC1 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC1.

IAC2 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC2.

01 Address bit match

IAC1 and IAC2 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 are equal to the contents of IAC1, also ANDed with the contents of IAC2.

If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

10 Inclusive address range compare

IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2.

If IAC1US≠IAC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

11 Exclusive address range compare

IAC1 and IAC2 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2.

If IAC1US≠AC2US or IAC1ER≠IAC2ER, results are boundedly undefined.

42:47 Reserved

48:49 ***Instruction Address Compare 3 User/ Supervisor Mode*** (IAC3US)

00 IAC3 debug events can occur
01 Reserved
10 IAC3 debug events can occur only if MSR_{PR}=0
11 IAC3 debug events can occur only if MSR_{PR}=1

50:51 ***Instruction Address Compare 3 Effective/ Real Mode*** (IAC3ER)

00 IAC3 debug events are based on effective addresses
01 IAC3 debug events are based on real addresses
10 IAC3 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC3 debug events are based on effective addresses and can occur only if MSR_{IS}=1

52:53 ***Instruction Address Compare 4 User/ Supervisor Mode*** (IAC4US)

00 IAC4 debug events can occur
01 Reserved
10 IAC4 debug events can occur only if MSR_{PR}=0
11 IAC4 debug events can occur only if MSR_{PR}=1

54:55 ***Instruction Address Compare 4 Effective/ Real Mode*** (IAC4ER)

00 IAC4 debug events are based on effective addresses
01 IAC4 debug events are based on real addresses
10 IAC4 debug events are based on effective addresses and can occur only if MSR_{IS}=0
11 IAC4 debug events are based on effective addresses and can occur only if MSR_{IS}=1

56:57 ***Instruction Address Compare 3/4 Mode*** (IAC34M)

00 Exact address compare

IAC3 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC3.

IAC4 debug events can occur only if the address of the instruction fetch is equal to the value specified in IAC4.

01 Address bit match

IAC3 and IAC4 debug events can occur only if the address of the data storage access, ANDed with the contents of IAC4 are equal to the contents of IAC3, also ANDed with the contents of IAC4.

If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

10 Inclusive address range compare

IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4.

If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

11 Exclusive address range compare

IAC3 and IAC4 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC3 or is greater than or equal to the value specified in IAC4.

If IAC3US≠IAC4US or IAC3ER≠IAC4ER, results are boundedly undefined.

58:63 Reserved

This register is hypervisor privileged.

Architecture Note

For IAC12M inclusive (10) and exclusive (11) range comparisons, either $DBCR0_{IAC1}$ or $DBCR0_{IAC2}$ can be set to one in order to enable the range comparison. It is permissible for both $DBCR0_{IAC1}$ and $DBCR0_{IAC2}$ to be enabled. In that case, both $DBSR_{IAC1}$ and $DBSR_{IAC2}$ will be set for a range match.

The same behavior holds for IAC34M with the appropriate substitutions.

10.5.1.3 Debug Control Register 2 (DBCR2)

The contents of the DBCR2 can be copied into bits 32:63 register RT using ***mfspr*** *RT,DBCR2*, setting bits 0:31 of register RT to 0. The contents of bits 32:63 of a register RS can be written to the DBCR2 using ***mtspr*** *DBCR2,RS*. The bit definitions for DBCR2 are shown below.

Bit(s) Description

32:33 ***Data Address Compare 1 User/Supervisor Mode*** (DAC1US)

00 DAC1 debug events can occur
01 Reserved
10 DAC1 debug events can occur only if MSR_{PR}=0
11 DAC1 debug events can occur only if MSR_{PR}=1

34:35 ***Data Address Compare 1 Effective/Real Mode*** (DAC1ER)

00 DAC1 debug events are based on effective addresses
01 DAC1 debug events are based on real addresses
10 DAC1 debug events are based on effective addresses and can occur only if MSR_{DS}=0
11 DAC1 debug events are based on effective addresses and can occur only if MSR_{DS}=1

36:37 ***Data Address Compare 2 User/Supervisor Mode*** (DAC2US)

00 DAC2 debug events can occur
01 Reserved
10 DAC2 debug events can occur only if MSRPR=0
11 DAC2 debug events can occur only if MSRPR=1

38:39 ***Data Address Compare 2 Effective/Real Mode*** (DAC2ER)

00 DAC2 debug events are based on effective addresses
01 DAC2 debug events are based on real addresses
10 DAC2 debug events are based on effective addresses and can occur only if MSR_{DS}=0
11 DAC2 debug events are based on effective addresses and can occur only if MSR_{DS}=1

40:41 ***Data Address Compare 1/2 Mode*** (DAC12M)

00 Exact address compare

DAC1 debug events can occur only if the address of the data storage access is equal to the value specified in DAC1.

DAC2 debug events can occur only if the address of the data storage access is equal to the value specified in DAC2.

01 Address bit match

DAC1 and DAC2 debug events can occur only if the address of the data storage access, ANDed with the contents of DAC2 are equal to the contents of DAC1, also ANDed with the contents of DAC2.

If DAC1US≠DAC2US or DAC1ER≠DAC2ER, results are boundedly undefined.

10 Inclusive address range compare

DAC1 and DAC2 debug events can occur only if the address of the data storage access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2.

If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined.

11 Exclusive address range compare

DAC1 and DAC2 debug events can occur only if the address of the data storage access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2.

If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined.

42:63 Reserved

Architecture Note

Bits 44:63 will be among the last to be assigned a meaning as these bits contained the DVC (Data Value Compare) function in earlier versions of the architecture.

This register is hypervisor privileged.

Architecture Note

For DAC12M inclusive (10) and exclusive (11) range comparisons, either $DBCR0_{DAC1}$ or $DBCR0_{DAC2}$ can be set to one in order to enable the range comparison. It is permissible for both $DBCR0_{DAC1}$ and $DBCR0_{DAC2}$ to be enabled. In that case, both $DBSR_{DAC1}$ and $DBSR_{DAC2}$ will be set for a range match.

The same behavior holds for DAC34M with the appropriate substitutions.

10.5.2 Debug Status Register

The Debug Status Register (DBSR) is a 32-bit register and contains status on debug events and the most recent thread reset.

The DBSR is set via hardware, and read and cleared via software. The contents of the DBSR can be read into bits 32:63 of a register RT using the ***mfspr*** instruction, setting bits 0:31 of RT to zero. Bits in the DBSR can be cleared using the ***mtspr*** instruction. Clearing is done by writing bits 32:63 of a register to the DBSR with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write-data to the DBSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

The bit definitions for the DBSR are shown below:

Bit(s) Description

32 ***Imprecise Debug Event*** (IDE)

Set to 1 if MSR_{DE}=0 and a debug event causes its respective Debug Status Register bit to be set to 1.

33 ***Unconditional Debug Event*** (UDE)

Set to 1 if an Unconditional debug event occurred. See Section 10.4.8.

34:35 ***Most Recent Reset*** (MRR)

Set to one of three values when a reset occurs. These two bits are undefined at power-up.

00 No reset occurred since these bits last cleared by software
01 Implementation-dependent reset information
10 Implementation-dependent reset information
11 Implementation-dependent reset information

36 ***Instruction Complete Debug Event*** (ICMP)

Set to 1 if an Instruction Completion debug event occurred and $DBCR0_{ICMP}$=1. See Section 10.4.5.

37 ***Branch Taken Debug Event*** (BRT)

Set to 1 if a Branch Taken debug event occurred and $DBCR0_{BRT}$=1. See Section 10.4.4.

38 ***Interrupt Taken Debug Event*** (IRPT)

Set to 1 if an Interrupt Taken debug event occurred and $DBCR0_{IRPT}$=1. See Section 10.4.6.

39 ***Trap Instruction Debug Event*** (TRAP)

Set to 1 if a Trap Instruction debug event occurred and $DBCR0_{TRAP}$=1. See Section 10.4.3.

40 ***Instruction Address Compare 1 Debug Event*** (IAC1)

Set to 1 if an IAC1 debug event occurred and $DBCR0_{IAC1}$=1. See Section 10.4.1.

41 ***Instruction Address Compare 2 Debug Event*** (IAC2)

Set to 1 if an IAC2 debug event occurred and $DBCR0_{IAC2}$=1. See Section 10.4.1.

42 ***Instruction Address Compare 3 Debug Event*** (IAC3)

Set to 1 if an IAC3 debug event occurred and $DBCR0_{IAC3}$=1. See Section 10.4.1.

43 ***Instruction Address Compare 4 Debug Event*** (IAC4)

Set to 1 if an IAC4 debug event occurred and $DBCR0_{IAC4}$=1. See Section 10.4.1.

44 ***Data Address Compare 1 Read Debug Event*** (DAC1R)

Set to 1 if a read-type DAC1 debug event occurred and $DBCR0_{DAC1}$=0b10 or $DBCR0_{DAC1}$=0b11. See Section 10.4.2.

45 ***Data Address Compare 1 Write Debug Event*** (DAC1W)

Set to 1 if a write-type DAC1 debug event occurred and $DBCR0_{DAC1}$=0b01 or $DBCR0_{DAC1}$=0b11. See Section 10.4.2.

46 ***Data Address Compare 2 Read Debug Event*** (DAC2R)

Set to 1 if a read-type DAC2 debug event occurred and $DBCR0_{DAC2}$=0b10 or $DBCR0_{DAC2}$=0b11. See Section 10.4.2.

47 ***Data Address Compare 2 Write Debug Event*** (DAC2W)

Set to 1 if a write-type DAC2 debug event occurred and $DBCR0_{DAC2}$=0b01 or $DBCR0_{DAC2}$=0b11. See Section 10.4.2.

48 ***Return Debug Event*** (RET)

Set to 1 if a Return debug event occurred and $DBCR0_{RET}$=1. See Section 10.4.2.

49:52 Reserved

53:56 Implementation-dependent

57 ***Critical Interrupt Taken Debug Event*** (CIRPT) **[Category: Embedded.Enhanced Debug]**
A Critical Interrupt Taken Debug Event occurs when $DBCR0_{CIRPT}$=1 and a critical interrupt (any interrupt that uses the critical class, i.e. uses CSRR0 and CSRR1) occurs.

0 Critical interrupt taken debug events are disabled.
1 Critical interrupt taken debug events are enabled.

58 ***Critical Interrupt Return Debug Event*** (CRET) **[Category: Embedded.Enhanced Debug]**
A Critical Interrupt Return Debug Event occurs when $DBCR0_{CRET}$=1 and a return from critical interrupt (an ***rfci*** instruction is executed) occurs.

0 Critical interrupt return debug events are disabled.
1 Critical interrupt return debug events are enabled.

59:63 Implementation-dependent

This register is hypervisor privileged.

10.5.3 Debug Status Register Write Register (DBSRWR)

The Debug Status Register Write Register (DBSRWR) allows a hypervisor state program to write the contents of the Debug Status Register (see Section 10.5.2, “Debug Status Register”). The format of the DBSRWR is shown in Figure 86 below.

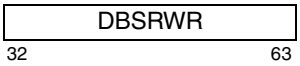


Figure 86. Debug Status Register Write Register

The DBSRWR is provided as a means to restore the contents of the DBSR on a partition switch.

The DBSRWR is hypervisor privileged.

Writing DBSRWR changes the value in the DBSR. Writing non-zero bits may enable an imprecise Debug exception which may cause later imprecise Debug Interrupts. In order to correctly write DBSRWR, software should ensure that MSR_{DE} = 0 when the value is written and perform a context synchronizing operation before setting MSR_{DE} to 1.

10.5.4 Instruction Address Compare Registers

The Instruction Address Compare Register 1, 2, 3, and 4 (IAC1, IAC2, IAC3, and IAC4 respectively) are each 64-bits, with bit 63 being reserved.

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in either IAC1, IAC2, IAC3, or IAC4, inside or outside a range specified by IAC1 and IAC2 or, inside or outside a range specified by IAC3 and IAC4, or to blocks of addresses specified by the combination of the IAC1 and IAC2, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address (see Section 10.4.1 on page 1215).

The contents of the Instruction Address Compare *i* Register (where *i*={1,2,3, or 4}) can be read into register RT using *mfspr RT,IACi*. The contents of register RS can be written to the Instruction Address Compare *i* Register using *mtspr IACi,RS*.

This register is hypervisor privileged.

10.5.5 Data Address Compare Registers

The Data Address Compare Register 1 and 2 (DAC1 and DAC2 respectively) are each 64-bits.

A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either the DAC1 or DAC2, inside or outside a range specified by the DAC1 and DAC2, or to blocks of addresses specified by the combination of the DAC1 and DAC1 (see Section 10.4.2).

The contents of the Data Address Compare *i* Register (where *i*={1 or 2}) can be read into register RT using *mfspr RT,DACi*. The contents of register RS can be written to the Data Address Compare *i* Register using *mtspr DACi,RS*.

The contents of the DAC1 or DAC2 are compared to the address generated by a data storage access instruction.

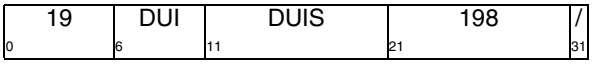
These registers are hypervisor privileged.

10.6 Debugger Notify Halt Instruction

The ***dnh*** instruction provides the means for the transfer of information between the thread and an implementation-dependent external debug facility. ***dnh*** also causes the thread to stop fetching and executing instructions.

Debugger Notify Halt *XFX-form*

dnh DUI,DUIS




```
if enabled by implementation-dependent means
then
   implementation-dependent register ← DUI
   halt thread
else
   illegal instruction exception
```

Execution of the ***dnh*** instruction causes the thread to stop fetching instructions and taking interrupts if execution of the instruction has been enabled. The contents of the DUI field are sent to the external debug facility to identify the reason for the halt.

If execution of the ***dnh*** instruction has not been previously enabled, executing the ***dnh*** instruction produces an Illegal Instruction exception. The means by which execution of the ***dnh*** instruction is enabled is implementation-dependent.

The current state of the debug facility, whether the thread is in IDM or EDM mode has no effect on the execution of the ***dnh*** instruction.

The instruction is context synchronizing.

Programming Note

The DUIS field in the instruction may be used to pass information to an external debug facility. After the ***dnh*** instruction has executed, the instruction itself can be read back by the Illegal Instruction Interrupt handler or the external debug facility if the contents of the DUIS field are of interest. If the thread entered the Illegal Instruction Interrupt handler, software can use SRR0 to obtain the address of the ***dnh*** instruction which caused the handler to be invoked.

Special Registers Altered:

None

Chapter 11. Processor Control [Category: Embedded.Processor Control]

11.1 Overview

The Processor Control facility provides a mechanism for threads within a coherence domain to send messages to all devices in the coherence domain. The facility provides a mechanism for sending interrupts that are not dependent on the interrupt controller to threads and allows message filtering by the threads that receive the message.

The Processor Control facility is also useful for sending messages to a device that provides specialized services such as secure boot operations controlled by a security device.

The Processor Control facility defines how threads send messages and what actions threads take on the receipt of a message. The actions taken by devices other than threads are not defined.

Programming Note

A common use of ***msgsnd*** is to deliver an external interrupt to a partition which has set MSR_{EE}=0. A guest doorbell message will interrupt to the hypervisor when MSR_{EE}=1 and MSR_{GS}=1. The hypervisor can then deliver the external interrupt to the partition. For Servers, a similar set of operations can also be performed using the Book III-S ***msgsnd*** instruction. These operations achieve a result analogous to the mediated external interrupt in Book III-S.

11.2 Programming Model

Threads initiate a message by executing the ***msgsnd*** instruction and specifying a message type and message payload in a general purpose register. Sending a message causes the message to be sent to all the devices, including the sending thread, in the coherence domain in a reliable manner.

Each device receives all messages that are sent. The actions that a device takes are dependent on the message type and payload. There are no restrictions on what messages a thread can send.

To provide inter thread interrupt capability the following *doorbell* message types are defined:

- *Processor Doorbell*
- *Processor Doorbell Critical*
- *Guest Processor Doorbell* <E.HV>
- *Guest Processor Doorbell Critical* <E.HV>
- *Guest Processor Doorbell Machine Check* <E.HV>

A doorbell message causes an interrupt to occur on threads when the message is received and the thread determines through examination of the payload that the message should be accepted. The examination of the payload for this purpose is termed *filtering*. The acceptance of a doorbell message causes an exception to be generated on the accepting thread.

Threads accept and filter messages defined in Section 11.2.1. Threads may also accept other implementation-dependent defined messages.

11.2.1 Message Handling and Filtering

Threads filter, accept, and handle message types defined as follows. The message type is specified in the message and is determined by the contents of register $RB_{32:36}$ used as the operand in the ***msgsnd*** instruction.The message type is interpreted as follows:

Value	Description
0	***Doorbell Interrupt*** (DBELL) A Processor Doorbell exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Processor Doorbell Interrupt occurs when no higher priority exception exists, a Processor Doorbell exception exists, and the interrupt is enabled (MSR_{EE}=1). If Category: Embedded.Hypervisor is supported, the interrupt is enabled if (MSR_{EE}=1 or MSR_{GS}=1).

1 ***Doorbell Critical Interrupt*** (DBELL_CRIT)
A Processor Doorbell Critical exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Processor Doorbell Critical exception exists, and the interrupt is enabled (MSR_{CE}=1). If Category: Embedded.Hypervisor is supported, the interrupt is enabled if (MSR_{CE}=1 or MSR_{GS}=1).

2 ***Guest Doorbell Interrupt*** (G_DBELL) <E.HV>
A Guest Processor Doorbell exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Guest Processor Doorbell Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell exception exists, and the interrupt is enabled (MSR_{EE}=1 and MSR_{GS}=1).

3 ***Guest Doorbell Interrupt Critical*** (G_DBELL_CRIT) <E.HV>
A Guest Processor Doorbell Critical exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Guest Processor Doorbell Critical Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Critical exception exists, and the interrupt is enabled (MSR_{CE}=1 and MSR_{GS}=1).

4 ***Guest Doorbell Interrupt Machine Check*** (G_DBELL_MC) <E.HV>
A Guest Processor Doorbell Machine Check exception is generated on the thread when the thread has filtered the message based on the payload and has determined that it should accept the message. A Guest Processor Doorbell Machine Check Interrupt occurs when no higher priority exception exists, a Guest Processor Doorbell Machine Check exception exists, and the interrupt is enabled (MSR_{ME}=1 and MSR_{GS}=1).

Message types other than these and their associated actions are implementation-dependent.

11.2.2 Doorbell Message Filtering

A thread receiving a DBELL message will filter the message and either ignore the message or accept the message and generate a Processor Doorbell exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the ***msgsnd*** instruction. The payload bits are defined below.

Bit **Description**

37 ***Broadcast*** (BRDCAST)
If set, the message is accepted by all threads regardless of the value of the PIR register and the value of PIRTAG.

- 0 If the value of PIR and PIRTAG are equal a Processor Doorbell exception is generated.
- 1 A Processor Doorbell exception is generated regardless of the value of PIRTAG and PIR.

38:49 ***LPID Tag*** (LPIDTAG) <E.HV>
The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.

50:63 ***PIR Tag*** (PIRTAG)
The contents of this field are compared with bits 50:63 of the PIR register.

If category E.HV is supported by the thread on which a DBELL message is received, the message will only be accepted if it is for this partition ($payload_{LPIDTAG}$ = LPIDR) or it is for all partitions ($payload_{LPIDTAG}$ = 0) and it meets the additional criteria for acceptance below.

If a DBELL message is received by a thread, the message is accepted and a Processor Doorbell exception is generated if one of the following conditions exist:

- This is a broadcast message ($payload_{BRDCAST}$=1);
- The message is intended for this thread ($PIR_{50:63}$=$payload_{PIRTAG}$).

The exception condition remains until a Processor Doorbell Interrupt is taken, or a ***msgclr*** instruction is executed on the receiving thread with a message type of DBELL. A change to any of the filtering criteria (i.e. changing the PIR register) will not clear a pending Processor Doorbell exception.

DBELL messages are not cumulative. That is, if a DBELL message is accepted and the interrupt is pended because MSR_{EE}=0, additional DBELL messages that would be accepted are ignored until the Processor Doorbell exception is cleared by taking the interrupt or cleared by executing a ***msgclr*** with a message type of DBELL on the receiving thread.

The temporal relationship between when a DBELL message is sent and when it is received in a given thread is not defined.

11.2.2.1 Doorbell Critical Message Filtering

A thread receiving a DBELL_CRIT message type will filter the message and either ignore the message or

accept the message and generate a Processor Doorbell Critical exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the ***msgsnd*** instruction. The payload bits are defined below.

Bit **Description**

37 ***Broadcast*** (BRDCAST)
If set, the message is accepted by all threads regardless of the value of the PIR register and the value of PIRTAG.

0 If the value of PIR and PIRTAG are equal a Processor Doorbell Critical exception is generated.
1 A Processor Doorbell Critical exception is generated regardless of the value of PIRTAG and PIR.

38:49 ***LPID Tag*** (LPIDTAG) <E.HV>
The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.

50:63 ***PIR Tag*** (PIRTAG)
The contents of this field are compared with bits 50:63 of the PIR register.

If category E.HV is supported by the thread on which a DBELL_CRIT message is received, the message will only be accepted if it is for this partition ($payload_{LPIDTAG}$ = LPIDR) or it is for all partitions ($payload_{LPIDTAG}$ = 0) and it meets the additional criteria for acceptance below.

If a DBELL_CRIT message is received by a thread, the message is accepted and a Processor Doorbell Critical exception is generated if one of the following conditions exist:

- This is a broadcast message ($payload_{BRDCAST}$=1);
- The message is intended for this thread ($PIR_{50:63}$=$payload_{PIRTAG}$).

DBELL_CRIT messages are not cumulative. That is, if a DBELL_CRIT message is accepted and the interrupt is pended because MSR_{CE}=0, additional DBELL_CRIT messages that would be accepted are ignored until the Processor Doorbell Critical exception is cleared by taking the interrupt or cleared by executing a ***msgclr*** with a message type of DBELL_CRIT on the receiving thread.

The temporal relationship between when a DBELL_CRIT message is sent and when it is received in a given thread is not defined.

The temporal relationship between when a DBELL_CRIT message is sent and when it is received in a given thread is not defined.

11.2.2.2 Guest Doorbell Message Filtering [Category: Embedded.Hypervisor]

A thread receiving a G_DBELL message type will filter the message and either ignore the message or accept the message and generate a Guest Processor Doorbell Critical exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the ***msgsnd*** instruction. The payload bits are defined below.

Bit **Description**

37 ***Broadcast*** (BRDCAST)
If set, the message is accepted by all threads regardless of the value of the GPIR register and the value of PIRTAG.

0 If the value of GPIR and PIRTAG are equal a Guest Processor Doorbell exception is generated.
1 A Guest Processor Doorbell exception is generated regardless of the value of PIRTAG and GPIR.

38:49 ***LPID Tag*** (LPIDTAG)
The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.

50:63 ***PIR Tag*** (PIRTAG)
The contents of this field are compared with bits 50:63 of the GPIR register.

When a G_DBELL message is received by a thread, the message will only be accepted if it is for this partition ($payload_{LPIDTAG}$ = LPIDR) or it is for all partitions ($payload_{LPIDTAG}$ = 0) and it meets the additional criteria for acceptance below.

The message is accepted and a Guest Processor Doorbell exception is generated if one of the following conditions exist:

- This is a broadcast message ($payload_{BRDCAST}$=1);
- The message is intended for this thread ($GPIR_{50:63}$=$payload_{PIRTAG}$).

G_DBELL messages are not cumulative. That is, if a G_DBELL message is accepted and the interrupt is pended because MSR_{CE}=0, additional G_DBELL messages that would be accepted are ignored until the Guest Processor Doorbell exception is cleared by taking the interrupt or cleared by executing a ***msgclr*** with a message type of G_DBELL on the receiving thread.

The temporal relationship between when a G_DBELL message is sent and when it is received in a given thread is not defined.

11.2.2.3 Guest Doorbell Critical Message Filtering [Category: Embedded.Hypervisor]

A thread receiving a G_DBELL_CRIT message type will filter the message and either ignore the message or accept the message and generate a Guest Processor Doorbell Critical exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the ***msgsnd*** instruction. The payload bits are defined below.

Bit	Description
37	***Broadcast*** (BRDCAST) If set, the message is accepted by all threads regardless of the value of the GPIR register and the value of PIRTAG. 0 If the value of GPIR and PIRTAG are equal a Guest Processor Doorbell Critical exception is generated. 1 A Guest Processor Doorbell Critical exception is generated regardless of the value of PIRTAG and GPIR.
38:49	***LPID Tag*** (LPIDTAG) The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.
50:63	***PIR Tag*** (PIRTAG) The contents of this field are compared with bits 50:63 of the GPIR register.

When a G_DBELL_CRIT message is received by a thread, the message will only be accepted if it is for this partition ($payload_{LPIDTAG}$ = LPIDR) or it is for all partitions ($payload_{LPIDTAG}$ = 0) and it meets the additional criteria for acceptance below.

If a G_DBELL_CRIT message is received by a thread, the message is accepted and a Guest Processor Doorbell Critical exception is generated if one of the following conditions exist:

- This is a broadcast message ($payload_{BRDCAST}$=1);
- The message is intended for this thread ($GPIR_{50:63}$=$payload_{PIRTAG}$).

G_DBELL_CRIT messages are not cumulative. That is, if a G_DBELL_CRIT message is accepted and the interrupt is pended because MSR_{CE}=0, additional G_DBELL messages that would be accepted are ignored until the Guest Processor Doorbell Critical exception is cleared by taking the interrupt or cleared by executing a ***msgclr*** with a message type of G_DBELL_CRIT on the receiving thread.

The temporal relationship between when a G_DBELL_CRIT message is sent and when it is received in a given thread is not defined.

11.2.2.4 Guest Doorbell Machine Check Message Filtering [Category: Embedded.Hypervisor]

A thread receiving a G_DBELL_MC message type will filter the message and either ignore the message or accept the message and generate a Guest Processor Doorbell Machine Check exception based on the payload and the state of the thread at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the ***msgsnd*** instruction. The payload bits are defined below.

Bit	Description
37	***Broadcast*** (BRDCAST) If set, the message is accepted by all threads regardless of the value of the GPIR register and the value of PIRTAG. 0 If the value of GPIR and PIRTAG are equal a Guest Processor Doorbell Machine Check exception is generated. 1 A Guest Processor Doorbell Machine Check exception is generated regardless of the value of PIRTAG and GPIR.
38:49	***LPID Tag*** (LPIDTAG) The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.
50:63	***PIR Tag*** (PIRTAG) The contents of this field are compared with bits 50:63 of the GPIR register.

When a G_DBELL_MC message is received by a thread, the message will only be accepted if it is for this partition ($payload_{LPIDTAG}$ = LPIDR) or it is for all partitions ($payload_{LPIDTAG}$ = 0) and it meets the additional criteria for acceptance below.

If a G_DBELL_MC message is received by a thread, the message is accepted and a Guest Processor Doorbell Machine Check exception is generated if one of the following conditions exist:

- This is a broadcast message ($payload_{BRDCAST}$=1);
- The message is intended for this thread ($GPIR_{50:63}$=$payload_{PIRTAG}$).

G_DBELL_MC messages are not cumulative. That is, if a G_DBELL_MC message is accepted and the interrupt is pended because MSR_{CE}=0, additional G_DBELL_MC messages that would be accepted are ignored until the Guest Processor Doorbell Machine Check exception is cleared by taking the interrupt or cleared by executing a ***msgclr*** with a message type of G_DBELL_MC on the receiving thread.

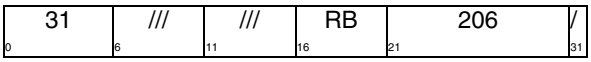
The temporal relationship between when a G_DBELL_MC message is sent and when it is received in a given thread is not defined.

11.3 Processor Control Instructions

msgsnd and ***msgclr*** instructions are provided for sending and clearing messages to threads and other devices in the coherence domain. These instructions are hypervisor privileged.

Message Send *X-form*

msgsnd RB




```
msgtype ← GPR(RB)32:36
payload ← GPR(RB)37:63
send_msg_to_choherence_domain(msgtype, payload)
```

msgsnd sends a message to all devices in the coherence domain. The message contains a type and a payload. The message type (*msgtype*) is defined by the contents of $RB_{32:36}$ and the message payload is defined by the contents of $RB_{37:63}$. Message delivery is reliable and guaranteed. Each device may perform specific actions based on the message type and payload or may ignore messages. Consult the implementation user's manual for specific actions taken based on message type and payload.

For threads, actions taken on receipt of a message are defined in Section 11.2.1.

This instruction is hypervisor privileged.

Special Registers Altered:

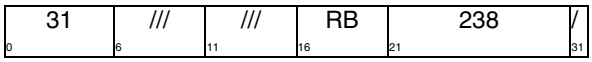
None

Programming Note

if ***msgsnd*** is used to send notify the receiver that updates have been made to storage, a ***sync*** should be placed between the stores and the ***msgsnd***. See Section 6.11.3, "Synchronize Instruction" on page 1124.

Message Clear *X-form*

msgclr RB




```
msgtype ← GPR(RB)32:36
clear_received_message(msgtype)
```

msgclr clears a message of *msgtype* previously accepted by the thread executing the ***msgclr***. *msgtype* is defined by the contents of $RB_{32:36}$. A message is said to be cleared when a pending exception generated by an accepted message has not yet taken its associated interrupt.

A context synchronizing instruction or event that is executed or occurs subsequent to the execution of ***msgclr*** ensures that the formerly pending exception will not result in an interrupt when the corresponding interrupt class is reenabled.

For threads, the types of messages that can be cleared are defined in Section 11.2.1.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Programming Note

Execution of a ***msgclr*** instruction that clears a pending exception when the associated interrupt is masked because the interrupt enable (MSR_{EE} or MSR_{CE}) is set to 0 will always clear the pending exception (and thus the interrupt will not occur) only if the instruction that sets MSR_{EE} or MSR_{CE} to 1 is context synchronizing or a context synchronizating operation occurs subsequent to the instruction but before MSR_{EE} or MSR_{CE} is set to 1.

Chapter 12. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers, the contents of TLB entries, or the contents of other system resources that control the context in which a program executes can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed and data accesses are performed. For example, changing certain bits in the MSR has the side effect of changing how instruction addresses are calculated. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed or data accesses are performed, is called a *context-altering instruction*. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 13 (for data access) and Table 12 (for instruction fetch and execution).

The notation "CSI" in the tables means any context synchronizing instruction (e.g., ***sc***, ***isync***, ***rfi***, ***rfci***, ***rfmci***, or ***rfdi*** [Category: Embedded. Enhanced Debug]). A context synchronizing interrupt (i.e., any interrupt except non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like "the synchronizing instruction", below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, "the synchronizing instruction before (after) the context-altering instruction" should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Programming Note

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as an ***rfi***, ***rfgi*** [Category: Embedded.Hypervisor], ***rfci, rfmci, or rfdi*** [Category: Embeddd.Enhanced Debug] that returns from an interrupt handler, provide the required synchronization.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing (e.g., ***rfi***, etc.) or when altering the MSR in most cases (see the tables). No software synchronization is required before most of the other alterations shown in Table 12, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the hardware must determine whether any of these preceding instructions are context synchronizing).

Unless otherwise stated, the material in this chapter assumes a single-threaded environment.

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfi	none	none	
rfci	none	none	
rfmci	none	none	
rfdi[Category:E.ED]	none	none	
rfgi	none	none	
sc	none	none	
mtmsr (GS)	none	CSI	
mtspr (LPIDR)	none	CSI	2
mtspr (GIVPR)	none	none	
mtspr (DBSRWR)	none	CSI	
mtspr (EPCR)	none	CSI	
mtspr (GIVORi)	none	none	
mtmsr (CM)	none	none	
mtmsr (UCLE)	none	none	
mtmsr (SPV)	none	CSI	
mtmsr (CE)	none	none	4
mtmsr (EE)	none	none	4
mtmsr (PR)	none	CSI	
mtmsr (FP)	none	CSI	
mtmsr (DE)	none	CSI	
mtmsr (ME)	none	CSI	3
mtmsr (FE0)	none	CSI	
mtmsr (FE1)	none	CSI	
mtmsr (IS)	none	CSI	2
mtspr (DEC)	none	none	7
mtspr (GDEC)	none	none	7
mtspr (PID)	none	CSI	2
mtspr (IVPR)	none	none	
mtspr (DBSR)	--	--	5
mtspr (DBCR0,DBCR1)	--	--	5
mtspr (IAC1,IAC2,IAC3, IAC4)	--	--	5
mtspr (IVORi)	none	none	
mtspr (TSR)	none	none	7
mtspr (GTSR)	none	none	7
mtspr (GTSRWR)	none	CSI	7
mtspr (TCR)	none	none	7
mtspr (GTCR)	none	none	7
mtspr (MMUCSR0) TLB invalidate all	none	CSI, or CSI and ***sync***	6,9
mtspr (MCIVPR)	none	none	
tlbilx	none	CSI, or CSI and ***sync***	6
Store(PTE)	none	{***sync***, CSI}	1,8
tlbivax	none	CSI, or CSI and ***sync***	1,6
tlbwe	none	CSI, or CSI and ***sync***	1,6
wrtee	none	none	4
wrteei	none	none	4

Table 12:Synchronization requirements for instruction fetch and/or execution

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfi	none	none	
rfci	none	none	
rfmci	none	none	
rfdi[Category:E.ED]	none	none	
rfgi	none	none	
sc	none	none	
mtmsr (GS)	none	CSI	
mtspr (LPIDR)	CSI	CSI	
mtmsr (CM)	none	CSI	
mtmsr (PR)	none	CSI	
mtmsr (ME)	none	CSI	3
mtmsr (DS)	none	CSI	
mtspr (PID)	CSI	CSI	
mtspr (DBSR)	--	--	5
mtspr (DBCR0,DBCR2)	---	---	5
mtspr (DAC1,DAC2)	--	--	5
mtspr (MMUCSR0) TLB invalidate all	CSI	CSI, or CSI and ***sync***	6,9
tlbilx	CSI	CSI, or CSI and ***sync***	6
Store(PTE)	none	{***sync***, CSI}	1,8
tlbivax	CSI	CSI, or CSI and ***sync***	1,6
tlbwe	CSI	CSI, or CSI and ***sync***	1,6

Table 13:Synchronization requirements for data access

Notes:

1. There are additional software synchronization requirements for this instruction in multi-threaded environments (e.g., it may be necessary to invalidate one or more TLB entries on all threads in the system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect); it is also necessary to execute a ***tlbsync*** instruction.
2. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be

independent of whether the alteration has taken effect.

3. A context synchronizing instruction is required after altering MSR_{ME} to ensure that the alteration takes effect for subsequent Machine Check interrupts, which may not be recoverable and therefore may not be context synchronizing.

4. The effect of changing MSR_{EE} or MSR_{CE} is immediate.

   If an ***mtmsr***, ***wrtee***, or ***wrteei*** instruction sets MSR_{EE} to '0', an External Input, DEC or FIT interrupt does not occur after the instruction is executed.

   If an ***mtmsr***, ***wrtee***, or ***wrteei*** instruction changes MSR_{EE} from '0' to '1' when an External Input, Decrementer, Fixed-Interval Timer, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***, ***wrtee***, or ***wrteei*** is executed, and before the next instruction is executed in the program that set MSR_{EE} to '1'.

   If an ***mtmsr*** instruction sets MSR_{CE} to '0', a Critical Input or Watchdog Timer interrupt does not occur after the instruction is executed.

   If an ***mtmsr*** instruction changes MSR_{CE} from '0' to '1' when a Critical Input, Watchdog Timer or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the ***mtmsr*** is executed, and before the next instruction is executed in the program that set MSR_{CE} to '1'.

5. Synchronization requirements for changing any of the Debug Facility Registers are implementation-dependent.

6. For data accesses, the context synchronizing instruction before the ***tlbwe, tlbilx***, or ***tlbivax*** instruction ensures that all storage accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the ***tlbwe***, ***tlbilx***, or ***tlbivax*** ensures that subsequent storage accesses (data and instruction) will use the updated value in the TLB entry(s) being affected. It does not ensure that all storage accesses previously translated by the TLB entry(s) being updated have completed with respect to storage; if these completions must be ensured, the ***tlbwe***, ***tlbilx***, or ***tlbivax*** must be followed by a ***sync*** instruction as well as by a context synchronizing instruction.

Programming Note

The following sequence illustrates why it is necessary, for data accesses, to ensure that all storage accesses due to instructions before the ***tlbwe*** or ***tlbivax*** have completed to a point at which they have reported all exceptions they will cause. Assume that valid TLB entries exist for the target storage location when the sequence starts.

- A program issues a load or store to a page.
- The same program executes a ***tlbwe*** or ***tlbivax*** that invalidates the corresponding TLB entry.
- The *Load* or *Store* instruction finally executes, and gets a TLB Miss exception.
- The TLB Miss exception is semantically incorrect. In order to prevent it, a context synchronizing instruction must be executed between steps 1 and 2.

7. The elapsed time between the Decrementer reaching zero, or the transition of the selected Time Base bit for the Fixed-Interval Timer or the Watchdog Timer, and the signalling of the Decrementer, Fixed-Interval Timer or the Watchdog Timer exception is not defined.

8. The notation "{***sync***, CSI}" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the *Store* instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the *Store* instruction are not performed again after the store has been performed (see Section 5.5). These properties ensure that all address translations associated with instructions preceding the *Store* instruction will be performed using the old contents of the PTE.

   The ***sync*** instruction after the *Store* instruction ensures that all lookups of the Page Table that are performed after the ***sync*** instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the ***sync*** instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).

The ***sync*** instruction also ensures that all storage accesses associated with instructions preceding the ***sync*** instruction, before the ***sync*** instruction is executed, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required or Alternate Coherence Mode attributes, before any data accesses caused by instructions following the ***sync*** instruction are performed with respect to that thread or mechanism.

9. After executing a ***mtspr*** that sets one of the TLB invalidate all bits in the MMUCSR0 to a 1, software must read MMUCSR0 using a ***mfspr*** instruction until the corresponding bit is zero and then perform the CSI, or CSI and ***sync*** as indicated in the "Required After" column.

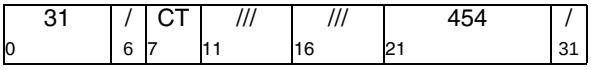
Appendix A. Implementation-Dependent Instructions

This appendix documents architectural resources that are allocated for specific implementation-sensitive functions which have scope-limited utility. Implementations may exercise reasonable flexibility in implementing these functions, but that flexibility should be limited to that allowed in this appendix.

A.1 Embedded Cache Initialization [Category: Embedded.Cache Initialization]

Data Cache Invalidate *X-form*

dci CT



If CT is not supported by the implementation, this instruction designates the primary data cache as the target data cache.

If CT is supported by the implementation, let CT designate either the primary data cache or another level of the data cache hierarchy, as specified in Section 4.3, “Cache Management Instructions”, in Book II, as the target data cache.

The contents of the target data cache of the thread executing the ***dci*** instruction are invalidated.

Software must place a ***sync*** instruction before the ***dci*** to guarantee all previous data storage accesses complete before the ***dci*** is performed.

Software must place a ***sync*** instruction after the ***dci*** to guarantee that the ***dci*** completes before any subsequent data storage accesses are performed.

This instruction is hypervisor privileged.

Special Registers Altered:

None

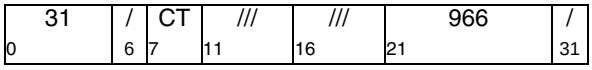
Extended Mnemonics:

Extended mnemonic for *Data Cache Invalidate*

Extended:	Equivalent to:
dccci	dci 0

Instruction Cache Invalidate *X-form*

ici CT



If CT is not supported by the implementation, this instruction designates the primary instruction cache as the target instruction cache.

If CT is supported by the implementation, let CT designate either the primary instruction cache or another level of the instruction cache hierarchy, as specified in Section 4.3, “Cache Management Instructions”, in Book II, as the target instruction cache.

The contents of the target instruction cache of the thread executing the ***ici*** instruction are invalidated.

Software must place a ***sync*** instruction before the ***ici*** to guarantee all previous instruction storage accesses complete before the ***ici*** is performed.

Software must place an ***isync*** instruction after the ***ici*** to invalidate any instructions that may have already been fetched from the previous contents of the instruction cache after the ***isync***.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonic for *Instruction Cache Invalidate*

Extended:	Equivalent to:
iccci	ici 0

A.2 Embedded Cache Debug Facility [Category: Embedded.Cache Debug]

A.2.1 Embedded Cache Debug Registers

A.2.1.1 Data Cache Debug Tag Register High

The Data Cache Debug Tag Register High (DCDBTRH) is a 32-bit Special Purpose Register. The Data Cache Debug Tag Register High is read using ***mfspr*** and is set by ***dcread***.

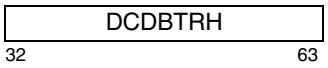


Figure 87. Data Cache Debug Tag Register High

Programming Note

An example implementation of DCDBTRH could have the following content and format.

Bit(s)	Description
32:55	**Tag Real Address** (TRA) Bits 0:23 of the lower 32 bits of the 36-bit real address associated with this cache block
56	**Valid** (V) The valid indicator for the cache block (1 indicates valid)
57:59	Reserved
60:63	**Tag Extended Real Address** (TERA) Upper 4 bits of the 36-bit real address associated with this cache block

Implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.

A.2.1.2 Data Cache Debug Tag Register Low

The Data Cache Debug Tag Register Low (DCDBTRL) is a 32-bit Special Purpose Register. The Data Cache Debug Tag Register Low is read using ***mfspr*** and is set by ***dcread***.

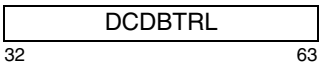


Figure 88. Data Cache Debug Tag Register Low

Programming Note

An example implementation of DCDBTRL could have the following content and format.

Bit(s)	Description
32:44	Reserved (TRA)
45	**U bit parity** (UPAR**)**
46:47	**Tag parity** (TPAR)
48:51	**Data parity** (DPAR)
52:55	**Modified** (dirty) **parity** (MPAR)
56:59	**Dirty Indicators** (D) The “dirty” (modified) indicators for each of the four doublewords in the cache block
60	**U0 Storage Attribute** (U0) The U0 storage attribute for the page associated with this cache block
61	**U1 Storage Attribute** (U1) The U1 storage attribute for the page associated with this cache block
62	**U2 Storage Attribute** (U2) The U2 storage attribute for the page associated with this cache block
63	**U3 Storage Attribute** (U3) The U3 storage attribute for the page associated with this cache block

Implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.

A.2.1.3 Instruction Cache Debug Data Register

The Instruction Cache Debug Data Register (ICDBDR) is a read-only 32-bit Special Purpose Register. The Instruction Cache Debug Data Register can be read using ***mfspr*** and is set by ***icread***.

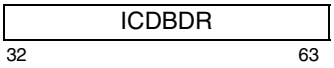


Figure 89. Instruction Cache Debug Data Register

This register is hypervisor privileged.

A.2.1.4 Instruction Cache Debug Tag Register High

The Instruction Cache Debug Tag Register High (ICDBTRH) is a 32-bit Special Purpose Register. The Instruction Cache Debug Tag Register High is read using ***mfspr*** and is set by ***icread***.

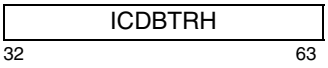


Figure 90. Instruction Cache Debug Tag Register High

Programming Note

An example implementation of ICDBTRH could have the following content and format.

Bit(s)	Description
32:55	**Tag Effective Address** (TEA) Bits 0:23 of the 32-bit effective address associated with this cache block
56	**Valid** (V) The valid indicator for the cache block (1 indicates valid)
57:58	**Tag parity** (TPAR)
59	**Instruction Data parity** (DPAR)
60:63	Reserved

Implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.

A.2.1.5 Instruction Cache Debug Tag Register Low

The Instruction Cache Debug Tag Register Low (ICDBTRL) is a 32-bit Special Purpose Register. The Instruction Cache Debug Tag Register Low is read using ***mfspr*** and is set by ***icread***.

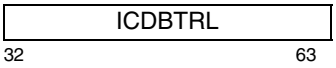


Figure 91. Instruction Cache Debug Tag Register Low

Programming Note

An example implementation of ICDBTRL could have the following content and format.

Bit(s)	Description
32:53	Reserved
54	**Translation Space** (TS) The address space portion of the virtual address associated with this cache block.
55	**Translation ID Disable** (TD) TID Disable field for the memory page associated with this cache block
56:63	**Translation ID** (TID) TID field portion of the virtual address associated with this cache block

Other implementations may support different content and format based on their cache implementation.

This register is hypervisor privileged.

A.2.2 Embedded Cache Debug Instructions

Data Cache Read *X-form*

dcread RT,RA,RB

31	RT	RA	RB	486	/
0	6	11	16	21	31

[Alternative Encoding]

31	RT	RA	RB	326	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
C  ← log2(cache size)
B  ← log2(cache block size)
IDX← EA64-C:63-B
WD ← EA64-B:61
RT0:31← undefined
RT32:63← (data cache data)[IDX]WDx32:WDx32+31
DCDBTRH← (data cache tag high)[IDX]
DCDBTRL← (data cache tag low)[IDX]
```

Let the effective address (EA) be the sum of the contents of register RA, or 0 if RA is equal to 0, and the contents of register RB.

Let C = $\log_2$(cache size in bytes).
Let B = $\log_2$(cache block size in bytes).

$EA_{64-C:63-B}$ selects one of the 2^{C-B} data cache blocks.

$EA_{64-B:61}$ selects one of the data words in the selected data cache block.

The selected word in the selected data cache block is placed into register RT.

The contents of the data cache directory entry associated with the selected data cache block are placed into DCDBTRH and DCDBTRL (see Figure 87 and Figure 88).

dcread requires software to guarantee execution synchronization before subsequent ***mfspr*** instructions can read the results of the ***dcread*** instruction into GPRs. In order to guarantee that the ***mfspr*** instructions obtain the results of the ***dcread*** instruction, a sequence such as the following must be used:

```
sync                  # ensure that all previous
                      # cache operations have
                      # completed

dcread    regT,regA,regB# read cache information;

isync                 # ensure dcread completes
                      # before attempting to
                      # read results

mfspr     regD,dcdbtrh # move high portion of tag
                      # into GPR D

mfspr     regE,dcdbtrl # move low portion of tag
                      # into GPR E
```

This instruction is hypervisor privileged.

Special Registers Altered:
DCDBTRH DCDBTRL

Programming Note

dcread can be used by a debug tool to determine the contents of the data cache, without knowing the specific addresses of the blocks which are currently contained within the cache.

Programming Note

Execution of ***dcread*** before the data cache has completed all cache operations associated with previously executed instructions (such as block fills and block flushes) is undefined.

Instruction Cache Read *X-form*

icread RA,RB

31	///	RA	RB	998	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
C  ← log2(cache size)
B  ← log2(cache block size)
IDX← EA64-C:63-B
WD ← EA64-B:61
ICDBDR← (instruction cache data)[IDX]WD×32:WD×32+31
ICDBTRH← (instruction cache tag high)[IDX]
ICDBTRL← (instruction cache tag low)[IDX]
```

Let the effective address (EA) be the sum of the contents of register RA, or 0 if RA is equal to 0, and the contents of register RB.

Let C = $\log_2$(cache size in bytes).
Let B = $\log_2$(cache block size in bytes).

$EA_{64-C:63-B}$ selects one of the 2^{C-B} instruction cache blocks.

$EA_{64-B:61}$ selects one of the data words in the selected instruction cache block.

The selected word in the selected instruction cache block is placed into ICDBDR.

The contents of the instruction cache directory entry associated with the selected cache block are placed into ICDBTRH and ICDBTRL (see Figure 90 and Figure 91).

icread requires software to guarantee execution synchronization before subsequent ***mfspr*** instructions can read the results of the ***icread*** instruction into GPRs. In order to guarantee that the ***mfspr*** instructions obtain the results of the ***icread*** instruction, a sequence such as the following must be used:

```
icread    regA,regB     # read cache information

isync                   # ensure icread completes
                        # before attempting to
                        # read results

mficdbdr  regC          # move instruction
                        # information into GPR C

mficdbtrh regD          # move high portion of
                        # tag into GPR D

mficdbtrl regE          # move low portion of tag
                        # into GPR E
```

This instruction is hypervisor privileged.

Special Registers Altered:
ICDBDR ICDBTRH ICDBTRL

Programming Note

icread can be used by a debug tool to determine the contents of the instruction cache, without knowing the specific addresses of the blocks which are currently contained within the cache.

Appendix B. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book III.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

B.1 Move To/From Special Purpose Register Mnemonics

This section defines extended mnemonics for the ***mtspr*** and ***mfspr*** instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, and for the *Move From Time Base* instruction defined in Book II.

The ***mtspr*** and ***mfspr*** instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the *Move From Time Base* instruction, which specifies the portion of the Time Base as a numeric operand.

Note: ***mftb*** serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

Table 14:Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR	
	Extended	**Equivalent to**	**Extended**	**Equivalent to**
Fixed-Point Exception Register	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Decrementer	mtdec Rx	mtspr 22,Rx	mfdec Rx	mfspr Rx,22
Save/Restore Register 0	mtsrr0 Rx	mtspr 26,Rx	mfsrr0 Rx	mfspr Rx,26
Save/Restore Register 1	mtsrr1 Rx	mtspr 27,Rx	mfsrr1 Rx	mfspr Rx,27
Special Purpose Registers G0 through G3	mtsprg n,Rx	mtspr 272+n,Rx	mfsprg Rx,n	mfspr Rx,272+n
Time Base [Lower]	mttbl Rx	mtspr 284,Rx	mftb Rx	mftb Rx,268[1] mfspr Rx,268
Time Base Upper	mttbu Rx	mtspr 285,Rx	mftbu Rx	mftb Rx,269[1] mfspr Rx,269
PPR32	mtppr32 Rx	mtspr 898, Rx	mfppr32 Rx	mfspr Rx, 898
Processor Version Register	-	-	mfpvr Rx	mfspr Rx,287

[1] The ***mftb*** instruction is Category: Phased-Out. Assemblers targeting Version 2.03 or later of the architecture should generate an ***mfspr*** instruction for the mftb and mftbu extended mnemonics; see the corresponding Assembler Note in the ***mftb*** instruction description (see Section 6.2.1 of Book II).

B.2 Data Cache Block Flush Mnemonics [Category: Embedded.Phased In]

The L field in the *Data Cache Block Flush by External PID* instruction controls the scope of the flush function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

Note: ***dcbfep*** serves as both a basic and an extended mnemonic. The Assembler will recognize a ***dcbfep*** mnemonic with three operands as the basic form, and a ***dcbfep*** mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

dcbfep RA,RB	(equivalent to: dcbfep RA,RB,0)
dcbflep RA,RB	(equivalent to: dcbfep RA,RB,1)
dcbflpep RA,RB	(equivalent to: dcbfep RA,RB,3)

Appendix C. Guidelines for 64-bit Implementations in 32-bit Mode and 32-bit Implementations

C.1 Hardware Guidelines

C.1.1 64-bit Specific Instructions

The instructions in the Category: 64-Bit are considered restricted only to 64-bit processing. A 32-bit implementation need not implement the group; likewise, the 32-bit applications will not utilize any of these instructions. All other instructions shall either be supported directly by the implementation, or sufficient infrastructure will be provided to enable software emulation of the instructions. A 64-bit implementation that is executing in 32-bit mode may choose to take an Unimplemented Instruction Exception when these 64-bit specific instructions are executed.

C.1.2 Registers on 32-bit Implementations

The Power ISA provides 32-bit and 64-bit registers. All 32-bit registers shall be supported as defined in the specification except the MSR and EPCR. The MSR shall be supported as defined in the specification except that CM is treated as a reserved bit. EPCR shall be supported as defined in the specification except that ICM and GICM are treated as reserved bits. Only bits 32:63 of the 64-bit registers are required to be implemented in hardware in a 32-bit implementation except for the 64-bit FPRs. Such 64-bit registers include the LR, the CTR, the XER, the 32 GPRs, SRR0, CSRR0, DSRR0 <E.ED>, MCSRR0. and GSRR0 <E.HV>.

For additional information, see Section 1.5.2 of Book I.

C.1.3 Addressing on 32-bit Implementations

Only bits 32:63 of the 64-bit instruction and data storage effective addresses need to be calculated and presented to main storage. Given that the only branch and data storage access instructions that are not included in Section C.1.1 are defined to prepend 32 0s to bits 32:63 of the effective address computation, a 32-bit implementation can simply bypass the prepending of the 32 0s when implementing these instructions. For Branch to Link Register and Branch to Count Register instructions, given the LR and CTR are implemented only as 32-bit registers, only concatenating 2 0s to the right of bits 32:61 of these registers is necessary to form the 32-bit branch target address.

For next sequential instruction address computation, the behavior is the same as for 64-bit implementations in 32-bit mode.

C.1.4 TLB Fields on 32-bit Implementations

32-bit implementations should support bits 32:53 of the Effective Page Number (EPN) field in the TLB. This size provides support for a 32-bit effective address, which Power ISA ABIs may have come to expect to be available. 32-bit implementations may support greater than 32-bit real addresses by supporting more than bits 32:53 of the Real Page Number (RPN) field in the TLB.

C.1.5 Thread Control and Status on 32-bit Implementations

As the TEN and TENSR are 32-bits on 32-bit implementations, the maximum number of threads for such implementations is limited to 32.

C.2 32-bit Software Guidelines

C.2.1 32-bit Instruction Selection

Any software that uses any of the instructions listed in Category: 64-Bit shall be considered 64-bit software, and correct execution cannot be guaranteed on 32-bit implementations. Generally speaking, 32-bit software should avoid using any instruction or instructions that depend on any particular setting of bits 0:31 of any 64-bit application-accessible system register, including General Purpose Registers, for producing the correct 32-bit results. Context switching may or may not preserve the upper 32 bits of application-accessible 64-bit

system registers and insertion of arbitrary settings of those upper 32 bits at arbitrary times during the execution of the 32-bit application must not affect the final result.

Appendix D. Example Performance Monitor [Category: Embedded.Performance Monitor]

D.1 Overview

This appendix describes an example of a Performance Monitor facility. It defines an architecture suitable for performance monitoring facilities in the Embedded environment. The architecture itself presents only programming model visible features in conjunction with architecturally defined behavioral features. Much of the selection of events is by necessity implementation-dependent and is not described as part of the architecture; however, this document provides guidelines for some features of a performance monitor implementation that should be followed by all implementations.

The example Performance Monitor facility provides the ability to monitor and count predefined events such as clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the Performance Monitor exception. While most of the specific events are not architected, the mechanism of controlling data collection is.

The example Performance Monitor facility can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize performance in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

D.2 Programming Model

The example Performance Monitor facility defines a set of Performance Monitor Registers (PMRs) that are used to collect and control performance data collection and an interrupt to allow intervention by software. The PMRs provide various controls and access to collected data. They are categorized as follows:

- Counter registers. These registers are used for data collection. The occurrence of selected events are counted here. These registers are named PMC0..15. User and supervisor level access to these registers is through different PMR numbers allowing different access rights.
- Global controls. This register control global settings of the Performance Monitor facility and affect all counters. This register is named PMGC0. User and supervisor level access to these registers is through different PMR numbers allowing different access rights. In addition, a bit in the MSR (MSR_{PMM}) is defined to enable/disable counting.
- Local controls. These registers control settings that apply only to a particular counter. These registers are named PMLCa0..15 and PMLCb0..15. User and supervisor level access to these registers is through different PMR numbers allowing different access rights. Each set of local control registers (PMLCa*n* and PMLCb*n*) contains controls that apply to the associated same numbered counter register (e.g. PMLCa0 and PMLCb0 contain controls for PMC0 while PMLCa1 and PMLCb1 contain controls for PMC1).

Assembler Note

The counter registers, global controls, and local controls have alias names which cause the assembler to use different PMR numbers. The names PMC0...15, PMGC0, PMLCa0...15, and PMLCb0...15 cause the assembler to use the supervisor level PMR number, and the names UPMC0...15, UPMGC0, UPMLCa0...15, and UPMLCb0...15 cause the assembler to use the user-level PMR number.

Architecture Note

The two mark values (0 and 1) are equivalent except with respect to interrupts. That is, either mark value can be specified for a given process, and either mark value can control whether the PMCs are incremented, but interrupts always cause the mark value in the MSR to be set to 0.

A given implementation may implement fewer counter registers (and their associated control registers) than are architected. Architected counter and counter control registers that are not implemented behave the same as unarchitected Performance Monitor Registers.

PMRs are described in Section D.3.

Software uses the global and local controls to select which events are counted in the counter registers, when such events should be counted, and what action should be taken when a counter overflows. Software can use the collected information to determine performance attributes of a given segment of code, a process, or the entire software system. PMRs can be read by software using the ***mfpmr*** instruction and PMRs can be written by using the ***mtpmr*** instruction. Both instructions are described in Section D.4.

Since counters are defined as 32-bit registers, it is possible for the counting of some events to overflow. A Performance Monitor interrupt is provided that can be programmed to occur in the event of a counter overflow. The Performance Monitor interrupt is described in detail in Section D.2.5 and Section D.2.6.

D.2.1 Event Counting

Event counting can be configured in several different ways. This section describes configurability and specific unconditional counting modes.

D.2.2 Thread Context Configurability

Counting can be enabled if conditions in the thread state match a software-specified condition. Because a software task scheduler may switch a thread's execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The Performance Monitor mark bit, MSR_{PMM}, is used for this purpose. System software may set this bit to 1 when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR_{PR} and MSR_{PMM} together define a state that the thread (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified by the PMLCan_{FCS}, PMLCan_{FCU}, PMLCan_{FCM1} and PMLCan_{FCM0} fields in PMLCa*n* (the state for which monitoring is enabled), counting is enabled for PMC*n*.

Each event, on an implementation basis, may count regardless of the value of MSR_{PMM}. The counting behavior of each event should be documented in the User's Manual.

The thread states and the settings of the PMLCan_{FCS}, PMLCan_{FCU}, PMLCan_{FCM1} and PMLCan_{FCM0} fields in PMLCa*n* necessary to enable monitoring of each thread state are shown in Figure 92.

Thread State	FCS	FCU	FCM1	FCM0
Marked	0	0	0	1
Not marked	0	0	1	0
Supervisor	0	1	0	0
User	1	0	0	0
Marked and supervisor	0	1	0	1
Marked and user	1	0	0	1
Not marked and supervisor	0	1	1	0
Not mark and user	1	0	1	0
All	0	0	0	0
None	X	X	1	1
None	1	1	X	X

Figure 92. Thread States and PMLCa*n* Bit Settings

Two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR_{PMM} and MSR_{PR}. This can be accomplished by setting PMLCan_{FCS}, PMLCan_{FCU}, PMLCan_{FCM1}, and PMLCan_{FCM0} to 0 for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR_{PMM} and MSR_{PR}. This can be accomplished by setting $PMGC0_{FAC}$ to 1 or by setting PMLCan_{FC} to 1 for each counter control. Alternatively, this can be accomplished by setting PMLCan_{FCM1} to 1 and PMLCan_{FCM0} to 1 for each counter control or by setting PMLCan_{FCS} to 1 and PMLCan_{FCU} to 1 for each counter control.

Programming Note

Events may be counted in a fuzzy manner. That is, events may not be counted precisely due to the nature of an implementation. Users of the Performance Monitor facility should be aware that an event may be counted even if it was precisely filtered, though it should not have been. In general such discrepancies are statistically unimportant and users should not assume that counts are explicitly accurate.

D.2.3 Event Selection

Events to count are determined by placing an implementation defined event value into the $PMLCa0..15_{EVENT}$ field. Which events may be programmed into which counter are implementation specific and should be defined in the User's Manual. In general, most events may be programmed into any of the implementation available counters. Programming a counter with an event that is not supported for that counter gives boundedly undefined results.

Programming Note

Event name and event numbers will differ greatly across implementations and software should not expect that events and event names will be consistent.

D.2.4 Thresholds

Thresholds are values that must be exceeded for an event to be counted. Threshold values are programmed in the PMLCb0..15$_{THRESHOLD}$ field. The events which may be thresholded and the units of each event that may be thresholded are implementation-dependent. Programming a threshold value for an event that is not defined to use a threshold gives boundedly undefined results.

D.2.5 Performance Monitor Exception

A Performance Monitor exception occurs when counter overflow detection is enabled and a counter overflows. More specifically, for each counter register *n*, if PMGC0$_{PMIE}$=1, PMLCan_{CE}=1, PMCn_{OV}=1, and the performance monitor interrupt is enabled in the MSR (see below), a Performance Monitor exception is said to exist. The Performance Monitor exception condition will cause a Performance Monitor interrupt if the exception is the highest priority exception.

The performance monitor interrupt enabling conditions in the MSR are as follows:

- If category E.HV is implemented, the performance monitor interrupt is enabled if:
  - EPCR$_{PMGS}$=0 and (MSR$_{EE}$=1 or MSR$_{GS}$=1). The interrupt will be directed to the hypervisor state. or if
  - EPCR$_{PMGS}$=1 and (MSR$_{EE}$=1 and MSR$_{GS}$=1). The interrupt will be directed to the guest state.
- If category E.HV is not implemented, the performance monitor interrupt is enabled if MSR$_{EE}$=1.

The Performance Monitor exception is level sensitive and the exception condition may cease to exist if any of the required conditions fail to be met. Thus it is possible for a counter to overflow and continue counting events until PMCn_{OV} becomes 0 without taking a Performance Monitor interrupt if the enabling conditions in the MSR are not met during the overflow condition. To avoid this, software should program the counters to freeze if an overflow condition is detected (see Section D.3.4).

D.2.6 Performance Monitor Interrupt

A Performance Monitor interrupt occurs when a Performance Monitor exception exists and no higher priority exception exists. When a Performance Monitor interrupt occurs, SRR0 and SRR1 (GSRR0 and GSRR1 if EPCR$_{PMGS}$=1 <E.HV>) record the current state of the NIA and the MSR and the MSR is set to handle the interrupt. Instruction execution resumes at an address based on which categories are supported:

- If both Interrupt Fixed Offsets [Category: Embedded.Phased-In] and Embedded.Hypervisor are supported, instruction execution resumes at:
  - address IVPR$_{0:51}$ || 0x260 when EPCR$_{PMGS}$=0. Otherwise, instruction execution resumes at address GIVPR$_{0:47}$ || 0x260.
- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] is supported and Embedded.Hypervisor is not supported, instruction execution resumes at address IVPR$_{0:51}$ || 0x260.
- If Interrupt Fixed Offsets [Category: Embedded.Phased-In] is not supported and Embedded.Hypervisor is supported, instruction execution resumes at:
  - address IVPR$_{0:51}$ || IVOR35$_{48:59}$ || 0b0000. when EPCR$_{PMGS}$=0. Otherwise, instruction execution resumes at address GIVPR$_{0:47}$ || GIVOR35$_{48:59}$ || 0b0000.
- If neither Interrupt Fixed Offsets [Category: Embedded.Phased-In] nor Embedded.Hypervisor is supported, instruction execution resumes at address IVPR$_{0:51}$ || IVOR35$_{48:59}$ || 0b0000.

The Performance Monitor interrupt is precise and asynchronous.

Programming Note

When taking a Performance Monitor interrupt software should clear the overflow condition by reading the counter register and setting the counter register to a non-overflow value since the normal return from the interrupt will set MSR$_{EE}$ back to 1.

D.3 Performance Monitor Registers

D.3.1 Performance Monitor Global Control Register 0

The Performance Monitor Global Control Register 0 (PMGC0) controls all Performance Monitor counters.

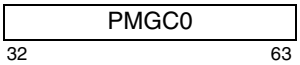


Figure 93. [User] Performance Monitor Global Control Register 0

These bits are interpreted as follows:

Bit	Description

32 ***Freeze All Counters*** (FAC)
The FAC bit is sticky; that is, once set to 1 it remains set to 1 until it is set to 0 by an ***mtpmr*** instruction.

0 The PMCs can be incremented (if enabled by other Performance Monitor control fields).
1 The PMCs can not be incremented.

33 ***Performance Monitor Interrupt Enable*** (PMIE)

0 Performance Monitor interrupts are disabled.
1 Performance Monitor interrupts are enabled and occur when an enabled condition or event occurs. Enabled conditions and events are described in Section D.2.5.

34 ***Freeze Counters on Enabled Condition or Event*** (FCECE)
Enabled conditions and events are described in Section D.2.5.

0 The PMCs can be incremented (if enabled by other Performance Monitor control fields).
1 The PMCs can be incremented (if enabled by other Performance Monitor control fields) only until an enabled condition or event occurs. When an enabled condition or event occurs, $PMGC0_{FAC}$ is set to 1. It is the user's responsibility to set $PMGC0_{FAC}$ to 0.

35:63 Reserved

The UPMGC0 register is an alias to the PMGC0 register for user mode read only access.

D.3.2 Performance Monitor Local Control A Registers

The Performance Monitor Local Control A Registers 0 through 15 (PMLCa0..15) function as event selectors and give local control for the corresponding numbered Performance Monitor counters. PMLCa works with the corresponding numbered PMLCb register.

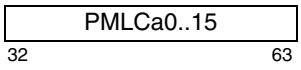


Figure 94. [User] Performance Monitor Local Control A Registers

PMLCa is set to 0 at reset. These bits are interpreted as follows:

Bit Description

32 ***Freeze Counter*** (FC)

0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
1 The PMC can not be incremented.

33 ***Freeze Counter in Supervisor State*** (FCS)

0 The PMC is incremented (if enabled by other Performance Monitor control fields).
1 The PMC can not be incremented if MSR_{PR} is 0.

34 ***Freeze Counter in User State*** (FCU)

0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
1 The PMC can not be incremented if MSR_{PR} is 1.

35 ***Freeze Counter while Mark is Set*** (FCM1)

0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
1 The PMC can not be incremented if MSR_{PMM} is 1.

36 ***Freeze Counter while Mark is Cleared*** (FCM0)

0 The PMC can be incremented (if enabled by other Performance Monitor control fields).
1 The PMC can not be incremented if MSR_{PMM} is 0.

37 ***Condition Enable*** (CE)

0 Overflow conditions for PMC*n* cannot occur (PMC*n* cannot cause interrupts, cannot freeze counters)
1 Overflow conditions occur when the most-significant-bit of PMC*n* is equal to 1.

It is recommended that CE be set to 0 when counter PMC*n* is selected for chaining; see Section D.5.1.

38 ***Freeze Counter in Guest State*** (FCGS)

0 The PMC is incremented (if enabled by other Performance Monitor control fields).
1 The PMC can not be incremented if MSR_{GS} is 1.

39 ***Freeze Counter in Hypervisor State*** (FCHS)

0 The PMC is incremented (if enabled by other Performance Monitor control fields).
1 The PMC can not be incremented if MSR_{GS} is 0 and MSR_{PR} is 0.

40 Reserved

41:47 ***Event Selector*** (EVENT)
Up to 128 events selectable; see Section D.2.3.

48:53 Setting is implementation-dependent.

54:63 Reserved

The UPMLCa0..15 registers are aliases to the PMLCa0..15 registers for user mode read only access.

D.3.3 Performance Monitor Local Control B Registers

The Performance Monitor Local Control B Registers 0 through 15 (PMLCb0..15) specify a threshold value and a multiple to apply to a threshold event selected for the corresponding Performance Monitor counter. Threshold capability is implementation counter dependent. Not all events or all counters of an implementation are guaranteed to support thresholds. PMLCb works with the corresponding numbered PMLCa register.

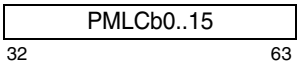


Figure 95. [User] Performance Monitor Local Control B Register

PMLCb is set to 0 at reset. These bits are interpreted as follows:

Bit **Description**

32:52 Reserved

53:55 ***Threshold Multiple*** (THRESHMUL)

000 Threshold field is multiplied by 1 (THRESHOLD × 1)
001 Threshold field is multiplied by 2 (THRESHOLD × 2)
010 Threshold field is multiplied by 4 (THRESHOLD × 4)
011 Threshold field is multiplied by 8 (THRESHOLD × 8)
100 Threshold field is multiplied by 16 (THRESHOLD × 16)
101 Threshold field is multiplied by 32 (THRESHOLD × 32)
110 Threshold field is multiplied by 64 (THRESHOLD × 64)
111 Threshold field is multiplied by 128 (THRESHOLD × 128)

56:57 Reserved

58:63 ***Threshold*** (THRESHOLD)
Only events that exceed the value THRESHOLD multiplied as described by THRESHMUL are counted. Events to which a threshold value applies are implementation-dependent as are the unit (for example duration in cycles) and the granularity with which the threshold value is interpreted.

Programming Note

By varying the threshold value, software can obtain a profile of the event characteristics subject to thresholding. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can measure the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

The UPMLCb0..15 registers are aliases to the PMLCb0..15 registers for user mode read only access.

D.3.4 Performance Monitor Counter Registers

The Performance Monitor Counter Registers (PMC0..15) are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count up to 128 events.

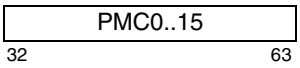


Figure 96. [User] Performance Monitor Counter Registers

PMCs are set to 0 at reset. These bits are interpreted as follows:

Bit **Description**

32 ***Overflow*** (OV)

0 Counter has not reached an overflow state.
1 Counter has reached an overflow state.

33:63 ***Counter Value*** (CV)
Indicates the number of occurrences of the specified event.

The minimum value for a counter is 0 (0x0000_0000) and the maximum value is 4,294,967,295 (0xFFFF_FFFF). A counter can increment up to the maximum value and then wraps to the minimum value. A counter enters the overflow state when the high-order bit is set to 1, which normally occurs only when the counter increments from a value below 2,147,483,648 (0x8000_0000) to a value greater than or equal to 2,147,483,648 (0x8000_0000).

Several different actions may occur when an overflow state is reached, depending on the configuration:

- If PMLCan_{CE} is 0, no special actions occur on overflow: the counter continues incrementing, and no exception is signaled.
- If PMLCan_{CE} and PMGC0$_{FCECE}$ are 1, all counters are frozen when PMC*n* overflows.
- If PMLCan_{CE}, PMGC0$_{PMIE}$, and MSR$_{EE}$ are 1, an exception is signalled when PMC*n* reaches over-

flow. Note that the interrupts are masked by setting MSR_{EE} to 0. An overflow condition may be present while MSR_{EE} is zero, but the interrupt is not taken until MSR_{EE} is set to 1.

If an overflow condition occurs while MSR_{EE} is 0 (the exception is masked), the exception is still signalled once MSR_{EE} is set to 1 if the overflow condition is still present and the configuration has not been changed in the meantime to disable the exception; however, if MSR_{EE} remains 0 until after the counter leaves the overflow state (MSB becomes 0), or if MSR_{EE} remains 0 until after PMLCan_{CE} or $PMGC0_{PMIE}$ are set to 0, the exception does not occur.

Programming Note

Loading a PMC with an overflowed value can cause an immediate exception. For example, if $PMLCan_{CE}$, $PMGC0_{PMIE}$, and MSR_{EE} are all 1, and an ***mtpmr*** loads an overflowed value into a PMCn that previously held a non-overflowed value, then an interrupt will be generated before any event counting has occurred.

The following sequence is generally recommended for setting the counter values and configurations.

1. Set $PMGC0_{FAC}$ to 1 to freeze the counters.
2. Perform a series of ***mtpmr*** operations to initialize counter values and configure the control registers
3. Release the counters by setting $PMGC0_{FAC}$ to 0 with a final ***mtpmr***.

D.4 Performance Monitor Instructions

Move From Performance Monitor Register XFX-form

mfpmr RT,PMRN

31	RT	pmrn	334	/
0	6	11	21	31

```
n ← pmrn5:9 || pmrn0:4
if length(PMR(n)) = 64 then
   RT ← PMR(n)
else
   RT ← 320 || PMR(n)32:63
```

Let PMRN denote a Performance Monitor Register number and PMR the set of Performance Monitor Registers.

The contents of the designated Performance Monitor Register are placed into register RT.

The list of defined Performance Monitor Registers and their privilege class is provided in Figure 97.

Execution of this instruction specifying a defined and privileged Performance Monitor Register when MSR_{PR}=1 will result in a Privileged Instruction exception.

Category: Embedded.Hypervisor]
If $MSRP_{PMMP}$ = 1 and MSR_{GS} = 1, execution of this instruction specifying a defined Performance Monitor Register sets RT to 0.

Execution of this instruction specifying an undefined Performance Monitor Register will either result in an Illegal Instruction exception or will produce an undefined value for register RT.

Special Registers Altered:

None

Move To Performance Monitor Register XFX-form

mtpmr PMRN,RS

31	RS	pmrn	462	/
0	6	11	21	31

```
n ← pmrn5:9 || pmrn0:4
if length(PMR(n)) = 64 then
   PMR(n) ← (RS)
else
   PMR(n) ← (RS)32:63
```

Let PMRN denote a Performance Monitor Register number and PMR the set of Performance Monitor Registers.

The contents of the register RS are placed into the designated Performance Monitor Register.

The list of defined Performance Monitor Registers and their privilege class is provided in Figure 97.

Execution of this instruction specifying a defined and privileged Performance Monitor Register when MSR_{PR}=1 will result in a Privileged Instruction exception.

[Category: Embedded.Hypervisor]
If $MSRP_{PMMP}$ = 1 and MSR_{GS} = 1 and MSR_{PR} = 0, execution of this instruction specifying a defined Performance Monitor Register results in a Embedded Hypervisor Privilege exception.

Execution of this instruction specifying an undefined Performance Monitor Register will either result in an Illegal Instruction exception or will perform no operation.

Special Registers Altered:

None

decimal	PMR[1]	Register Name	Privileged		Cat
	$pmrn_{5:9}$ $pmrn_{0:4}$		mtpmr	mfpmr	
0-15	00000 0xxxx	PMC0..15	-	no	E.PM
16-31	00000 1xxxx	PMC0..15	yes	yes	E.PM
128-143	00100 0xxxx	PMLCA0..15	-	no	E.PM
144-159	00100 1xxxx	PMLCA0..15	yes	yes	E.PM
256-271	01000 0xxxx	PMLCB0..15	-	no	E.PM
272-287	01000 1xxxx	PMLCB0..15	yes	yes	E.PM
384	01100 00000	PMGC0	-	no	E.PM
400	01100 10000	PMGC0	yes	yes	E.PM

\- This register is not defined for this instruction.
1 Note that the order of the two 5-bit halves of the PMR number is reversed.

Figure 97. Embedded.Peformance Monitor PMRs

D.5 Performance Monitor Software Usage Notes

D.5.1 Chaining Counters

An implementation may contain events that are used to "chain" counters together to provide a larger range of event counts. This is accomplished by programming the desired event into one counter and programming another counter with an event that occurs when the first counter transitions from 1 to 0 in the most significant bit.

The counter chaining feature can be used to decrease the processing pollution caused by Performance Monitor interrupts, (things like cache contamination, and pipeline effects), by allowing a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where the first counter's carry-out event acts like a carry-out feeding the second counter. By defining the event of interest to be another PMC's overflow generation, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, an overflow may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:
  mfpmr     Rx,pmctr1     #load from upper counter
  mfpmr     Ry,pmctr0     #load from lower counter
  mfpmr     Rz,pmctr1     #load from upper counter
  cmp       cr0,0,Rz,Rx   #see if 'old' = 'new'
  bc        4,2,loop
             #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. The above sequence is not necessary if the counters are frozen.

D.5.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. Assume an example event, dLFB load miss cycles, requires a threshold value. A dLFB load miss cycles event is counted only when the number of cycles spent recovering from the miss is greater than the threshold. If the event is counted on two counters and each counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds expedites code profiling significantly.

Book VLE:

Power ISA Operating Environment Architecture - Variable Length Encoding (VLE) Environment [Category: Variable Length Encoding]

Chapter 1. Variable Length Encoding Introduction

This chapter describes computation modes, document conventions, a processor overview, instruction formats, storage addressing, and instruction addressing.

1.1 Overview

Variable Length Encoding (VLE) is a code density optimized re-encoding of much of the instruction set defined by Books I, II, and III-E using both 16-bit and 32-bit instruction formats.

VLE offers more efficient binary representations of applications for the Embedded processor spaces where code density plays a major role in affecting overall system cost, and to a somewhat lesser extent, performance.

VLE is a supplement to the instruction set defined by Book I-III and code pages using VLE encoding or non-VLE encoding can be intermingled in a system providing focus on both high performance and code density where most needed.

VLE provides alternative encodings to instructions defined in Books I-III to enable reduced code footprint. This set of alternative encodings is selected on a page basis. A single storage attribute bit selects between standard instruction encodings and VLE instructions for that page of memory.

Instruction encodings in pages marked as VLE are either 16 or 32 bits long, and are aligned on 16-bit boundaries. Because of this, all instruction pages marked as VLE are required to use Big-Endian byte ordering.

The programming model uses the same register set with both instruction set encodings, although some registers are not accessible by VLE instructions using the 16-bit formats and not all condition register (CR) fields are used by *Conditional Branch* instructions or instructions that access the condition register executing from a VLE instruction page. In addition, immediate fields and displacements differ in size and use, due to the more restrictive encodings imposed by VLE instruction formats.

VLE additional instruction fields are described in Section 1.4.19, “Instruction Fields”.

Other than the requirement of Big-Endian byte ordering for instruction pages and the additional storage attribute to identify whether the instruction page corresponds to a VLE section of code, VLE complies with the memory model, register model, timer facilities, debug facilities, and interrupt/exception model defined in Book I-III and therefore execute in the same environment as non-VLE instructions.

1.2 Documentation Conventions

Book VLE adheres to the documentation conventions defined inSection 1.3 of Book I. Note however that this book defines instructions that apply to the User Instruction Set Architecture, the Virtual Environment Architecture, and the Operating Environment Architecture.

1.2.1 Description of Instruction Operation

The RTL (register transfer language) descriptions in Book VLE conform to the conventions described in Section 1.3.4 of Book I.

1.3 Instruction Mnemonics and Operands

The description of each instruction includes the mnemonic and a formatted list of operands. VLE instruction semantics are either identical or similar to those of other instructions in the architecture. Where the semantics, side-effects, and binary encodings are identical, the standard mnemonics and formats are used. Such unchanged instructions are listed and appropriately referenced, but the instruction definitions are not replicated in this book. Where the semantics are similar but the binary encodings differ, the standard mnemonic is typically preceded with an ***e***_ to denote a VLE instruction. To distinguish between similar instructions available in both 16- and 32-bit forms under VLE and standard instructions, VLE instructions encoded with 16 bits have an ***se***_ prefix. The following are examples:

```
stwx RS,RA,RB     // standard Book I instruction
e_stw RS,D(RA)    // 32-bit VLE instruction
se_stw RZ,SD4(RX) // 16-bit VLE instruction
```

1.4 VLE Instruction Formats

All VLE instructions to be executed are either two or four bytes long and are halfword-aligned in storage. Thus, whenever instruction addresses are presented to the processor (as in *Branch* instructions), the low-order bit is treated as 0. Similarly, whenever the processor generates an instruction address, the low-order bit is zero.

The format diagrams given below show horizontally all valid combinations of instruction fields. Only those formats that are unique to VLE-defined instructions are included here. Instruction forms that are available in VLE or non-VLE mode are described in Section 1.6 of Book I and are not repeated here.

In some cases an instruction field must contain a particular value. If a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are as described for invalid instruction forms in Book I.

VLE instructions use split field notation as defined in Section 1.6 of Book I.

1.4.1 BD8-form (16-bit Branch Instructions)

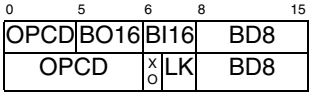


Figure 1. BD8 instruction format

1.4.2 C-form (16-bit Control Instructions)

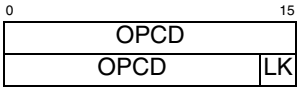


Figure 2. C instruction format

1.4.3 IM5-form (16-bit register + immediate Instructions)

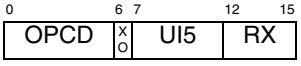


Figure 3. IM5 instruction format

1.4.4 OIM5-form (16-bit register + offset immediate Instructions)

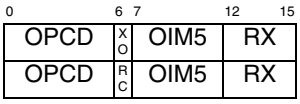


Figure 4. OIM5 instruction format

1.4.5 IM7-form (16-bit Load immediate Instructions)

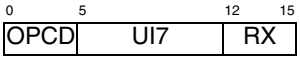


Figure 5. IM7 instruction format

1.4.6 R-form (16-bit Monadic Instructions)

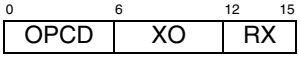


Figure 6. R instruction format

1.4.7 RR-form (16-bit Dyadic Instructions)

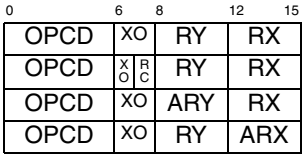


Figure 7. RR instruction format

1.4.8 SD4-form (16-bit Load/Store Instructions)

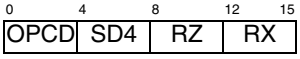


Figure 8. SD4 instruction format

1.4.9 BD15-form

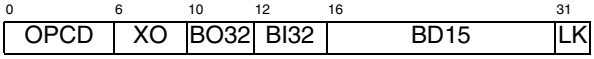


Figure 9. BD15 instruction format

1.4.10 BD24-form

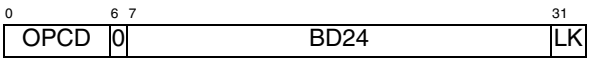


Figure 10. BD24 instruction format

1.4.11 D8-form

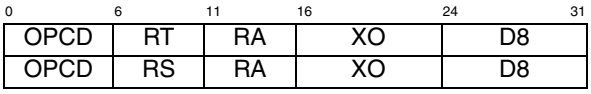
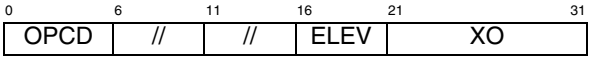


Figure 11. D8 instruction format

1.4.12 ESC-form



1.4.13 I16A-form

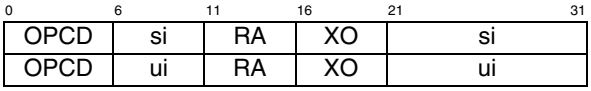


Figure 12. I16A instruction format

1.4.14 I16L-form

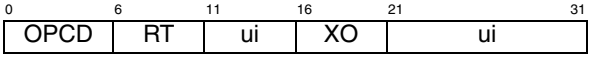


Figure 13. I16L instruction format

1.4.15 M-form

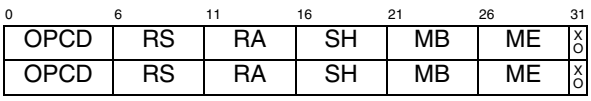


Figure 14. M instruction format

1.4.16 SCI8-form

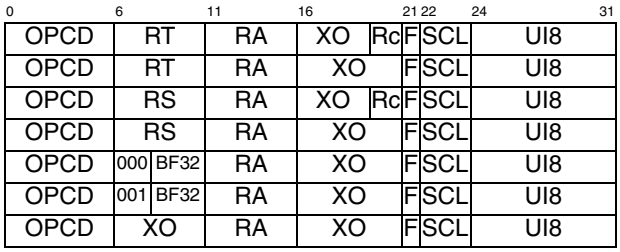


Figure 15. SC18 instruction format

1.4.17 LI20-form

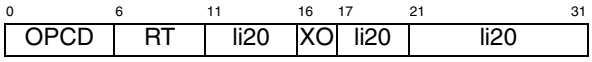


Figure 16. LI20 instruction format

1.4.18 X-form

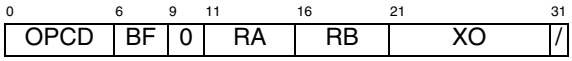


Figure 17. X instruction format

1.4.19 Instruction Fields

VLE uses instruction fields defined in Section 1.6.28 of Book I as well as VLE-defined instruction fields defined below.

ARX (12:15)

Field used to specify an “alternate” General Purpose Register in the range R8:R23 to be used as a destination.

ARY (8:11)

Field used to specify an “alternate” General Purpose Register in the range R8:R23 to be used as a source.

BD8 (8:15), BD15 (16:30), BD24 (7:30)

Immediate field specifying a signed two's complement branch displacement which is concatenated on the right with 0b0 and sign-extended to 64 bits.

BD15. (Used by 32-bit branch conditional class instructions) A 15-bit signed displacement that is sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BD24. (Used by 32-bit branch class instructions) A 24-bit signed displacement that is

sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BD8. (Used by 16-bit branch and branch conditional class instructions) An 8-bit signed displacement that is sign-extended and shifted left one bit (concatenated with 0b0) and then added to the current instruction address to form the branch target address.

BI16 (6:7), BI32 (12:15)
Field used to specify one of the Condition Register fields to be used as a condition of a Branch Conditional instruction.

BO16 (5), BO32 (10:11)

Field used to specify whether to branch if the condition is true, false, or to decrement the Count Register and branch if the Count Register is not zero in a Branch Conditional instruction.

BF32 (9:10)
Field used to specify one of the Condition Register fields to be used as a target of a compare instruction.

D8 (24:31)
The D8 field is a 8-bit signed displacement which is sign-extended to 64 bits.

ELEV (16:20)
Field used by the ***e_sc*** instruction.

F (21) Fill value used to fill the remaining 56 bits of a scaled-immediate 8 value.

LI20 (17:20 || 11:15 || 21:31)
A 20-bit signed immediate value which is sign-extended to 64 bits for the ***e_li*** instruction.

LK (7, 15, 31)
LINK bit.

0 Do not set the Link Register.
1 Set the Link Register. The sum of the value 2 or 4 and the address of the *Branch* instruction is placed into the Link Register.

OIM5 (7:11)
Offset Immediate field used to specify a 5-bit unsigned fixed-point value in the range [1:32] encoded as [0:31]. Thus the binary encoding of 0b00000 represents an immediate value of 1, 0b00001 represents an immediate value of 2, and so on.

OPCD (0:3, 0:4, 0:5, 0:9, 0:14, 0:15)
Primary opcode field.

Rc (6, 7, 20, 31)
RECORD bit.

0 Do not alter the Condition Register.
1 Set Condition Register Field 0.

RX (12:15)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source or as a destination. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

RY (8:11)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

RZ (8:11)
Field used to specify a General Purpose Register in the ranges R0:R7 or R24:R31 to be used as a source or as a destination for load/store data. R0 is encoded as 0b0000, R1 as 0b0001, etc. R24 is encoded as 0b1000, R25 as 0b1001, etc.

SCL (22:23)
Field used to specify a scale amount in *Immediate* instructions using the SCI8-form. Scaling involves left shifting by 0, 8, 16, or 24 bits.

SD4 (4:7)
Used by 16-bit load and store class instructions. The SD4 field is a 4-bit unsigned immediate value zero-extended to 64 bits, shifted left according to the size of the operation, and then added to the base register to form a 64-bit EA. For byte operations, no shift is performed. For half-word operations, the immediate is shifted left one bit (concatenated with 0b0). For word operations, the immediate is shifted left two bits (concatenated with 0b00).SI (6:10 || 21:31, 11:15 || 21:31)
A 16-bit signed immediate value sign-extended to 64 bits and used as one operand of the instruction.

UI (6:10 || 21:31, 11:15 || 21:31)
A 16-bit unsigned immediate value zero-extended to 64 bits or padded with 16 zeros and used as one operand of the instruction. The instruction encoding differs between the I16A and I16L instruction formats as shown in Section 1.4.13 and Section 1.4.14.

UI5 (7:11)
Immediate field used to specify a 5-bit unsigned fixed-point value.

UI7 (5:11)
Immediate field used to specify a 7-bit unsigned fixed-point value.

UI8 (24:31)
Immediate field used to specify an 8-bit unsigned fixed-point value.

XO (6, 6:7, 6:10, 6:11, 16, 16:19, 16:20, 16:23, 31)
Extended opcode field.

Assembler Note

For scaled immediate instructions using the SCI8-form, the instruction assembly syntax requires a single immediate value, *sci8*, that the assembler will synthesize into the appropriate F, SCL, and UI8 fields. The F, SCL, and UI8 fields must be able to be formed correctly from the given *sci8* value or the assembler will flag the assembly instruction as an error.

Chapter 2. VLE Storage Addressing

A program references memory using the effective address (EA) computed by the processor when it executes a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II and Book III-E), or when it fetches the next sequential instruction.

2.1 Data Storage Addressing Modes

Table 1 lists data storage addressing modes supported by the VLE category.

Table 1: Data Storage Addressing Modes

Mode	Form	Description
Base+16-bit displacement (32-bit instruction format)	D-form	The 16-bit D field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0 to produce the EA.
Base+8-bit displacement (32-bit instruction format)	D8-form	The 8-bit D8 field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0 to produce the EA.
Base+scaled 4-bit displacement (16-bit instruction format)	SD4-form	The 4-bit SD4 field zero-extended, scaled (shifted left) according to the size of the operand, and added to the contents of the GPR designated by RX to produce the EA. (Note that RX = 0 is not a special case.)
Base+Index (32-bit instruction format)	X-form	The GPR contents designated by RB are added to the GPR contents designated by RA or to zero if RA = 0 to produce the EA.

2.2 Instruction Storage Addressing Modes

Table 2 lists instruction storage addressing modes supported by the VLE category.

Table 2: Instruction Storage Addressing Modes

Mode	Description
Taken BD24-form Branch instructions (32-bit instruction format)	The 24-bit BD24 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction.
Taken B15-form Branch instructions (32-bit instruction format)	The 15-bit BD15 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction to form the EA of the next instruction.
Take BD8-form Branch instructions (16-bit instruction format)	The 8-bit BD8 field is concatenated on the right with 0b0, sign-extended, and then added to the address of the branch instruction to form the EA of the next instruction.
Sequential instruction fetching (or non-taken branch instructions)	The value 4 [2] is added to the address of the current 32-bit [16-bit] instruction to form the EA of the next instruction. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFC [0xFFFF_FFFF_FFFF_FFFE] in 64-bit mode or 0xFFFF_FFFC [0xFFFF_FFFE] in 32-bit mode, the address of the next sequential instruction is undefined.
Any Branch instruction with LK = 1 (32-bit instruction format)	The value 4 is added to the address of the current branch instruction and the result is placed into the LR. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFC in 64-bit mode o r0xFFFF_FFFC in 32-bit mode, the result placed into the LR is undefined.
Branch ***se_bl. se_blrl. se_bctrl*** instructions (16-bit instruction format)	The value 2 is added to the address of the current branch instruction and the result is placed into the LR. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFE in 64-bit mode or 0xFFFF_FFFE in 32-bit mode, the result placed into the LR is undefined.

2.2.1 Misaligned, Mismatched, and Byte Ordering Instruction Storage Exceptions

A *Misaligned Instruction Storage Exception* occurs when an implementation which supports VLE attempts to execute an instruction that is not 32-bit aligned and the VLE storage attribute is not set for the page that corresponds to the effective address of the instruction. The attempted execution can be the result of a *Branch* instruction which has bit 62 of the target address set to 1 or the result of an ***rfi***, ***se_rfi***, ***rfci***, ***se_rfci***, ***rfdi***, ***se_rfdi***, ***rfgi***, ***se_rfgi***, ***rfmci***, or ***se_rfmci*** instruction which has bit 62 set in SRR0, SRR0, CSRR0, CSRR0, DSRR0, DSRR0, GSRR0, GSRR0, MCSRR0, or MCSRR0 respectively. If a Misaligned Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0(GSRR0) to the misaligned address for which execution was attempted.

A *Mismatched Instruction Storage Exception* occurs when an implementation which supports VLE attempts to execute an instruction that crosses a page boundary for which the first page has the VLE storage attribute set to 1 and the second page has the VLE storage attribute bit set to 0. If a Mismatched Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0(GSRR0) to the misaligned address for which execution was attempted.

A *Byte Ordering Instruction Storage Exception* occurs when an implementation which supports VLE attempts to execute an instruction that has the VLE storage attribute set to 1 and the E (Endian) storage attribute set to 1 for the page that corresponds to the effective address of the instruction. If a Byte Ordering Instruction Storage Exception is detected and no higher priority exception exists, an Instruction Storage Interrupt will occur setting SRR0(GSRR0) to the address for which execution was attempted.

2.2.2 VLE Exception Syndrome Bits

Two bits in the Exception Syndrome Register (ESR) (see Section 7.2.13 of Book III-E) are provided to facilitate VLE exception handling, VLEMI and MIF.

$ESR(GESR)_{VLEMI}$ is set when an exception and subsequent interrupt is caused by the execution or attempted

execution of an instruction that resides in memory with the VLE storage attribute set.

$ESR(GESR)_{MIF}$ is set when an Instruction Storage Interrupt is caused by a Misaligned Instruction Storage Exception or when an Instruction TLB Error Interrupt was caused by a TLB miss on the second half of a misaligned 32-bit instruction.

$ESR(GESR)_{BO}$ is set when an Instruction Storage Interrupt is caused by a Mismatched Instruction Storage Exception or a Byte Ordering Instruction Storage Exception.

Programming Note

When an Instruction TLB Error Interrupt occurs as the result of a Instruction TLB miss on the second half of a 32-bit VLE instruction that is aligned to only 16-bits, SRR0 will point to the first half of the instruction and ESR_{MIF} will be set to 1. Any other status posted as a result of the TLB miss (such as MAS register updates described in Chapter 6 of Book III-E) will reflect the page corresponding to the second half of the instruction which caused the Instruction TLB miss.

Chapter 3. VLE Compatibility with Books I–III

This chapter addresses the relationship between VLE and Books I–III.

3.1 Overview

Category VLE uses the same semantics as Books I–III. Due to the limited instruction encoding formats, VLE instructions typically support reduced immediate fields and displacements, and not all operations defined by Books I–III are encoded in category VLE. The basic philosophy is to capture all useful operations, with most frequent operations given priority. Immediate fields and displacements are provided to cover the majority of ranges encountered in Embedded control code. Instructions are encoded in either a 16- or 32-bit format, and these may be freely intermixed.

VLE instructions cannot access floating-point registers (FPRs). VLE instructions use GPRs and SPRs with the following limitations:

- VLE instructions using the 16-bit formats are limited to addressing GPR0–GPR7, and GPR24–GPR31 in most instructions. Move instructions are provided to transfer register contents between these registers and GPR8–GPR23.
- VLE compare and bit test instructions using the 16-bit formats implicitly set their results in CR0.

VLE instruction encodings are generally different than instructions defined by Books I–III, except that most instructions falling within primary opcode 31 are encoded identically and have identical semantics unless they affect or access a resource not supported by category VLE.

3.2 VLE Processor and Storage Control Extensions

This section describes additional functionality to support category VLE.

3.2.1 Instruction Extensions

This section describes extensions to support VLE operations. Because instructions may reside on a half-word boundary, bit 62 is not masked by instructions that read an instruction address from a register, such as the LR, CTR, or a save/restore register 0, that holds an instruction address:

The instruction set defined by Books I-III is modified to support halfword instruction addressing, as follows:

- For *Return From Interrupt* instructions, such as ***rfi***, ***rfci***, ***rfdi***, ***rfgi***, and ***rfmci*** no longer mask bit 62 of the respective save/restore register 0. The destination address is $SRR0_{0:62}$ || 0b0, $CSRR0_{0:62}$ || 0b0, $DSRR0_{0:62}$ || 0b0, $GSRR0_{0:62}$ || 0b0, and $MCSRR0_{0:62}$ || 0b0, respectively.
- For ***bclr***, ***bclrl***, ***bcctr***, and ***bcctrl*** no longer mask bit 62 of the LR or CTR. The destination address is $LR_{0:62}$ || 0b0 or $CTR_{0:62}$ || 0b0.

3.2.2 MMU Extensions

VLE operation is indicated by the VLE storage attribute. When the VLE storage attribute for a page is set to 1, instruction fetches from that page are decoded and processed as VLE instructions. See Section 6.8.3 of Book III-E.

When instructions are executing from a page that has the VLE storage attribute set to 1, the processor is said to be in *VLE mode*.

3.3 VLE Limitations

VLE instruction fetches are valid only when performed in a Big-Endian mode. Attempting to fetch an instruction in a Little-Endian mode from a page with the VLE storage attribute set causes an Instruction Storage Byte-ordering exception.

Support for concurrent modification and execution of VLE instructions is implementation-dependent.

Chapter 4. Branch Operation Instructions

This section defines *Branch* instructions that can be executed when a processor is in VLE mode and the registers that support them.

4.1 Branch Facility Registers

The registers that support branch operations are:

- Section 4.1.1, “Condition Register (CR)”
- Section 4.1.2, “Link Register (LR)”
- Section 4.1.3, “Count Register (CTR)”

4.1.1 Condition Register (CR)

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching). The CR is more fully defined in Book I.

Category VLE uses the entire CR, but some comparison operations and all *Branch* instructions are limited to using CR0–CR3. The full Book I condition register field and logical operations are provided however.

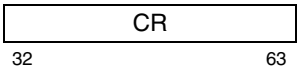


Figure 18. Condition Register

The bits in the Condition Register are grouped into eight 4-bit fields, CR Field 0 (CR0) ... CR Field 7 (CR7), which are set by VLE defined instructions in one of the following ways.

- Specified fields of the condition register can be set by a move to the CR from a GPR (***mtcrf***, ***mtocrf***).
- A specified CR field can be set by a move to the CR from another CR field (***e_mcrf***) or from $XER_{32:35}$ (***mcrxr***).
- CR field 0 can be set as the implicit result of a fixed-point instruction.
- A specified CR field can be set as the result of a fixed-point compare instruction.
- CR field 0 can be set as the result of a fixed-point bit test instruction.

Other instructions from implemented categories may also set bits in the CR in the same manner that they would when not in VLE mode.

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

For all fixed-point instructions in which the Rc bit is defined and set, and for ***e_add2i.***, ***e_and2i.***,and ***e_and2is.,*** the first three bits of CR field 0 ($CR_{32:34}$) are set by signed comparison of the result to zero, and the fourth bit of CR field 0 (CR_{35}) is copied from the final state of XER_{SO}. “Result” here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the value placed into the target register in 32-bit mode.

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if      (target_register)M:63 < 0 then c ← 0b100
else if (target_register)M:63 > 0 then c ← 0b010
else                                   c ← 0b001
CR0 ← c || XERSO
```

If any portion of the result is undefined, the value placed into the first three bits of CR field 0 is undefined.

The bits of CR field 0 are interpreted as shown below.

CR Bit	Description
32	***Negative*** (LT) The result is negative.
33	***Positive*** (GT) The result is positive.
34	***Zero*** (EQ) The result is 0.
35	***Summary overflow*** (SO) This is a copy of the contents of XER_{SO} at the completion of the instruction.

4.1.1.1 Condition Register Setting for Compare Instructions

For compare instructions, a CR field specified by the BF operand for the ***e_cmph***, ***e_cmphl***, ***e_cmpi***, and ***e_cmpli*** instructions, or CR0 for the ***se_cmpl***, ***e_cmp16i***, ***e_cmph16i***, ***e_cmphl16i***, ***e_cmpl16i***, ***se_cmp***, ***se_cmph***, ***se_cmphl***, ***se_cmpi***, and ***se_cmpli*** instructions, is set to reflect the result of the comparison. The CR field bits are interpreted as shown below. A complete description of how the bits are set is

given in the instruction descriptions and Section 5.6, “Fixed-Point Compare and Bit Test Instructions”.

Condition register bits settings for compare instructions are interpreted as follows. (Note: ***e_cmpi***, and ***e_cmpli*** instructions have a BF32 field instead of BF field; for these instructions, BF32 should be substituted for BF in the list below.)

CR Bit Description

4×BF + 32
Less Than (LT)
For signed fixed-point compare, (RA) or (RX) < sci8, SI, (RB), or (RY).
For unsigned fixed-point compare, (RA) or (RX) $<^{u}$ sci8, UI, UI5, (RB), or (RY).

4×BF + 33
Greater Than (GT)
For signed fixed-point compare, (RA) or (RX) > sci8, SI, (RB), or (RY).
For unsigned fixed-point compare, (RA) or (RX) $>^{u}$ sci8, UI, UI5, (RB), or (RY).

4×BF + 34
Equal (EQ)
For fixed-point compare, (RA) or (RX) = sci8, UI, UI5, SI, (RB), or (RY).

4×BF + 35
Summary Overflow (SO)
For fixed-point compare, this is a copy of the contents of XER_{SO} at the completion of the instruction.

4.1.1.2 Condition Register Setting for the Bit Test Instruction

The Bit Test Immediate instruction, ***se_btsti***, also sets CR field 0. See the instruction description and also Section 5.6, “Fixed-Point Compare and Bit Test Instructions”.

4.1.2 Link Register (LR)

VLE instructions use the Link Register (LR) as defined in Book I, although category VLE defines a subset of all variants of Book I conditional branches involving the LR.

4.1.3 Count Register (CTR)

VLE instructions use the Count Register (CTR) as defined in Book I, although category VLE defines a subset of the variants of Book I conditional branches involving the CTR.

4.2 Branch Instructions

The sequence of instruction execution can be changed by the branch instructions. Because VLE instructions must be aligned on half-word boundaries, the low-order bit of the generated branch target address is forced to 0 by the processor in performing the branch.

The branch instructions compute the EA of the target in one of the following ways, as described in Section 2.2, “Instruction Storage Addressing Modes”

1. Adding a displacement to the address of the branch instruction.
2. Using the address contained in the LR (Branch to Link Register [and Link]).
3. Using the address contained in the CTR (Branch to Count Register [and Link]).

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK = 1), the EA of the instruction following the branch instruction is placed into the LR after the branch target address has been computed; this is done regardless of whether the branch is taken.

In branch conditional instructions, the BI32 or BI16 instruction field specifies the CR bit to be tested. For 32-bit instructions using BI32, $CR_{32:47}$ (corresponding to bits in CR0:CR3) may be specified. For 16-bit instructions using BI16, only $CR_{32:35}$ (bits within CR0) may be specified.

In branch conditional instructions, the BO32 or BO16 field specifies the conditions under which the branch is taken and how the branch is affected by or affects the CR and CTR. Note that VLE instructions also have different encodings for the BO32 and BO16 fields than in Book I’s BO field.

If the BO32 field specifies that the CTR is to be decremented, in 64-bit mode $CTR_{0:63}$ are decremented, and in 32-bit mode $CTR_{32:63}$ are decremented. If BO16 or BO32 specifies a condition that must be TRUE or FALSE, that condition is obtained from the contents of $CR_{BI32+32}$ or $CR_{BI16+32}$. (Note that CR bits are numbered 32:63. BI32 or BI16 refers to the condition register bit field in the branch instruction encoding. For example, specifying BI32 = 2 refers to CR_{34}.)

For Figure 19 let M = 0 in 64-bit mode and M = 32 in 32-bit mode.

Encodings for the BO32 field for VLE are shown in Figure 19.

BO32	Description
00	Branch if the condition is false.
01	Branch if the condition is true.
10	Decrement $CTR_{M:63}$, then branch if the decremented $CTR_{M:63} \neq 0$
11	Decrement $CTR_{M:63}$, then branch if the decremented $CTR_{M:63} = 0$.

Figure 19. BO32 field encodings

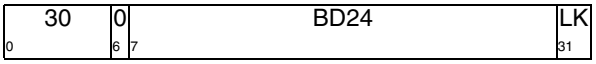
Encodings for the BO16 field for VLE are shown in Figure 20.

BO16	Description
0	Branch if the condition is false.
1	Branch if the condition is true.

Figure 20. BO16 field encodings

Branch [and Link] *BD24-form*

e_b target_addr (LK=0)
e_bl target_addr (LK=1)




```
NIA ←iea CIA + EXTS(BD24 || 0b0)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

The branch target address is the sum of BD24 || 0b0 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Branch [and Link] *BD8-form*

se_b target_addr (LK=0)
se_bl target_addr (LK=1)


```
NIA ←iea CIA + EXTS(BD8 || 0b0)
if LK then LR ←iea CIA + 2
```

target_addr specifies the branch target address.

The branch target address is the sum of BD8 || 0b0 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

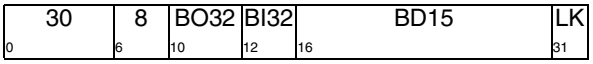
If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Branch Conditional [and Link] BD15-form

e_bc BO32,BI32,target_addr (LK=0)
e_bcl BO32,BI32,target_addr (LK=1)




```
if (64-bit mode)
  then M ← 0
  else M ← 32
if BO32_0 then CTR_M:63 ← CTR_M:63 - 1
ctr_ok  ← ¬BO32_0 | ((CTR_M:63 ≠ 0) ⊕ BO32_1)
cond_ok ← BO32_0 | (CR_BI32+32 ≡ BO32_1)
if ctr_ok & cond_ok then
  NIA ←iea (CIA + EXTS(BD15 || 0b0))
else
  NIA ←iea CIA + 4
if LK then LR ←iea CIA + 4
```

The BI32 field specifies the Condition Register bit to be tested. The BO32 field is used to resolve the branch as described in Figure 19. *target_addr* specifies the branch target address.

The branch target address is the sum of BD15 || 0b0 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

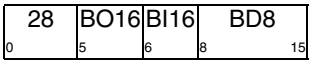
If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR (if $BO32_0$=1)
LR (if LK=1)

Branch Conditional Short Form BD8-form

se_bc BO16,BI16,target_addr




```
cond_ok ← (CR_BI16+32 ≡ BO16)
if cond_ok then
               NIA ←iea CIA + EXTS(BD8 || 0b0)
else           NIA ←iea CIA + 2
```

The BI16 field specifies the Condition Register bit to be tested. The BO16 field is used to resolve the branch as described in Figure 20. *target_addr* specifies the branch target address.

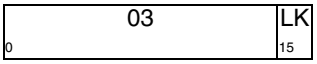
The branch target address is the sum of BD8 || 0b0 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

Special Registers Altered:

None

Branch to Count Register [and Link] C-form

se_bctr (LK=0)
se_bctrl (LK=1)




```
NIA ←iea CTR0:62 || 0b0
if LK then LR ←iea CIA + 2
```

The branch target address is $CTR_{0:62}$ || 0b0 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

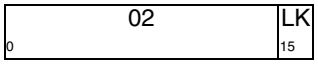
If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Branch to Link Register [and Link] C-form

se_blr (LK=0)
se_blrl (LK=1)




```
NIA ←iea LR0:62 || 0b0
if LK then LR ←iea CIA + 2
```

The branch target address is $LR_{0:62}$ || 0b0 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

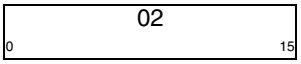
4.3 System Linkage Instructions

The *System Linkage* instructions enable the program to call upon the system to perform a service and provide a means by which the system can return from performing a service or from processing an interrupt. System Linkage instructions defined by the VLE category are identical in semantics to *System Linkage* instructions defined in Book I and Book III-E with the exception of the LEV field, but are encoded differently.

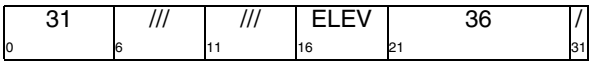
se_sc provides the same functionality as the Book I (and Book III-E) instruction ***sc*** without the LEV field. ***se_rfi***, ***se_rfci***, ***se_rfdi***, and ***se_rfmci*** provide the same functionality as the Book III-E instructions ***rfi***, ***rfci***, ***rfdi***, and ***rfmci*** respectively.

System Call *C-form,ESC-form*

se_sc



e_sc ELEV [Category:Embedded.Hypervisor]




```
lev = ELEV
if 'se_sc' then
   lev ← 0
   rr0 ←iea CIA + 2
else if 'e_sc' then
   lev ← ELEV
   rr0 ←iea CIA + 4
if lev = 0 then
   if MSRGS = 1 then
          GSRR0 ←iea rr0
          GSRR1 ← MSR
          if IVORs supported then
                 NIA ← GIVPR0:47 ||
                       GIVOR848:59 || 0b0000
          else
                 NIA ← GIVPR0:51 || 0x120
          MSR ← new_value (see below)
   else
          SRR0 ←iea rr0
          SRR1 ← MSR
          if IVORs supported then
                 NIA ← IVPR0:47 ||
                       IVOR848:59 || 0b0000
          else
                 NIA ← IVPR0:51 || 0x120
          MSR ← new_value (see below)
else if ELEV = 1 then
   SRR0 ←iea CIA + 4
   SRR1 ← MSR
   if IVORs supported then
          NIA ← IVPR0:47 || IVOR4048:59 || 0b0000
   else
          NIA ← IVPR0:51 || 0x300
   MSR ← new_value (see below)
```

If category E.HV is not implemented, the *System Call* instruction behaves as if MSR_{GS} = 0 and ELEV = 0.

If MSR_{GS} = 0 or if ELEV = 1, the effective address of the instruction following the *System Call* instruction is placed into SRR0 and the contents of the MSR are copied into SRR1. Otherwise, the effective address of the instruction following the *System Call* instruction is placed into GSRR0 and the contents of the MSR are copied into GSRR1. ELEV values greater than 1 are reserved. Bits 0:3 of the ELEV field (instruction bits 16:19) are treated as a reserved field.

If ELEV=0, a System Call interrupt is generated. If ELEV=1, an Embedded Hypervisor System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 7.6.10 and Section 7.6.30 of Book III-E.

If ELEV=0 or ***se_sc*** is executed, and the processor is in guest state, instruction execution resumes at the address given by one of the following.

- $GIVPR_{0:47}$ || $GIVOR8_{48:59}$||0b0000 if IVORs [Category:Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x120 if Interrupt Fixed Offsets [Category:Embedded.Phased-In] are supported.

If ELEV=0 or ***se_sc*** is executed, and the processor is in hypervisor state, instruction execution resumes at the address given by one of the following.

- $IVPR_{0:47}$ || $IVOR8_{48:59}$||0b0000 if IVORs [Category:Embedded.Phased-Out] are supported.
- $IVPR_{0:51}$||0x120 if Interrupt Fixed Offsets [Category:Embedded.Phased-In] are supported.

If ELEV=1, the interrupt causes instruction execution to resume at the address given by one of the following.

- $GIVPR_{0:47}$ || $GIVOR40_{48:59}$||0b0000 if IVORs [Category:Embedded.Phased-Out] are supported.
- $GIVPR_{0:51}$||0x300 if Interrupt Fixed Offsets [Category:Embedded.Phased-In] are supported.

This instruction is context synchronizing.

Special Registers Altered:

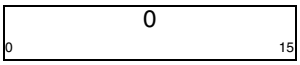
SRR0 GSRR0 SRR1 GSRR1 MSR

Programming Note

e_sc serves as both a basic and an extended mnemonic. The Assembler will recognize an ***e_sc*** mnemonic with one operand as the basic form, and an ***e_sc*** mnemonic with no operand as the extended form. In the extended form, the ELEV operand is omitted and assumed to be 0.

Illegal *C-form*

se_illegal



se_illegal is used to request an Illegal Instruction exception.

The behavior is the same as if an illegal instruction was executed.

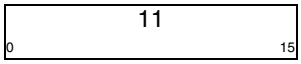
This instruction is context synchronizing.

Special Registers Altered:

SRR0 SRR1 MSR ESR

Return From Machine Check Interrupt C-form

se_rfmci




```
MSR ← MCSRR1
NIA ←iea MCSRR0 0:62 || 0b0
```

The ***se_rfmci*** instruction is used to return from a machine check class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of MCSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $MCSRR0_{0:62}$||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the values placed into the save/restore registers by the interrupt processing mechanism (see Chapter 7 of Book III-E) is the address and MSR value of the instruction that would have been executed next had the interrupt not occurred (that is, the address in MCSRR0 at the time of the execution of the ***se_rfmci***).

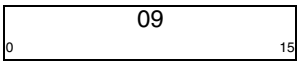
This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Return From Critical Interrupt *C-form*

se_rfci




```
MSR ← CSRR1
NIA ←iea CSRR0 0:62 || 0b0
```

The ***se_rfci*** instruction is used to return from a critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of CSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $CSRR0_{0:62}$||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the values placed into the save/restore registers by the interrupt processing mechanism (see Chapter 7 of Book III-E) is the address and MSR value of the instruction that would have been executed next had the interrupt not occurred (that is, the address in CSRR0 at the time of the execution of the ***se_rfci***).

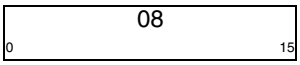
This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Return From Interrupt *C-form*

se_rfi




```
MSR ← SRR1
NIA ←iea SRR0 0:62 || 0b0
```

The ***se_rfi*** instruction is used to return from a non-critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of SRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched under control of the new MSR value from the address $SRR0_{0:62}$||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the values placed into the save/restore registers by the interrupt processing mechanism (see Chapter 7 of Book III-E) is the address and MSR value of the instruction that would have been executed next had the interrupt not occurred (that is, the address in SRR0 at the time of the execution of the ***se_rfi***).

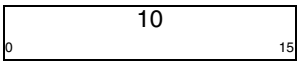
This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Return From Debug Interrupt *C-form*

se_rfdi




```
MSR ← DSRR1
NIA ←iea DSRR0[32:62] || 0b0
```

The ***se_rfdi*** instruction is used to return from a debug class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of DSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $DSRR0_{0:62}$||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the save/restore registers by the interrupt processing mechanism (see Chapter 7 of Book III-E) is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in DSRR0 at the time of the execution of ***se_rfdi***).

This instruction is privileged and context synchronizing.

Special Registers Altered:

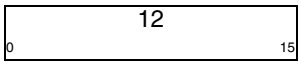
MSR

Corequisite Categories:

Embedded.Enhanced Debug

Return From Guest Interrupt *C-form*

se_rfgi i




```
newmsr ← GSRR1
if MSRGS = 1 then
     newmsrGS,WE ← MSRGS,WE
     prots ← MSRPUCLEP,DEP,PMMP
     newmsr ← prots & MSR | ~prots & newmsr
MSR ← newmsr
NIA ←iea GSRR0[0:62] || 0b0
```

The ***se_rfgi*** instruction is used to return from a guest state base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of Guest Save/Restore Register 1 are placed into the MSR. If the ***se_rfgi*** is executed in the guest supervisor state ($MSR_{GS\ PR}$ = 0b10), the bits $MSR_{GS\ WE}$ are not modified and the bits $MSR_{UCLE\ DE\ PMM}$ are modified only if the associated bits in the Machine State Register Protect (MSRP) Register are set to 0. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $GSRR0_{0:62}$||0b0. If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the associated save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in GSRR0 at the time of the execution of the ***se_rfgi***).

This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Corequisite Categories:

Embedded.Hypervisor

4.4 Condition Register Instructions

Condition Register instructions are provided to transfer values to and from various portions of the CR. Category VLE does not introduce any additional functionality beyond that defined in Book I for CR operations, but does remap the CR-logical and ***mcrf*** instruction functionality into primary opcode 31. These instructions operate identically to the Book I instructions, but are encoded differently.

Condition Register AND *XL-form*

e_crand BT,BA,BB

31	BT	BA	BB	257	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \ \& \ CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register AND with Complement XL-form

e_crandc BT,BA,BB

31	BT	BA	BB	129	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \ \& \ \neg CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ANDed with the one's complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register Equivalent *XL-form*

e_creqv BT,BA,BB

31	BT	BA	BB	289	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \equiv CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register NAND *XL-form*

e_crnand BT,BA,BB

31	BT	BA	BB	225	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow \neg (CR_{BA+32} \ \& \ CR_{BB+32})$

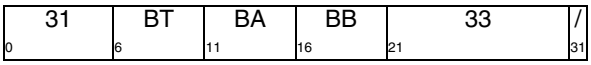
The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register NOR *XL-form*

e_crnor BT,BA,BB



$CR_{BT+32} \leftarrow \neg(CR_{BA+32} \mid CR_{BB+32})$

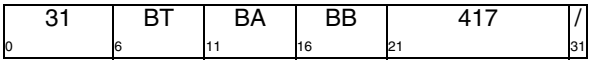
The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register OR with Complement *XL-form*

e_crorc BT,BA,BB



$CR_{BT+32} \leftarrow CR_{BA+32} \mid \neg CR_{BB+32}$

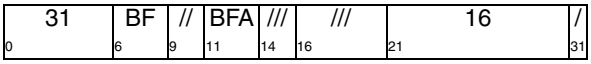
The bit in the Condition Register specified by BA+32 is ORed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Move CR Field *XL-form*

e_mcrf BF,BFA



$CR_{4xBF+32:4xBF+35} \leftarrow CR_{4xBFA+32:4xBFA+35}$

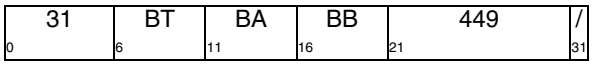
The contents of Condition Register field BFA are copied to Condition Register field BF.

Special Registers Altered:

CR field BF

Condition Register OR *XL-form*

e_cror BT,BA,BB



$CR_{BT+32} \leftarrow CR_{BA+32} \mid CR_{BB+32}$

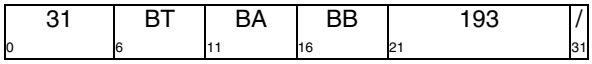
The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register XOR *XL-form*

e_crxor BT,BA,BB



$CR_{BT+32} \leftarrow CR_{BA+32} \oplus CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Chapter 5. Fixed-Point Instructions

This section lists the fixed-point instructions supported by category VLE.

5.1 Fixed-Point Load Instructions

The fixed-point *Load* instructions compute the effective address (EA) of the memory to be accessed as described in Section 2.1, “Data Storage Addressing Modes”

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into RT or RZ.

Category VLE supports both Big- and Little-Endian byte ordering for data accesses.

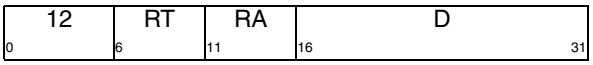
Some fixed-point load instructions have an update form in which RA is updated with the EA. For these forms, if RA≠0 and RA≠RT, the EA is placed into RA and the memory element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT. If RA=0 or RA =RT, the instruction form is invalid. This is the same behavior as specified for load with update instructions in Book I.

The fixed-point *Load* instructions from Book I, ***lbzx***, ***lbzux***, ***lhzx***, ***lhzux***, ***lhax***, ***lhaux***, ***lwzx***, and ***lwzux*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.2 of Book I for the instruction definitions.

The fixed-point *Load* instructions from Book I, ***lwax***, ***lwaux***, ***ldx***, and ***ldux*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.2 of Book Ifor the instruction definitions.

Load Byte and Zero *D-form*

e_lbz RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 56 0 || MEM(EA, 1)
```

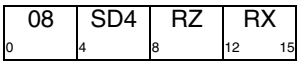
Let the effective address (EA) be the sum (RA|0) + D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Byte and Zero Short Form SD4-form

se_lbz RZ,SD4(RX)




```
EA ← (RX)+ 60 0 || SD4
RZ ← 56 0 || MEM(EA, 1)
```

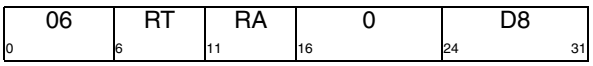
Let the effective address (EA) be the sum RX + SD4. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Byte and Zero with Update D8-form

e_lbzu RT,D8(RA)




```
EA ← (RA) + EXTS(D8)
RT ← 56 0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

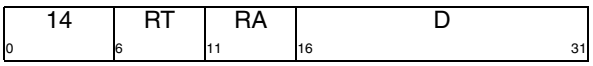
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic *D-form*

e_lha RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← EXTS(MEM(EA, 2))
```

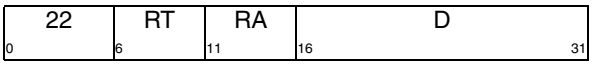
Let the effective address (EA) be the sum (RA|0) + D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Load Halfword and Zero *D-form*

e_lhz RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 48 0 || MEM(EA, 2)
```

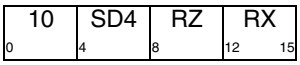
Let the effective address (EA) be the sum (RA|0) + D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero Short Form SD4-form

se_lhz RZ,SD4(RX)




```
EA ← (RX)+ (59 0 || SD4 || 0)
RZ ← 48 0 || MEM(EA, 2)
```

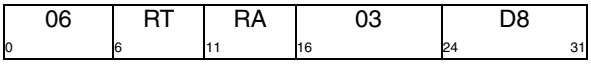
Let the effective address (EA) be the sum (RX) + (SD4 || 0). The halfword in storage addressed by EA is loaded into $RZ_{48:63}$. $RZ_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword Algebraic with Update D8-form

e_lhau RT,D8(RA)




```
EA ← (RA) + EXTS(D8)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into RA.

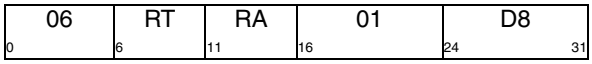
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword and Zero with Update D8-form

e_lhzu RT,D8(RA)




```
EA ← (RA) + EXTS(D8)
RT ← 48 0 || MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

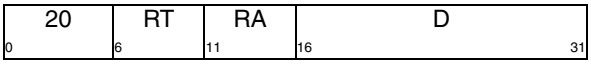
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero D-form

e_lwz RT,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
RT ← 32 0 || MEM(EA, 4)
```

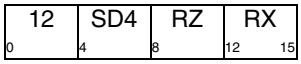
Let the effective address (EA) be the sum (RAl0) + D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero Short FormSD4-form

se_lwz RZ,SD4(RX)




```
EA ← (RX)+ (58 0 || SD4 || 2 0)
RZ ← 32 0 || MEM(EA, 2)
```

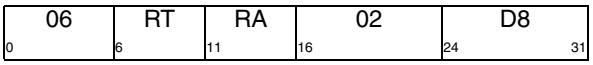
Let the effective address (EA) be the sum (RX) + (SD4 || 00). The word in storage addressed by EA is loaded into $RZ_{32:63}$. $RZ_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update D8-form

e_lwzu RT,D8(RA)




```
EA ← (RA) + EXTS(D8)
RT ← 32^0 || MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

5.2 Fixed-Point Store Instructions

The fixed-point *Store* instructions compute the EA of the memory to be accessed as described in Section 2.1, “Data Storage Addressing Modes”.

The contents of register RS or RZ are stored into the byte, halfword, word, or doubleword in storage addressed by EA.

Category VLE supports both Big- and Little-Endian byte ordering for data accesses.

Some fixed-point store instructions have an update form, in which register RA is updated with the effective address. For these forms, the following rules (from Book I) apply.

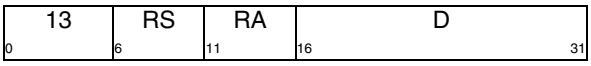
- If RA≠0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS are copied to the target memory element and then EA is placed into register RA (RS).

The fixed-point *Store* instructions from Book I, ***stbx***, ***stbux***, ***sthx***, ***sthux***, ***stwx***, and ***stwux*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.3 of Book I for the instruction definitions.

The fixed-point *Store* instructions from Book I, ***stdx*** and ***stdux*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.3 of Book I for the instruction definitions.

Store Byte *D-form*

e_stb RS,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 1) ← (RS)56:63
```

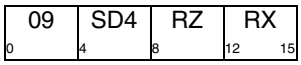
Let the effective address (EA) be the sum (RA|0)+ D. $(RS)_{56:63}$ are stored in the byte in storage addressed by EA.

Special Registers Altered:

None

Store Byte Short Form *SD4-form*

se_stb RZ,SD4(RX)




```
EA ← (RX) + EXTS(SD4)
MEM(EA, 1) ← (RZ)56:63
```

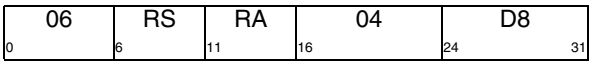
Let the effective address (EA) be the sum (RX) + SD4. $(RZ)_{56:63}$ are stored in the byte in storage addressed by EA.

Special Registers Altered:

None

Store Byte with Update *D8-form*

e_stbu RS,D8(RA)




```
EA ← (RA) + EXTS(D8)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. $(RS)_{56:63}$ are stored in the byte in storage addressed by EA.

EA is placed into register RA.

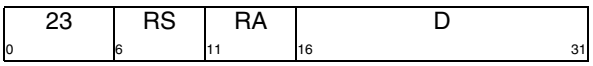
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword *D-form*

e_sth RS,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 2) ← (RS)48:63
```

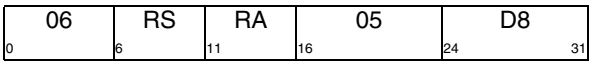
Let the effective address (EA) be the sum (RA|0) + D. $(RS)_{48:63}$ are stored in the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Halfword with Update *D8-form*

e_sthu RS,D8(RA)




```
EA ← (RA) + EXTS(D8)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. $(RS)_{48:63}$ are stored in the halfword in storage addressed by EA.

EA is placed into register RA.

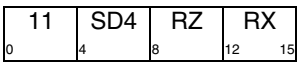
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword Short Form *SD4-form*

se_sth RZ,SD4(RX)




```
EA ← (RX) + (59 0 || SD4 || 0)
MEM(EA, 2) ← (RZ)48:63
```

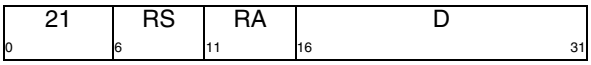
Let the effective address (EA) be the sum (RX) + (SD4 || 0). $(RZ)_{48:63}$ are stored in the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Word *D-form*

e_stw RS,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← (RS)32:63
```

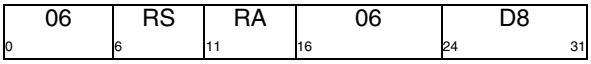
Let the effective address (EA) be the sum (RAl0) + D. $(RS)_{32:63}$ are stored in the word in storage addressed by EA.

Special Registers Altered:

None

Store Word with Update *D8-form*

e_stwu RS,D8(RA)




```
EA ← (RA) + EXTS(D8)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA) + D8. $(RS)_{32:63}$ are stored in the word in storage addressed by EA.

EA is placed into register RA.

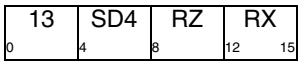
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word Short Form *SD4-form*

se_stw RZ,SD4(RX)




```
EA ← (RX) + (58 0 || SD4 || 2 0)
MEM(EA, 4) ← (RZ)32:63
```

Let the effective address (EA) be the sum (RX)+ (SD4 || 00). $(RZ)_{32:63}$ are stored in the word in storage addressed by EA.

Special Registers Altered:

None

5.3 Fixed-Point Load and Store with Byte Reversal Instructions

The fixed-point *Load with Byte Reversal* and *Store with Byte Reversal* instructions from Book I, ***lhbrx***, ***lwbrx***, ***sthbrx***, and ***stwbrx*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I. See Section 3.3.5 of Book I for the instruction definitions.

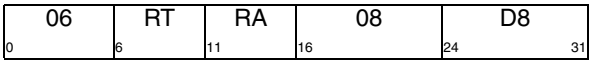
5.4 Fixed-Point Load and Store Multiple Instructions

The *Load/Store Multiple* instructions have preferred forms; see Section 1.8.1 of Book I. In the preferred forms storage alignment satisfies the following rule.

- The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned quadword in storage.

Load Multiple Word *D8-form*

e_lmw RT,D8(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D8)
r ← RT
do while r ≤31
   GPR(r) ← 32 0 || MEM(EA,4)
   r  ← r + 1
   EA ← EA + 4
```

Let n = (32-RT). Let the effective address (EA) be the sum (RA|0) + D8.

n consecutive words starting at EA are loaded into the low-order 32 bits of GPRs RT through 31. The high-order 32 bits of these GPRs are set to zero.

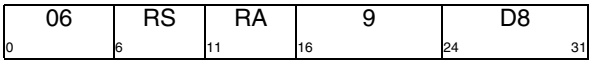
If RA is in the range of registers to be loaded, including the case in which RA = 0, the instruction form is invalid.

Special Registers Altered:

None

Store Multiple Word *D8-form*

e_stmw RS,D8(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D8)
r ← RS
do while r ≤31
   MEM(EA,4) ← GPR(r)32:63
   r  ← r + 1
   EA ← EA + 4
```

Let n = (32-RS). Let the effective address (EA) be the sum (RA|0) + D8.

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs RS through 31.

Special Registers Altered:

None

5.5 Fixed-Point Arithmetic Instructions

The fixed-point *Arithmetic* instructions use the contents of the GPRs as source operands, and place results into GPRs, into status bits in the XER and into CR0.

The fixed-point *Arithmetic* instructions treat source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The ***e_add2i.*** instruction and other *Arithmetic* instructions with Rc=1 set the first three bits of CR0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to 0. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

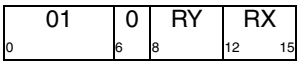
e_addic[.] and ***e_subfic***[.] always set CA to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode.

The fixed-point *Arithmetic* instructions from Book I, ***add[.], addo[.]***, ***addc[.]***, ***addco[.]***, ***adde[.]***, ***addeo[.]***, ***addme[.]***, ***addmeo[.]***, ***addze[.]***, ***addzeo[.]***, ***divw[.]***, ***divwo[.]***, ***divwu[.]***, ***divwuo[.]***, ***mulhw[.]***, ***mulhwu[.]***, ***mullw[.]***, ***mullwo[.]*** ***neg[.]***, ***nego[.]***, ***subf[.]***, ***subfo[.]*** ***subfe[.]***, ***subfeo[.]***, ***subfme[.]***, ***subfmeo[.]***, ***subfze[.]***, ***subfzeo[.]***, ***subfc[.]***, and ***subfco[.]*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.9 of Book I for the instruction definitions.

The fixed-point *Arithmetic* instructions from Book I, ***mulld[.]***, ***mulldo[.]***, ***mulhd[.]***, ***mulhdu[.]***, ***muldu[.]***, ***divd[.]***, ***divdo[.]***, ***divdu[.]***, and ***divduo[.]*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to these in Book I; see Section 3.3.9 of Book I for the instruction definitions.

Add Short Form *RR-form*

se_add RX,RY



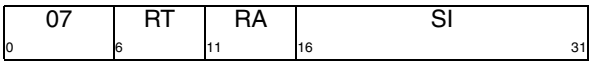

```
RX ← (RX) + (RY)
```

The sum (RX) + (RY) is placed into register RX.

Special Registers Altered:
None

Add Immediate *D-form*

e_add16i RT,RA,SI



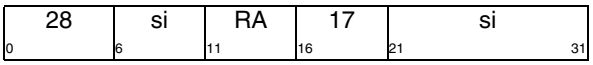

```
RT ← (RA) + EXTS(SI)
```

The sum (RA) + SI is placed into register RT.

Special Registers Altered:
None

Add (2 operand) Immediate and Record *I16A-form*

e_add2i. RA,si



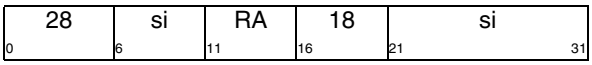

```
RA ← (RA) + EXTS(si)
```

The sum (RA) + si is placed into register RT.

Special Registers Altered:
CR0

Add (2 operand) Immediate Shifted *I16A-form*

e_add2is RA,si



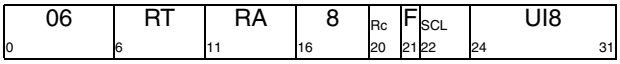

```
RA ← (RA) + EXTS(si || 16 0)
```

The sum (RA) + (si || 0x0000) is placed into register RA.

Special Registers Altered:
None

Add Scaled Immediate *SCI8-form*

e_addi RT,RA,sci8 (Rc=0)
e_addi. RT,RA,sci8 (Rc=1)



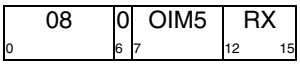

```
sci8 ← 56-SCLx8 F || UI8 || SCLx8 F
RT ← (RA) + sci8
```

The sum (RA) + sci8 is placed into register RT.

Special Registers Altered:
CR0 (if Rc=1)

Add Immediate Short Form *OIM5-form*

se_addi RX,oimm



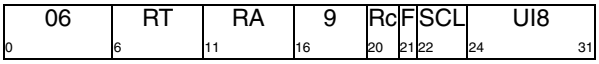

```
oimm ← (59 0 || OIM5) + 1
RX ← (RX) + oimm
```

The sum (RX) + oimm is placed into RX. The value of oimm must be in the range of 1 to 32.

Special Registers Altered:
None

Add Scaled Immediate Carrying *SCI8-form*

e_addic RT,RA,sci8 (Rc=0)
e_addic. RT,RA,sci8 (Rc=1)



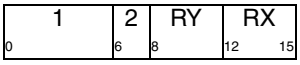

```
sci8 ← 56-SCL×8F || UI8 ||SCL×8F
RT ← (RA) + sci8
```

The sum (RA) + sci8 is placed into register RT.

Special Registers Altered:
CR0 (if Rc=1)
CA

Subtract *RR-form*

se_sub RX,RY



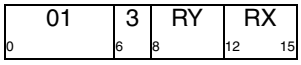

```
RX ← (RX) + ¬(RY) + 1
```

The sum (RX) + ¬(RY) + 1 is placed into register RX.

Special Registers Altered:
None

Subtract From Short Form *RR-form*

se_subf RX,RY



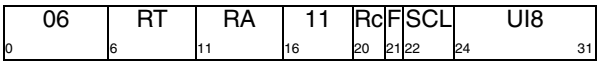

```
RX ← ¬(RX) + (RY) + 1
```

The sum ¬(RX) + (RY) + 1 is placed into register RX.

Special Registers Altered:
None

Subtract From Scaled Immediate Carrying *SCI8-form*

e_subfic RT,RA,sci8 (Rc=0)
e_subfic. RT,RA,sci8 (Rc=1)



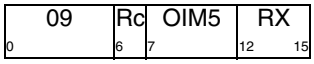

```
sci8 ← 56-SCL×8F || UI8 ||SCL×8F
RT ← ¬(RA) + sci8 + 1
```

The sum ¬(RA) + sci8 + 1 is placed into register RT.

Special Registers Altered:
CR0 (if Rc=1)
CA

Subtract Immediate *OIM5-form*

se_subi RX,oimm (Rc=0)
se_subi. RX,oimm (Rc=1)



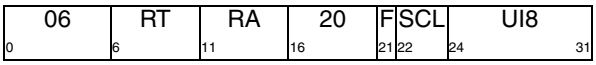

```
oimm ← (590 || OIM5) + 1
RX ← (RX) + ¬oimm + 1
```

The sum (RA) + ¬oimm + 1 is placed into register RX. The value of oimm must be in the range 1 to 32.

Special Registers Altered:
CR0 (if Rc=1)

Multiply Low Scaled Immediate SCI8-form

e_mulli RT,RA,sci8



$sci8 \leftarrow {}^{56-SCL \times 8}F \,||\, UI8 \,||^{SCL \times 8}F$
$prod_{0:127} \leftarrow (RA) \times sci8$
$RT \leftarrow prod_{64:127}$

The 64-bit first operand is (RA). The 64-bit second operand is the sci8 operand. The low-order 64-bits of the 128-bit product of the operands are placed into register RT.

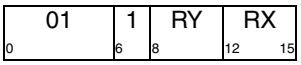
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

None

Multiply Low Word Short Form RR-form

se_mullw RX,RY



$RX \leftarrow (RX)_{32:63} \times (RY)_{32:63}$

The 32-bit operands are the low-order 32-bits of RX and of RY. The 64-bit product of the operands is placed into register RX.

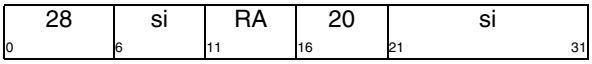
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

None

Multiply (2 operand) Low Immediate I16A-form

e_mull2i RA,si



$prod_{0:127} \leftarrow (RA) \times EXTS(si)$
$RA \leftarrow prod_{64:127}$

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the si operand. The low-order 64-bits of the 128-bit product of the operands are placed into register RA.

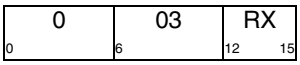
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

None

Negate Short Form R-form

se_neg RX



$RX \leftarrow \neg(RX) + 1$

The sum ¬(RX) + 1 is placed into register RX

If the processor is in 64-bit mode and register RX contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative 64-bit number. Similarly, if the processor is in 32-bit mode and register RX contains the most negative 32-bit number (0x8000_0000), the result is the most negative 32-bit number.

Special Registers Altered:

None

5.6 Fixed-Point Compare and Bit Test Instructions

The fixed-point *Compare* instructions compare the contents of register RA or register RX with one of the following:

- The value of the scaled immediate field *sci8* formed from the F, UI8, and SCL fields as:
  sci8 ← $^{56-SCL\times8}F$ || UI8 || $^{SCL\times8}F$
- The zero-extended value of the UI field
- The zero-extended value of the UI5 field
- The sign-extended value of the SI field
- The contents of register RB or register RY.

The following comparisons are signed: ***e_cmph***, ***e_cmpi***, ***e_cmp16i***, ***e_cmph16i***, ***se_cmp***, ***se_cmph***, and ***se_cmpi***.

The following comparisons are unsigned: ***e_cmphl***, ***e_cmpli***, ***e_cmphl16i***, ***e_cmpl16i***, ***se_cmpli***, ***se_cmpl***, and ***se_cmphl***.

The fixed-point *Bit Test* instruction tests the bit specified by the UI5 instruction field and sets the CR0 field as follows.

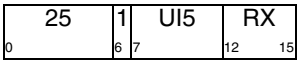
.

Bit	Name	Description
0	LT	Always set to 0
1	GT	$RX_{ui5} = 1$
2	EQ	$RX_{ui5} = 0$
3	SO	Summary overflow from the XER

The fixed-point *Compare* instructions from Book I, ***cmp*** and ***cmpl*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.10 of Book I for the instruction definitions.

Bit Test Immediate *IM5-form*

se_btsti RX,UI5




```
a ← UI5
b ← a+32 0 || 1 || 31-a 0
c ← (RX) & b
if c = 0 then d ← 0b001 else d ← 0b010
CR0 ← d || XER_SO
```

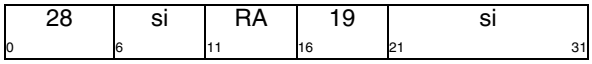
Bit UI5+32 of register RX is tested for equality to '0' and the result is recorded in CR0. EQ is set if the tested bit is 0, LT is cleared, and GT is set to the inverse value of EQ.

Special Registers Altered:

CR0

Compare Immediate Word *I16A-form*

e_cmp16i RA,si




```
b ← EXTS(si)
if (RA)32:63 < b32:63 then c ← 0b100
if (RA)32:63 > b32:63 then c ← 0b010
if (RA)32:63 = b32:63 then c ← 0b001
CR0 ← c || XER_SO
```

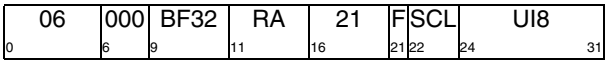
The low-order 32 bits of register RA are compared with si, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Scaled Immediate Word *SCI8-form*

e_cmpi BF32,RA,sci8




```
sci8 ← 56-SCL×8^F || UI8 ||SCL×8^F
if (RA)32:63 < sci832:63 then c ← 0b100
if (RA)32:63 > sci832:63 then c ← 0b010
if (RA)32:63 = sci832:63 then c ← 0b001
CR4×BF32+32:4×BF32+35 ← c || XERSO
```

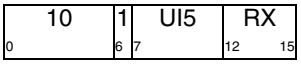
The low-order 32 bits of register RA are compared with sci8, treating operands as signed integers. The result of the comparison is placed into CR field BF32.

Special Registers Altered:

CR field BF32

Compare Immediate Word Short Form *IM5-form*

se_cmpi RX,UI5




```
b ← 59^0 || UI5
if (RX)32:63 < b32:63 then c ← 0b100
if (RX)32:63 > b32:63 then c ← 0b010
if (RX)32:63 = b32:63 then c ← 0b001
CR0 ← c || XERSO
```

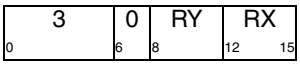
The low-order 32 bits of register RX are compared with UI5, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Word *RR-form*

se_cmp RX,RY




```
if (RX)32:63 < (RY)32:63 then c ← 0b100
if (RX)32:63 > (RY)32:63 then c ← 0b010
if (RX)32:63 = (RY)32:63 then c ← 0b001
CR0 ← c || XERSO
```

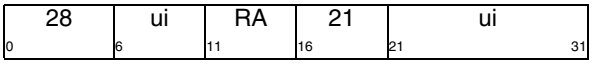
The low-order 32 bits of register RX are compared with the low-order 32 bits of register RY, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Logical Immediate Word *I16A-form*

e_cmpl16i RA,ui




```
b ← 48^0 || ui
if (RA)32:63 <u b32:63 then c ← 0b100
if (RA)32:63 >u b32:63 then c ← 0b010
if (RA)32:63 = b32:63 then c ← 0b001
CR0 ← c || XERSO
```

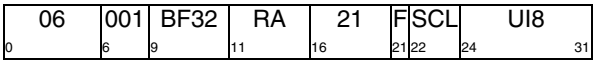
The low-order 32 bits of register RA are compared with ui, treating operands as unsigned integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Logical Scaled Immediate Word SCI8-form

e_cmpli BF32,RA,sci8




```
sci8 ← 56-SCL×8F || UI8 ||SCL×8F
if (RA)32:63 <u sci832:63 then c ← 0b100
if (RA)32:63 >u sci832:63 then c ← 0b010
if (RA)32:63 = sci832:63 then c ← 0b001
CR4×BF32+32:4×BF32+35 ← c || XERSO
```

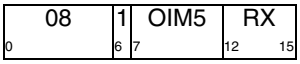
The low-order 32 bits of register RA are compared with sci8, treating operands as unsigned integers. The result of the comparison is placed into CR field BF32.

Special Registers Altered:

CR field BF32

Compare Logical Immediate Word OIM5-form

se_cmpli RX,oimm




```
oimm ← 590 || (OIM5 + 1)
if (RX)32:63 <u oimm32:63 then c ← 0b100
if (RX)32:63 >u oimm32:63 then c ← 0b010
if (RX)32:63 = oimm32:63 then c ← 0b001
CR0 ← c || XERSO
```

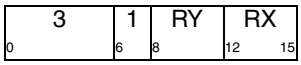
The low-order 32 bits of register RX are compared with oimm, treating operands as unsigned integers. The result of the comparison is placed into CR0. The value of oimm must be in the range of 1 to 32.

Special Registers Altered:

CR0

Compare Logical Word RR-form

se_cmpl RX,RY




```
if (RX)32:63 <u (RY)32:63 then c ← 0b100
if (RX)32:63 >u (RY)32:63 then c ← 0b010
if (RX)32:63 = (RY)32:63 then c ← 0b001
CR0 ← c || XERSO
```

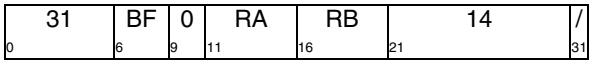
The low-order 32 bits of register RX are compared with the low-order 32 bits of register RY, treating operands as unsigned integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Halfword X-form

e_cmph BF,RA,RB




```
a ← EXTS((RA)48:63)
b ← EXTS((RB)48:63)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

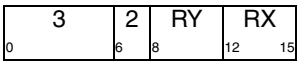
The low-order 16 bits of register RA are compared with the low-order 16 bits of register RB, treating operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Compare Halfword Short Form RR-form

se_cmph RX,RY




```
a ← EXTS((RX)48:63)
b ← EXTS((RY)48:63)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XERSO
```

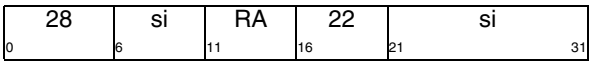
The low-order 16 bits of register RX are compared with the low-order 16 bits of register RY, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Halfword Immediate I16A-form

e_cmph16i RA,si




```
a ← EXTS((RA)48:63)
b ← EXTS(si)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XERSO
```

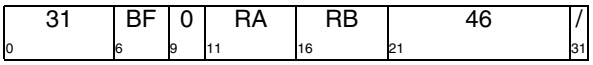
The low-order 16 bits of register RA are compared with si, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Halfword Logical X-form

e_cmphl BF,RA,RB




```
a ← EXTZ((RA)48:63)
b ← EXTZ((RB)48:63)
if a <u b then c ← 0b100
if a >u b then c ← 0b010
if a = b then c ← 0b001
CR4xBF+32:4xBF+35 ← c || XERSO
```

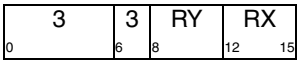
The low-order 16 bits of register RA are compared with the low-order 16 bits of register RB, treating operands as unsigned integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Compare Halfword Logical Short Form RR-form

se_cmphl RX,RY




```
a ← (RX)48:63
b ← (RY)48:63
if a <u b then c ← 0b100
if a >u b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XERSO
```

The low-order 16 bits of register RX are compared with the low-order 16 bits of register RY, treating operands as unsigned integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

Compare Halfword Logical Immediate I16A-form

e_cmphl16i RA,ui

28	ui	RA	23	ui
0	6	11	16	21 31

```
a ← 48 0 || (RA)48:63
b ← 48 0 || ui
if a <u b then c ← 0b100
if a >u b then c ← 0b010
if a = b then c ← 0b001
CR0 ← c || XERSO
```

The low-order 16 bits of register RA are compared with the ui field, treating operands as signed integers. The result of the comparison is placed into CR0.

Special Registers Altered:

CR0

5.7 Fixed-Point Trap Instructions

The fixed-point *Trap* instruction from Book I, ***tw*** is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction is identical to that in Book I; see Section 3.3.11 of Book I for the instruction definition.

The fixed-point *Trap* instruction from Book I, ***td*** is available while executing in VLE mode on 64-bit implementations. The mnemonic, decoding, and semantics for the ***td*** instruction are identical to those in Book I; see Section 3.3.11 of Book I for the instruction definitions.

5.8 Fixed-Point Select Instruction

The fixed-point *Select* instruction provides a means to select one of two registers and place the result in a destination register under the control of a predicate value supplied by a CR bit.

The fixed-point *Select* instruction from Book I, ***isel*** is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction is identical to that in Book I; see Section of Book I for the instruction definition.

5.9 Fixed-Point Logical, Bit, and Move Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands. The *Bit* instructions manipulate a bit, or create a bit mask, in a register. The *Move* instructions move a register or an immediate value into a register.

The X-form Logical instructions with Rc=1, the SCI8-form Logical instructions with Rc=1, the RR-form Logical instructions with Rc=1, the ***e_and2i.*** instruction, and the ***e_and2is.*** instruction set the first three bits of CR field 0 as the arithmetic instructions described in Section 5.5, “Fixed-Point Arithmetic Instructions”. (Also see Section 4.1.1.) The Logical instructions do not change the SO, OV, and CA bits in the XER.

The fixed-point *Logical* instructions from Book I, ***and[.]***, ***or[.]***, ***xor[.]***, ***nand[.]***, ***nor[.]***, ***eqv[.]***, ***andc[.]***, ***orc[.]***, ***extsb[.]***, ***extsh[.]***, ***cntlzw[.]***, and ***popcntb*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.13 of Book I for the instruction definitions.

The fixed-point *Logical* instructions from Book I, ***extsw[.]*** and ***cntlzd[.]*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.13 of Book I for the instruction definitions.

AND (two operand) Immediate I16L-form

e_and2i. RT,ui

28	RT	ui	25	ui
0	6	11	16	21 31

`RT ← (RT) & (`^{48}`0 || ui)`

The contents of register RT are ANDed with $^{48}0$ || ui and the result is placed into register RT.

Special Registers Altered:

CR0

AND (2 operand) Immediate Shifted I16L-form

e_and2is. RT,ui

28	RT	ui	29	ui
0	6	11	16	21 31

`RT ← (RT) & (`^{32}`0 || ui ||` ^{16}`0)`

The contents of register RT are ANDed with $^{32}0$ || ui || $^{16}0$ and the result is placed into register RT.

Special Registers Altered:

CR0

AND Scaled Immediate SCI8-form

e_andi RA,RS,sci8 (Rc=0)
e_andi. RA,RS,sci8 (Rc=1)

06	RS	RA	12	Rc	F	SCL	UI8
0	6	11	16	20	21	22 24	31

`sci8 ←` $^{56-SCL\times8}$`F || UI8 ||` $^{SCL\times8}$`F`
`RA ← (RS) & sci8`

The contents of register RS are ANDed with sci8 and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

AND Immediate Short Form IM5-form

se_andi RX,UI5

11	1	UI5	RX
0	6	7 12	15

`RX ← (RX) &` ^{59}`0 || UI5`

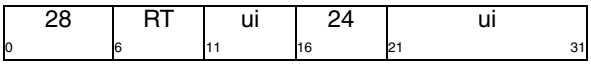
The contents of register RX are ANDed with $^{59}0$ || UI5 and the result is placed into register RX.

Special Registers Altered:

None

OR (two operand) Immediate I16L-form

e_or2i RT,ui




```
RT ← (RT) | (48 0 || ui)
```

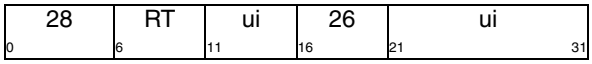
The contents of register RT are ORed with $^{48}0$ || ui and the result is placed into register RT.

Special Registers Altered:

None

OR (2 operand) Immediate Shifted I16L-form

e_or2is RT,ui




```
RT ← (RT) | (32 0 || ui || 16 0)
```

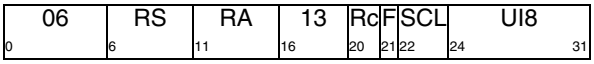
The contents of register RT are ORed with $^{32}0$ || ui || $^{16}0$ and the result is placed into register RT.

Special Registers Altered:

None

OR Scaled Immediate SCI8-form

e_ori RA,RS,sci8 (Rc=0)
e_ori. RA,RS,sci8 (Rc=1)




```
sci8 ← 56-SCL×8 F || UI8 || SCL×8 F
RA ← (RS) | sci8
```

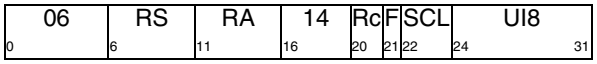
The contents of register RS are ORed with sci8 and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

XOR Scaled Immediate SCI8-form

e_xori RA,RS,sci8 (Rc=0)
e_xori. RA,RS,sci8 (Rc=1)




```
sci8 ← 56-SCL×8 F || UI8 || SCL×8 F
RA ← (RS) ⊕ sci8
```

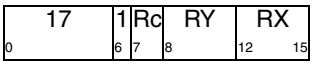
The contents of register RS are XORed with sci8 and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

AND Short Form RR-form

se_and RX,RY (Rc=0)
se_and. RX,RY (Rc=1)




```
RX ← (RX) & (RY)
```

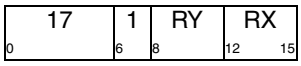
The contents of register RX are ANDed with the contents of register RY and the result is placed into register RX.

Special Registers Altered:

CR0 (if Rc=1)

AND with Complement Short Form RR-form

se_andc RX,RY




```
RX ← (RX) & ¬(RY)
```

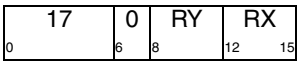
The contents of register RX are ANDed with the complement of the contents of register RY and the result is placed into register RX.

Special Registers Altered:

None

OR Short Form *RR-form*

se_or RX,RY




```
RX ← (RX) | (RY)
```

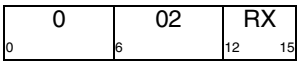
The contents of register RX are ORed with the contents of register RY and the result is placed into register RX.

Special Registers Altered:

None

NOT Short Form *R-form*

se_not RX




```
RX ← ¬(RX)
```

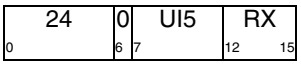
The contents of RX are complemented and placed into register RX.

Special Registers Altered:

None

Bit Clear Immediate *IM5-form*

se_bclri RX,UI5




```
a  ← UI5
RX ← (RX) & (a+32 1 || 0 || 31-a 1)
```

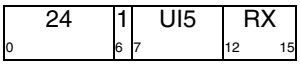
Bit UI5+32 of register RX is set to 0.

Special Registers Altered:

None

Bit Generate Immediate *IM5-form*

se_bgeni RX,UI5




```
a  ← UI5
RX ← (a+32 0 || 1 || 31-a 0)
```

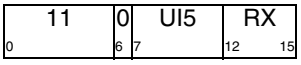
Bit UI5+32 of register RX is set to 1. All other bits in register RX are set to 0.

Special Registers Altered:

None

Bit Mask Generate Immediate *IM5-form*

se_bmaski RX,UI5




```
a ← UI5
if a = 0 then RX ← 64 1
else          RX ← 64-a 0 || a 1
```

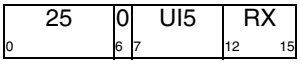
If UI5 is not zero, the low-order UI5 bits are set to 1 in register RX and all other bits in register RX are set to 0. If UI5 is 0, all bits in register RX are set to 1.

Special Registers Altered:

None

Bit Set Immediate *IM5-form*

se_bseti RX,UI5




```
a  ← UI5
RX ← (RX) | (a+32 0 || 1 || 31-a 0)
```

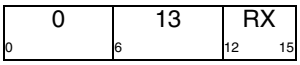
Bit UI5+32 of register RX is set to 1.

Special Registers Altered:

None

Extend Sign Byte Short Form R-form

se_extsb RX



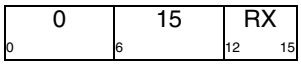
$s \leftarrow (RX)_{56}$
$RX \leftarrow {}^{56}s \parallel (RX)_{56:63}$

$(RX)_{56:63}$ are placed into $RX_{56:63}$. Bit 56 of register RX is placed into $RX_{0:55}$.

Special Registers Altered:
None

Extend Sign Halfword Short Form R-form

se_extsh RX



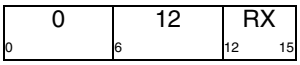
$s \leftarrow (RX)_{48}$
$RX \leftarrow {}^{48}s \parallel (RX)_{48:63}$

$(RX)_{48:63}$ are placed into $RX_{48:63}$. Bit 48 of register RX is placed into $RX_{0:47}$.

Special Registers Altered:
None

Extend Zero Byte R-form

se_extzb RX



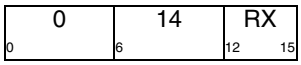
$RX \leftarrow {}^{56}0 \parallel (RX)_{56:63}$

$(RX)_{56:63}$ are placed into $RX_{56:63}$. $RX_{0:55}$ are set to 0.

Special Registers Altered:
None

Extend Zero Halfword R-form

se_extzh RX



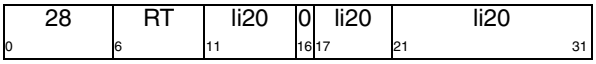
$RX \leftarrow {}^{48}0 \parallel (RX)_{48:63}$

$(RX)_{48:63}$ are placed into $RX_{48:63}$. $RX_{0:47}$ are set to 0.

Special Registers Altered:
None

Load Immediate LI20-form

e_li RT,LI20



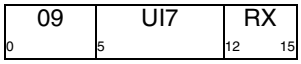
$RT \leftarrow EXTS(li20_{1:4} \parallel li20_{5:8} \parallel li20_{0} \parallel li20_{9:19})$

The sign-extended LI20 field is placed into RT.

Special Registers Altered:
None

Load Immediate Short Form IM7-form

se_li RX,UI7



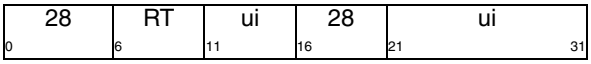
$RX \leftarrow {}^{57}0 \parallel UI7$

The zero-extended UI7 field is placed into RX.

Special Registers Altered:
None

Load Immediate Shifted I16L-form

e_lis RT,ui



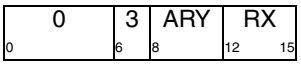
$RT \leftarrow {}^{32}0 \parallel ui \parallel {}^{16}0$

The zero-extended value of ui shifted left 16 bits is placed into RT.

Special Registers Altered:
None

Move from Alternate Register *RR-form*

se_mfar RX,ARY




```
r ← ARY+8
RX ← GPR(r)
```

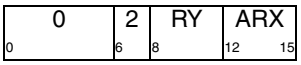
The contents of register ARY+8 are placed into RX. ARY specifies a register in the range R8:R23.

Special Registers Altered:

None

Move To Alternate Register *RR-form*

se_mtar ARX,RY




```
r ← ARX+8
GPR(r) ← (RY)
```

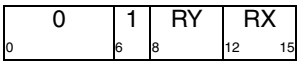
The contents of register RY are placed into register ARX+8. ARX specifies a register in the range R8:R23.

Special Registers Altered:

None

Move Register *RR-form*

se_mr RX,RY




```
RX ← (RY)
```

The contents of register RY are placed into RX.

Special Registers Altered:

None

5.10 Fixed-Point Rotate and Shift Instructions

The fixed-point *Shift* instructions from Book I, ***slw[.]***, ***srw[.]***, ***srawi[.]***, and ***sraw[.]*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.14.2 of Book I for the instruction definitions.

The fixed-point *Shift* instructions from Book I, ***sld[.]***, ***srd[.]***, ***sradi[.]***, and ***srad[.]*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.14.2 of Book I for the instruction definitions.

Rotate Left Word *X-form*

e_rlw RA,RS,RB (Rc=0)
e_rlw. RA,RS,RB (Rc=1)

31	RS	RA	RB	280	Rc
0	6	11	16	21	31

$n \leftarrow (RB)_{59:63}$
$RA \leftarrow ROTL_{32}((RS)_{32:63}, n)$

The contents of register RS are rotated$_{32}$ left the number of bits specified by $(RB)_{59:63}$ and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Rotate Left Word Immediate *X-form*

e_rlwi RA,RS,SH (Rc=0)
e_rlwi. RA,RS,SH (Rc=1)

31	RS	RA	SH	312	Rc
0	6	11	16	21	31

$n \leftarrow SH$
$RA \leftarrow ROTL_{32}((RS)_{32:63}, n)$

The contents of register RS are rotated$_{32}$ left SH bits and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Rotate Left Word Immediate then Mask Insert *M-form*

e_rlwimi RA,RS,SH,MB,ME

29	RS	RA	SH	MB	ME	0
0	6	11	16	21	26	31

$n \leftarrow SH$
$r \leftarrow ROTL_{32}((RS)_{32:63}, n)$
$m \leftarrow MASK(MB+32, ME+32)$
$RA \leftarrow r\&m \mid (RA)\&\neg m$

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data is inserted into register RA under control of the generated mask.

Special Registers Altered:
None

Rotate Left Word Immediate then AND with Mask *M-form*

e_rlwinm RA,RS,SH,MB,ME

29	RS	RA	SH	MB	ME	1
0	6	11	16	21	26	31

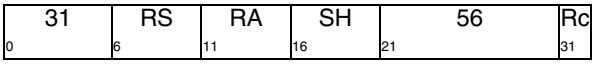
$n \leftarrow SH$
$r \leftarrow ROTL_{32}((RS)_{32:63}, n)$
$m \leftarrow MASK(MB+32, ME+32)$
$RA \leftarrow r \,\&\, m$

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:
None

Shift Left Word Immediate *X-form*

e_slwi RA,RS,SH (Rc=0)
e_slwi. RA,RS,SH (Rc=1)




```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(32, 63-n)
RA ← r & m
```

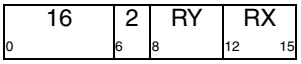
The contents of the low-order 32 bits of register RS are shifted left SH bits. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to 0.

Special Registers Altered:

CR0 (if Rc=1)

Shift Left Word *RR-form*

se_slw RX,RY




```
n ← (RY)58:63
r ← ROTL32((RX)32:63, n)
if (RY)58 = 0 then m ← MASK(32, 63-n)
else               m ← 64 0
RX ← r & m
```

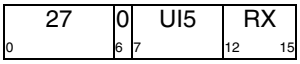
The contents of the low-order 32 bits of register RX are shifted left the number of bits specified by $(RY)_{58:63}$. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RX_{32:63}$. $RX_{0:31}$ are set to 0. Shift amounts from 32-63 give a zero result.

Special Registers Altered:

None

Shift Left Word Immediate Short Form *IM5-form*

se_slwi RX,UI5




```
n ← UI5
r ← ROTL32((RX)32:63, n)
m ← MASK(32, 63-n)
RX ← r & m
```

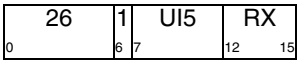
The contents of the low-order 32 bits of register RX are shifted left UI5 bits. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RX_{32:63}$. $RX_{0:31}$ are set to 0.

Special Registers Altered:

None

Shift Right Algebraic Word Immediate *IM5-form*

se_srawi RX,UI5




```
n ← UI5
r ← ROTL32((RX)32:63, 64-n)
m ← MASK(n+32, 63)
s ← (RX)32
RX ← r&m | (64s)&¬m
CA ← s & ((r&¬m)32:63≠0)
```

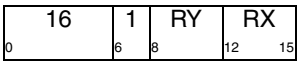
The contents of the low-order 32 bits of register RX are shifted right UI5 bits. Bits shifted out of position 63 are lost, and bit 32 of RX is replicated to fill the vacated positions on the left. Bit 32 of RX is replicated to fill $RX_{0:31}$ and the 32-bit result is placed into $RX_{32:63}$. CA is set to 1 if the low-order 32 bits of register RX contain a negative value and any 1-bits are shifted out of bit position 63; otherwise CA is set to 0. A shift amount of zero causes RX to receive $EXTS((RX)_{32:63})$, and CA to be set to 0.

Special Registers Altered:

CA

Shift Right Algebraic Word *RR-form*

se_sraw RX,RY




```
n ← (RY)59:63
r ← ROTL32((RX)32:63, 64-n)
if (RY)58 = 0 then m ← MASK(n+32, 63)
else               m ← 640
s ← (RX)32
RX ← r&m | (64s)&¬m
CA ← s & ((r&¬m)32:63≠0)
```

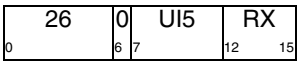
The contents of the low-order 32 bits of register RX are shifted right the number of bits specified by $(RY)_{58:63}$. Bits shifted out of position 63 are lost, and bit 32 of RX is replicated to fill the vacated positions on the left. Bit 32 of RX is replicated to fill $RX_{0:31}$ and the 32-bit result is placed into $RX_{32:63}$. CA is set to 1 if the low-order 32 bits of register RX contain a negative value and any 1-bits are shifted out of bit position 63; otherwise CA is set to 0. A shift amount of zero causes RX to receive $EXTS((RX)_{32:63})$, and CA to be set to 0. Shift amounts from 32-63 give a result of 64 sign bits, and cause CA to receive the sign bit of $(RX)_{32:63}$.

Special Registers Altered:

CA

Shift Right Word Immediate Short Form *IM5-form*

se_srwi RX,UI5




```
n ← UI5
r ← ROTL32((RX)32:63, 64-n)
m ← MASK(n+32, 63)
RX ← r & m
```

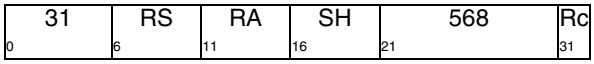
The contents of the low-order 32 bits of register RX are shifted right UI5 bits. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RX_{32:63}$. $RX_{0:31}$ are set to 0.

Special Registers Altered:

None

Shift Right Word Immediate *X-form*

e_srwi RA,RS,SH (Rc=0)
e_srwi. RA,RS,SH (Rc=1)




```
n ← SH
r ← ROTL32((RS)32:63, 64-n)
m ← MASK(n+32, 63)
RA ← r & m
```

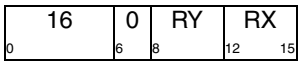
The contents of the low-order 32 bits of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to 0.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Word *RR-form*

se_srw RX,RY




```
n ← (RY)59:63
r ← ROTL32((RX)32:63, 64-n)
if (RY)58 = 0 then m ← MASK(n+32, 63)
else               m ← 640
RX ← r & m
```

The contents of the low-order 32 bits of register RX are shifted right the number of bits specified by $(RY)_{58:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RX_{32:63}$. $RX_{0:31}$ are set to 0. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

None

5.11 Move To/From System Register Instructions

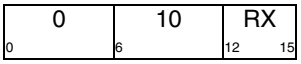
The VLE category provides 16-bit forms of instructions to move to/from the LR and CTR.

The fixed-point *Move To/From System Register* instructions from Book I, ***mfspr***, ***mtcrf***, ***mfcr***, ***mfdcrx***, ***mtocrf***, ***mfocrf***, ***mcrxr***, ***mtdcrx***, ***mtdcrux***, ***mfdcrux***, and ***mtspr*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book I; see Section 3.3.16 of Book I for the instruction definitions.

The fixed-point *Move To/From System Register* instructions from Book III-E, ***mfspr***<E.DC>, ***mtspr***, ***mfdcr***, ***mtdcr***<E.DC>, ***mtmsr***, ***mfmsr***, ***wrtee***, and ***wrteei*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E; see Section 5.4.1 of Book III-E for the instruction definitions.

Move From Count Register *R-form*

se_mfctr RX



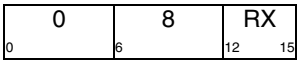
RX ← CTR

The CTR contents are placed into register RX.

Special Registers Altered:
None

Move From Link Register *R-form*

se_mflr RX



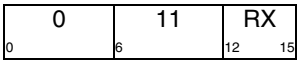
RX ← LR

The LR contents are placed into register RX.

Special Registers Altered:
None

Move To Count Register *R-form*

se_mtctr RX



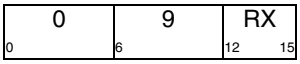
CTR ← (RX)

The contents of register RX are placed into the CTR.

Special Registers Altered:
CTR

Move To Link Register *R-form*

se_mtlr RX



LR ← (RX)

The contents of register RX are placed into the LR.

Special Registers Altered:
LR

Chapter 6. Storage Control Instructions

6.1 Storage Synchronization Instructions

The memory synchronization instructions implemented by category VLE are identical in semantics to those defined in Book II and Book III-E. The ***se_isync*** instruction is defined by category VLE, but has the same semantics as ***isync***.

The *Load and Reserve* and *Store Conditional* instructions from Book II, ***lbarx***, ***lharx***, ***lwarx***, ***stbcx.***, ***sthcx.***, and ***stwcx.,*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 4.4.2 of Book II for the instruction definitions.

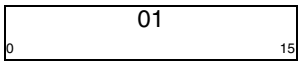
The *Load and Reserve* and *Store Conditional* instructions from Book II, ***ldarx*** and ***stdcx.*** are available while executing in VLE mode on 64-bit implementations. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 4.4.2 of Book II for the instruction definitions.

The *Memory Barrier* instructions from Book II, ***sync*** and ***mbar*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Section 4.4.3 of Book II for the instruction definitions.

The ***wait*** instruction from Book II is available while executing in VLE mode if the category Wait is implemented. The mnemonics, decoding, and semantics for ***wait*** are identical to those in Book II; see Section 4.4 of Book II for the instruction definition.

Instruction Synchronize *C-form*

se_isync



Executing an ***se_isync*** instruction ensures that all instructions preceding the ***se_isync*** instruction have completed before the ***se_isync*** instruction completes, and that no subsequent instructions are initiated until after the ***se_isync*** instruction completes. It also ensures that all instruction cache block invalidations caused by ***icbi*** instructions preceding the ***se_isync*** instruction have been performed with respect to the processor executing the ***se_isync*** instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the ***se_isync*** instruction may complete before storage accesses associated with instructions preceding the ***se_isync*** instruction have been performed. This instruction is context synchronizing.

The ***se_isync*** instruction has identical semantics to the Book II ***isync*** instruction, but has a different encoding.

Special Registers Altered:

None

6.2 Cache Management Instructions

Cache management instructions implemented by category VLE are identical to those defined in Book II and Book III-E.

The *Cache Management* instructions from Book II, ***dcba***, ***dcbf***, ***dcbst***, ***dcbt***, ***dcbtst***, ***dcbz***, ***icbi***, and ***icbt*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book II; see Section 4.3 of Book II for the instruction definitions.

The *Cache Management* instruction from Book III-E, ***dcbi*** is available while executing in VLE mode. The mnemonics, decoding, and semantics for this instruction are identical to those in Book III-E; see Section 6.11.1 of Book III-E for the instruction definition.

6.3 Cache Locking Instructions

Cache locking instructions implemented by category VLE are identical to those defined in Book III-E. If the *Cache Locking* instructions are implemented in category VLE, the Category: Embedded Cache Locking must also be implemented.

The *Cache Locking* instructions from Book III-E, ***dcbtls***, ***dcbtstls***, ***dcblc***, ***dcblq.***, ***icbtls***, ***icblq.***, and ***icblc*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E; see Section 6.11.2 of Book III-E for the instruction definitions.

6.4 TLB Management Instructions

The *TLB Management* instructions implemented by category VLE are identical to those defined in Book III-E.

The *TLB Management* instructions from Book III-E, ***tlbre***, ***tlbwe***, ***tlbivax***, ***tlbilx***, ***tlbsync***, ***tlbsrx.*** <E.TWC>, and ***tlbsx*** are available while executing in VLE mode. The mnemonics, decoding, and semantics for these instructions are identical to those in Book III-E. See Section 6.11.4.9 of Book III-E.

Instructions and resources described in Chapter 6 of Book III-E are available if the appropriate category is implemented.

6.5 Instruction Alignment and Byte Ordering

Only Big-Endian instruction memory is supported when executing from a page of VLE instructions. Attempting to fetch VLE instructions from a page marked as Little-Endian generates an instruction storage interrupt byte-ordering exception.

Chapter 7. Additional Categories Available in VLE

Instructions and resources from categories other than Base and Embedded are available in VLE. These include categories for which all the instructions in the category use primary opcode 4 or primary opcode 31.

7.1 Move Assist

Move Assist instructions implemented by category VLE are identical to those defined in Book I. If category Move Assist is supported in non-VLE mode, *Move Assist* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Section 3.3.7 of Book I for the instruction definitions.

7.2 Vector

Vector instructions implemented by category VLE are identical to those defined in Book I. If category Vector is supported in non-VLE mode, *Vector* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 5 of Book I for the instruction definitions.

7.3 Signal Processing Engine

Signal Processing Engine instructions implemented by category VLE are identical to those defined in Book I. If category Signal Processing Engine is supported in non-VLE mode, *Signal Processing Engine* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 8 of Book I for the instruction definitions.

7.4 Embedded Floating Point

Embedded Floating Point instructions implemented by category VLE are identical to those defined in Book I. If category SPE.Embedded Float Scalar Double, SPE.Embedded Float Scalar Single, or SPE.Embedded Float Vector is supported in non-VLE mode, the appropriate *Embedded Floating Point* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 9 of Book I for the instruction definitions.

7.5 Legacy Move Assist

Legacy Move Assist instructions implemented by category VLE are identical to those defined in Book I. If category Legacy Move Assist is supported in non-VLE mode, *Legacy Move Assist* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book I; see Chapter 10 of Book I for the instruction definitions.

7.6 Embedded Hypervisor

Embedded Hypervisor instructions implemented by category VLE are not identical to those defined in Book III - E. The ***ehpriv*** instruction is identical in mnemonics, decoding, and semantics to the instruction defined in Book III-E. See Section 4.3.1 of Book III-E for the instruction definition. The ***sc*** instruction which provides a LEV field for executing calls to the hypervisor software is implemented as ***e_sc***, which is defined in Section 4.3 of Book VLE. The ***rfgi*** instruction is implemented as ***se_rfgi***, which is also defined in Section 4.3 of Book VLE. If Category: Embedded Hypervisor is supported in non-VLE mode, Embedded Hypervisor instructions are also supported in VLE mode.

7.7 External PID

External Process ID instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.External PID is supported in non-VLE mode, *External Process ID* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Chapter 5.3.7 of Book III-E for the instruction definitions.

7.8 Embedded Performance Monitor

Embedded Performance Monitor instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Performance Monitor is supported in non-VLE mode, *Embedded Performance Monitor* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Appendix D of Book III-E for the instruction definitions.

7.9 Processor Control

Processor Control instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Processor Control is supported in non-VLE mode, *Processor Control* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; see Chapter 11. for the instruction definitions.

7.10 Decorated Storage

Decorated Storage instructions implemented by category VLE are identical to those defined in Book II. If category *Decorated Storage* is supported in non-VLE mode, Decorated Storage instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book II; see Chapter 8 of Book II for the instruction definitions.

7.11 Embedded Cache Initialization

Embedded Cache Initialization instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Cache Initialization is supported in non-VLE mode, *Embedded Cache Initialization* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; Chapter A.1 of Book III-E for the instruction definitions.

7.12 Embedded Cache Debug

Embedded Cache Debug instructions implemented by category VLE are identical to those defined in Book III-E. If Category: Embedded.Cache Debug is supported in non-VLE mode, *Embedded Cache Debug* instructions are also supported in VLE mode. The mnemonics, decoding, and semantics for those instructions are identical to those in Book III-E; Chapter A.2 of Book III-E for the instruction definitions.

Appendix A. VLE Instruction Set Sorted by Mnemonic

This appendix lists all the instructions available in VLE mode in the Power ISA, in order by mnemonic. Opcodes that are not defined below are treated as illegal by category VLE.

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
XO	7C000214	SR		B	add[o][.]	Add
XO	7C000014	SR		B	addc[o][.]	Add Carrying
XO	7C000114	SR		B	adde[o][.]	Add Extended
XO	7C0001D4	SR		B	addme[o][.]	Add to Minus One Extended
XO	7C000194	SR		B	addze[o][.]	Add to Zero Extended
X	7C000038	SR		B	and[.]	AND
X	7C000078	SR		B	andc[.]	AND with Complement
EVX	1000020F			SP	brinc	Bit Reversed Increment
X	7C000000			B	cmp	Compare
X	7C000040			B	cmpl	Compare Logical
X	7C000074	SR		64	cntlzd[.]	Count Leading Zeros Doubleword
X	7C000034	SR		B	cntlzw[.]	Count Leading Zeros Word
X	7C0005EC			E	dcba	Data Cache Block Allocate
X	7C0000AC			B	dcbf	Data Cache Block Flush
X	7C0000FE		P	E.PD	dcbfep	Data Cache Block Flush by External PID
X	7C0003AC		P	E	dcbi	Data Cache Block Invalidate
X	7C00030C		M	ECL	dcblc	Data Cache Block Lock Clear
X	7C00034D		M	ECL	dcblq.	Data Cache Block Lock Query
X	7C00006C			B	dcbst	Data Cache Block Store
X	7C00007E			E.PD	dcbstep	Data Cache Block Store by External PID
X	7C00022C			B	dcbt	Data Cache Block Touch
X	7C00027E		P	E.PD	dcbtep	Data Cache Block Touch by External PID
X	7C00014C		M	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	7C0001EC			B	dcbtst	Data Cache Block Touch for Store
X	7C0001FE		P	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID
X	7C00010C		M	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	7C0007EC			B	dcbz	Data Cache Block set to Zero
X	7C0007FE		P	E.PD	dcbzep	Data Cache Block set to Zero by External PID
X	7C00038C		H	E.CI	dci	Data Cache Invalidate
X	7C0003CC		H	E.CD	dcread	Data Cache Read [Alternative Encoding]
XO	7C0003D2	SR		64	divd[o][.]	Divide Doubleword
XO	7C000392	SR		64	divdu[o][.]	Divide Doubleword Unsigned
XO	7C0003D6	SR		B	divw[o][.]	Divide Word
XO	7C000396	SR		B	divwu[o][.]	Divide Word Unsigned
X	7C00009C	SR		LMV	dlmzb[.]	Determine Leftmost Zero Byte
X	7C0003C6			DS	dsn	Decorated Storage Notify
D	1C000000			VLE	e_add16i	Add Immediate
I16A	70008800	SR		VLE	e_add2i.	Add (2 operand) Immediate and Record
I16A	70009000			VLE	e_add2is	Add (2 operand) Immediate Shifted
SCI8	18008000	SR		VLE	e_addi[.]	Add Scaled Immediate
SCI8	18009000	SR		VLE	e_addic[.]	Add Scaled Immediate Carrying
I16L	7000C800	SR		VLE	e_and2i.	AND (two operand) Immediate
I16L	7000E800	SR		VLE	e_and2is.	AND (2 operand) Immediate Shifted
SCI8	1800C000	SR		VLE	e_andi[.]	AND Scaled Immediate
BD24	78000000			VLE	e_b[l]	Branch [and Link]

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
BD15	7A000000	CT		VLE	e_bc[l]	Branch Conditional [and Link]
I16A	70009800			VLE	e_cmp16i	Compare Immediate Word
X	7C00001C			VLE	e_cmph	Compare Halfword
I16A	7000B000			VLE	e_cmph16i	Compare Halfword Immediate
X	7C00005C			VLE	e_cmphl	Compare Halfword Logical
I16A	7000B800			VLE	e_cmphl16i	Compare Halfword Logical Immediate
SCI8	1800A800			VLE	e_cmpi	Compare Scaled Immediate Word
I16A	7000A800			VLE	e_cmpl16i	Compare Logical Immediate Word
SCI8	1880A800			VLE	e_cmpli	Compare Logical Scaled Immediate Word
XL	7C000202			VLE	e_crand	Condition Register AND
XL	7C000102			VLE	e_crandc	Condition Register AND with Complement
XL	7C000242			VLE	e_creqv	Condition Register Equivalent
XL	7C0001C2			VLE	e_crnand	Condition Register NAND
XL	7C000042			VLE	e_crnor	Condition Register NOR
XL	7C000382			VLE	e_cror	Condition Register OR
XL	7C000342			VLE	e_crorc	Condition Register OR with Complement
XL	7C000182			VLE	e_crxor	Condition Register XOR
D	30000000			VLE	e_lbz	Load Byte and Zero
D8	18000000			VLE	e_lbzu	Load Byte and Zero with Update
D	38000000			VLE	e_lha	Load Halfword Algebraic
D8	18000300			VLE	e_lhau	Load Halfword Algebraic with Update
D	58000000			VLE	e_lhz	Load Halfword and Zero
D8	18000100			VLE	e_lhzu	Load Halfword and Zero with Update
LI20	70000000			VLE	e_li	Load Immediate
I16L	7000E000			VLE	e_lis	Load Immediate Shifted
D8	18000800			VLE	e_lmw	Load Multiple Word
D	50000000			VLE	e_lwz	Load Word and Zero
D8	18000200			VLE	e_lwzu	Load Word and Zero with Update
XL	7C000020			VLE	e_mcrf	Move CR Field
I16A	7000A000			VLE	e_mull2i	Multiply (2 operand) Low Immediate
SCI8	1800A000			VLE	e_mulli	Multiply Low Scaled Immediate
I16L	7000C000			VLE	e_or2i	OR (two operand) Immediate
I16L	7000D000			VLE	e_or2is	OR (2 operand) Immediate Shifted
SCI8	1800D000	SR		VLE	e_ori[.]	OR Scaled Immediate
X	7C000230	SR		VLE	e_rlw[.]	Rotate Left Word
X	7C000270	SR		VLE	e_rlwi[.]	Rotate Left Word Immediate
M	74000000			VLE	e_rlwimi	Rotate Left Word Immediate then Mask Insert
M	74000001			VLE	e_rlwinm	Rotate Left Word Immediate then AND with Mask
ESC	7C000048			VLE, E.HV	e_sc	System Call
X	7C000070	SR		VLE	e_slwi[.]	Shift Left Word Immediate
X	7C000470	SR		VLE	e_srwi[.]	Shift Right Word Immediate
D	34000000			VLE	e_stb	Store Byte
D8	18000400			VLE	e_stbu	Store Byte with Update
D	5C000000			VLE	e_sth	Store Halfword
D8	18000500			VLE	e_sthu	Store Halfword with Update
D8	18000900			VLE	e_stmw	Store Multiple Word
D	54000000			VLE	e_stw	Store Word
D8	18000600			VLE	e_stwu	Store Word with Update
SCI8	1800B000	SR		VLE	e_subfic[.]	Subtract From Scaled Immediate Carrying
SCI8	1800E000	SR		VLE	e_xori[.]	XOR Scaled Immediate
EVX	100002E4			SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	100002E0			SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	100002EF			SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	100002F3			SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	100002F1			SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	100002E3			SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002F2			SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	100002F0			SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	100002E2			SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	100002EE			SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	100002EC			SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	100002ED			SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	100002F7			SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	100002F5			SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	100002EB			SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	100002FA			SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	100002F6			SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	100002F4			SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	100002EA			SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	100002F8			SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	100002E9			SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	100002E8			SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	100002E5			SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	100002E6			SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	100002E1			SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	100002FE			SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	100002FC			SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	100002FD			SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	100002C4			SP.FS	efsabs	Floating-Point Single-Precision Absolute Value
EVX	100002C0			SP.FS	efsadd	Floating-Point Single-Precision Add
EVX	100002CF			SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	100002D3			SP.FS	efscfsf	Convert Floating-Point Single-Precision from Signed Fraction
EVX	100002D1			SP.FS	efscfsi	Convert Floating-Point Single-Precision from Signed Integer
EVX	100002D2			SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	100002D0			SP.FS	efscfui	Convert Floating-Point Single-Precision from Unsigned Integer
EVX	100002CE			SP.FS	efscmpeq	Floating-Point Single-Precision Compare Equal
EVX	100002CC			SP.FS	efscmpgt	Floating-Point Single-Precision Compare Greater Than
EVX	100002CD			SP.FS	efscmplt	Floating-Point Single-Precision Compare Less Than
EVX	100002D7			SP.FS	efsctsf	Convert Floating-Point Single-Precision to Signed Fraction
EVX	100002D5			SP.FS	efsctsi	Convert Floating-Point Single-Precision to Signed Integer
EVX	100002DA			SP.FS	efsctsiz	Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero
EVX	100002D6			SP.FS	efsctuf	Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	100002D4			SP.FS	efsctui	Convert Floating-Point Single-Precision to Unsigned Integer
EVX	100002D8			SP.FS	efsctuiz	Convert Floating-Point Single-Precision to Unsigned Integer with Round Towards Zero
EVX	100002C9			SP.FS	efsdiv	Floating-Point Single-Precision Divide

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002C8			SP.FS	efsmul	Floating-Point Single-Precision Multiply
EVX	100002C5			SP.FS	efsnabs	Floating-Point Single-Precision Negative Absolute Value
EVX	100002C6			SP.FS	efsneg	Floating-Point Single-Precision Negate
EVX	100002C1			SP.FS	efssub	Floating-Point Single-Precision Subtract
EVX	100002DE			SP.FS	efststeq	Floating-Point Single-Precision Test Equal
EVX	100002DC			SP.FS	efststgt	Floating-Point Single-Precision Test Greater Than
EVX	100002DD			SP.FS	efststlt	Floating-Point Single-Precision Test Less Than
XL	7C00021C			E.HV	ehpriv	Embedded Hypervisor Privilege
X	7C000238	SR		B	eqv[.]	Equivalent
EVX	10000208			SP	evabs	Vector Absolute Value
EVX	10000202			SP	evaddiw	Vector Add Immediate Word
EVX	100004C9			SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	100004C1			SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	100004C8			SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	100004C0			SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	10000200			SP	evaddw	Vector Add Word
EVX	10000211			SP	evand	Vector AND
EVX	10000212			SP	evandc	Vector AND with Complement
EVX	10000234			SP	evcmpeq	Vector Compare Equal
EVX	10000231			SP	evcmpgts	Vector Compare Greater Than Signed
EVX	10000230			SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	10000233			SP	evcmplts	Vector Compare Less Than Signed
EVX	10000232			SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	1000020E			SP	evcntlsw	Vector Count Leading Signed Bits Word
EVX	1000020D			SP	evcntlzw	Vector Count Leading Zeros Word
EVX	100004C6			SP	evdivws	Vector Divide Word Signed
EVX	100004C7			SP	evdivwu	Vector Divide Word Unsigned
EVX	10000219			SP	eveqv	Vector Equivalent
EVX	1000020A			SP	evextsb	Vector Extend Sign Byte
EVX	1000020B			SP	evextsh	Vector Extend Sign Halfword
EVX	10000284			SP.FV	evfsabs	Vector Floating-Point Single-Precision Absolute Value
EVX	10000280			SP.FV	evfsadd	Vector Floating-Point Single-Precision Add
EVX	10000293			SP.FV	evfscfsf	Vector Convert Floating-Point Single-Precision from Signed Fraction
EVX	10000291			SP.FV	evfscfsi	Vector Convert Floating-Point Single-Precision from Signed Integer
EVX	10000292			SP.FV	evfscfuf	Vector Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	10000290			SP.FV	evfscfui	Vector Convert Floating-Point Single-Precision from Unsigned Integer
EVX	1000028E			SP.FV	evfscmpeq	Vector Floating-Point Single-Precision Compare Equal
EVX	1000028C			SP.FV	evfscmpgt	Vector Floating-Point Single-Precision Compare Greater Than
EVX	1000028D			SP.FV	evfscmplt	Vector Floating-Point Single-Precision Compare Less Than
EVX	10000297			SP.FV	evfsctsf	Vector Convert Floating-Point Single-Precision to Signed Fraction
EVX	10000295			SP.FV	evfsctsi	Vector Convert Floating-Point Single-Precision to Signed Integer
EVX	1000029A			SP.FV	evfsctsiz	Vector Convert Floating-Point Single-Precision to Signed Integer with Round Toward Zero
EVX	10000296			SP.FV	evfsctuf	Vector Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	10000294			SP.FV	evfsctui	Vector Convert Floating-Point Single-Precision to Unsigned Integer
EVX	10000298			SP.FV	evfsctuiz	Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000289			SP.FV	evfsdiv	Vector Floating-Point Single-Precision Divide
EVX	10000288			SP.FV	evfsmul	Vector Floating-Point Single-Precision Multiply
EVX	10000285			SP.FV	evfsnabs	Vector Floating-Point Single-Precision Negative Absolute Value
EVX	10000286			SP.FV	evfsneg	Vector Floating-Point Single-Precision Negate
EVX	10000281			SP.FV	evfssub	Vector Floating-Point Single-Precision Subtract
EVX	1000029E			SP.FV	evfststeq	Vector Floating-Point Single-Precision Test Equal
EVX	1000029C			SP.FV	evfststgt	Vector Floating-Point Single-Precision Test Greater Than
EVX	1000029D			SP.FV	evfststlt	Vector Floating-Point Single-Precision Test Less Than
EVX	10000301			SP	evldd	Vector Load Double Word into Double Word
EVX	7C00063E		P	E.PD	evlddepx	Vector Load Doubleword into Doubleword by External Process ID Indexed
EVX	10000300			SP	evlddx	Vector Load Double Word into Double Word Indexed
EVX	10000305			SP	evldh	Vector Load Double Word into Four Halfwords
EVX	10000304			SP	evldhx	Vector Load Double Word into Four Halfwords Indexed
EVX	10000303			SP	evldw	Vector Load Double Word into Two Words
EVX	10000302			SP	evldwx	Vector Load Double Word into Two Words Indexed
EVX	10000309			SP	evlhhesplat	Vector Load Halfword into Halfwords Even and Splat
EVX	10000308			SP	evlhhesplatx	Vector Load Halfword into Halfwords Even and Splat Indexed
EVX	1000030F			SP	evlhhossplat	Vector Load Halfword into Halfword Odd and Splat
EVX	1000030E			SP	evlhhossplatx	Vector Load Halfword into Halfword Odd Signed and Splat Indexed
EVX	1000030D			SP	evlhhousplat	Vector Load Halfword into Halfword Odd Unsigned and Splat
EVX	1000030C			SP	evlhhousplatx	Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed
EVX	10000311			SP	evlwhe	Vector Load Word into Two Halfwords Even
EVX	10000310			SP	evlwhex	Vector Load Word into Two Halfwords Even Indexed
EVX	10000317			SP	evlwhos	Vector Load Word into Two Halfwords Odd Signed (with sign extension)
EVX	10000316			SP	evlwhosx	Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)
EVX	10000315			SP	evlwhou	Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)
EVX	10000314			SP	evlwhoux	Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)
EVX	1000031D			SP	evlwhsplat	Vector Load Word into Two Halfwords and Splat
EVX	1000031C			SP	evlwhsplatx	Vector Load Word into Two Halfwords and Splat Indexed
EVX	10000319			SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	10000318			SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
EVX	1000022C			SP	evmergehi	Vector Merge High
EVX	1000022E			SP	evmergehilo	Vector Merge High/Low
EVX	1000022D			SP	evmergelo	Vector Merge Low
EVX	1000022F			SP	evmergelohi	Vector Merge Low/High
EVX	1000052B			SP	evmhegsmfaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	100005AB			SP	evmhegsmfan	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	10000529			SP	evmhegsmiaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	100005A9			SP	evmhegsmian	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	10000528			SP	evmhegumiaa	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	100005A8			SP	evmhegumian	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	1000040B			SP	evmhesmf	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	1000042B			SP	evmhesmfa	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulate
EVX	1000050B			SP	evmhesmfaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	1000058B			SP	evmhesmfanw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	10000409			SP	evmhesmi	Vector Multiply Halfwords, Even, Signed, Modulo, Integer
EVX	10000429			SP	evmhesmia	Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX	10000509			SP	evmhesmiaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	10000589			SP	evmhesmianw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	10000403			SP	evmhessf	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional
EVX	10000423			SP	evmhessfa	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator
EVX	10000503			SP	evmhessfaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000583			SP	evmhessfanw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000501			SP	evmhessiaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words
EVX	10000581			SP	evmhessianw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	10000408			SP	evmheumi	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer
EVX	10000428			SP	evmheumia	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator
EVX	10000508			SP	evmheumiaaw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	10000588			SP	evmheumianw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000500			SP	evmheusiaaw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words
EVX	10000580			SP	evmheusianw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	1000052F			SP	evmhogsmfaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	100005AF			SP	evmhogsmfan	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	1000052D			SP	evmhogsmiaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer, and Accumulate
EVX	100005AD			SP	evmhogsmian	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	1000052C			SP	evmhogumiaa	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	100005AC			SP	evmhogumian	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	1000040F			SP	evmhosmf	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional
EVX	1000042F			SP	evmhosmfa	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator
EVX	1000050F			SP	evmhosmfaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	1000058F			SP	evmhosmfanw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	1000040D			SP	evmhosmi	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	1000042D			SP	evmhosmia	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator
EVX	1000050D			SP	evmhosmiaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	1000058D			SP	evmhosmianw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	10000407			SP	evmhossf	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional
EVX	10000427			SP	evmhossfa	Vector Multiply Halfwords, Odd, Signed, Fractional to Accumulator
EVX	10000507			SP	evmhossfaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000587			SP	evmhossfanw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000505			SP	evmhossiaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	10000585			SP	evmhossianw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	1000040C			SP	evmhoumi	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer
EVX	1000042C			SP	evmhoumia	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	1000050C			SP	evmhoumiaaw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	1000058C			SP	evmhoumianw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000504			SP	evmhousiaaw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words
EVX	10000584			SP	evmhousianw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	100004C4			SP	evmra	Initialize Accumulator
EVX	1000044F			SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	1000046F			SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	1000044D			SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	1000046D			SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	10000447			SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	10000467			SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	1000044C			SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
EVX	1000046C			SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	10000549			SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words
EVX	100005C9			SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
EVX	10000541			SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words
EVX	100005C1			SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
EVX	10000448			SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
EVX	10000468			SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	10000548			SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words
EVX	100005C8			SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000540			SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words
EVX	100005C0			SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	1000045B			SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	1000047B			SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
EVX	1000055B			SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	100005DB			SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
EVX	10000459			SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	10000479			SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	10000559			SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	100005D9			SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	10000453			SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	10000473			SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	10000553			SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	100005D3			SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	10000458			SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	10000478			SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	10000558			SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	100005D8			SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	1000021E			SP	evnand	Vector NAND
EVX	10000209			SP	evneg	Vector Negate
EVX	10000218			SP	evnor	Vector NOR
EVX	10000217			SP	evor	Vector OR
EVX	1000021B			SP	evorc	Vector OR with Complement
EVX	10000228			SP	evrlw	Vector Rotate Left Word
EVX	1000022A			SP	evrlwi	Vector Rotate Left Word Immediate
EVX	1000020C			SP	evrndw	Vector Round Word
EVS	10000278			SP	evsel	Vector Select
EVX	10000224			SP	evslw	Vector Shift Left Word
EVX	10000226			SP	evslwi	Vector Shift Left Word Immediate
EVX	1000022B			SP	evsplatfi	Vector Splat Fractional Immediate
EVX	10000229			SP	evsplati	Vector Splat Immediate
EVX	10000223			SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	10000222			SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	10000221			SP	evsrws	Vector Shift Right Word Signed
EVX	10000220			SP	evsrwu	Vector Shift Right Word Unsigned
EVX	10000321			SP	evstdd	Vector Store Double of Double
EVX	7C00073E		P	E.PD	evstddepx	Vector Store Doubleword into Doubleword by External Process ID Indexed
EVX	10000320			SP	evstddx	Vector Store Doubleword of Doubleword Indexed
EVX	10000325			SP	evstdh	Vector Store Double of Four Halfwords
EVX	10000324			SP	evstdhx	Vector Store Double of Four Halfwords Indexed
EVX	10000323			SP	evstdw	Vector Store Double of Two Words
EVX	10000322			SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	10000331			SP	evstwhe	Vector Store Word of Two Halfwords from Even
EVX	10000330			SP	evstwhex	Vector Store Word of Two Halfwords from Even Indexed
EVX	10000335			SP	evstwho	Vector Store Word of Two Halfwords from Odd

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000334			SP	evstwhox	Vector Store Word of Two Halfwords from Odd Indexed
EVX	10000339			SP	evstwwe	Vector Store Word of Word from Even
EVX	10000338			SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	1000033D			SP	evstwwo	Vector Store Word of Word from Odd
EVX	1000033C			SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	100004CB			SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	100004C3			SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	100004CA			SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	100004C2			SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	10000204			SP	evsubfw	Vector Subtract from Word
EVX	10000206			SP	evsubifw	Vector Subtract Immediate from Word
EVX	10000216			SP	evxor	Vector XOR
X	7C000774	SR		B	extsb[.]	Extend Sign Byte
X	7C000734	SR		B	extsh[.]	Extend Sign Halfword
X	7C0007B4	SR		64	extsw[.]	Extend Sign Word
X	7C0007AC			B	icbi	Instruction Cache Block Invalidate
X	7C0007BE		P	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	7C0001CC		M	ECL	icblc	Instruction Cache Block Lock Clear
X	7C00018D		M	ECL	icblq.	Instruction Cache Block Lock Query
X	7C00002C			B	icbt	Instruction Cache Block Touch
X	7C0003CC		M	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	7C00078C		H	E.CI	ici	Instruction Cache Invalidate
X	7C0007CC		H	E.CD	icread	Instruction Cache Read
A	7C00001E			B	isel	Integer Select
X	7C000068			B	lbarx	Load Byte and Reserve Indexed
X	7C000406			DS	lbdx	Load Byte with Decoration Indexed
X	7C0000BE		P	E.PD	lbepx	Load Byte by External Process ID Indexed
X	7C0000EE			B	lbzux	Load Byte and Zero with Update Indexed
X	7C0000AE			B	lbzx	Load Byte and Zero Indexed
X	7C0000A8			64	ldarx	Load Doubleword And Reserve Indexed
X	7C0004C6			DS	lddx	Load Doubleword with Decoration Indexed
X	7C00003A		P	E.PD;64	ldepx	Load Doubleword by External Process ID Indexed
X	7C00006A			64	ldux	Load Doubleword with Update Indexed
X	7C00002A			64	ldx	Load Doubleword Indexed
X	7C000646			DS	lfddx	Load Floating Doubleword with Decoration Indexed
X	7C0004BE		P	E.PD	lfdepx	Load Floating-Point Double by External Process ID Indexed
X	7C0000E8			B	lharx	Load Halfword and Reserve Indexed
X	7C0002EE			B	lhaux	Load Halfword Algebraic with Update Indexed
X	7C0002AE			B	lhax	Load Halfword Algebraic Indexed
X	7C00062C			B	lhbrx	Load Halfword Byte-Reverse Indexed
X	7C000446			DS	lhdx	Load Halfword with Decoration Indexed
X	7C00023E		P	E.PD	lhepx	Load Halfword by External Process ID Indexed
X	7C00026E			B	lhzux	Load Halfword and Zero with Update Indexed
X	7C00022E			B	lhzx	Load Halfword and Zero Indexed
X	7C0004AA			MA	lswi	Load String Word Immediate
X	7C00042A			MA	lswx	Load String Word Indexed
X	7C00000E			V	lvebx	Load Vector Element Byte Indexed
X	7C00004E			V	lvehx	Load Vector Element Halfword Indexed
X	7C00024E		P	E.PD	lvepx	Load Vector by External Process ID Indexed
X	7C00020E		P	E.PD	lvepxl	Load Vector by External Process ID Indexed LRU
X	7C00008E			V	lvewx	Load Vector Element Word Indexed
X	7C00000C			V	lvsl	Load Vector for Shift Left Indexed
X	7C00004C			V	lvsr	Load Vector for Shift Right Indexed
X	7C0000CE			V	lvx	Load Vector Indexed
X	7C0002CE			V	lvxl	Load Vector Indexed LRU
X	7C000028			B	lwarx	Load Word And Reserve Indexed

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C0002EA			64	lwaux	Load Word Algebraic with Update Indexed
X	7C0002AA			64	lwax	Load Word Algebraic Indexed
X	7C00042C			B	lwbrx	Load Word Byte-Reverse Indexed
X	7C000486			DS	lwdx	Load Word with Decoration Indexed
X	7C00003E		P	E.PD	lwepx	Load Word by External Process ID Indexed
X	7C00006E			B	lwzux	Load Word and Zero with Update Indexed
X	7C00002E			B	lwzx	Load Word and Zero Indexed
XO	10000158	SR		LMA	macchw[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	100001D8	SR		LMA	macchws[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	10000198	SR		LMA	macchwsu[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
XO	10000118	SR		LMA	macchwu[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
XO	10000058	SR		LMA	machhw[o][.]	Multiply Accumulate High Halfword to Word Modulo Signed
XO	100000D8	SR		LMA	machhws[o][.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	10000098	SR		LMA	machhwsu[o][.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
XO	10000018	SR		LMA	machhwu[o][.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
XO	10000358	SR		LMA	maclhw[o][.]	Multiply Accumulate Low Halfword to Word Modulo Signed
XO	100003D8	SR		LMA	maclhws[o][.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	10000398	SR		LMA	maclhwsu[o][.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
XO	10000318	SR		LMA	maclhwu[o][.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
X	7C0006AC			E	mbar	Memory Barrier
X	7C000400			E	mcrxr	Move To Condition Register from XER
XFX	7C000026			B	mfcr	Move From Condition Register
XFX	7C000286		P	E.DC	mfdcr	Move From Device Control Register
X	7C000246			E.DC	mfdcrux	Move From Device Control Register User-mode Indexed
X	7C000206		P	E.DC	mfdcrx	Move From Device Control Register Indexed
X	7C0000A6		P	B	mfmsr	Move From Machine State Register
XFX	7C100026			B	mfocrf	Move From One Condition Register Field
XFX	7C00029C		O	E.PM	mfpmr	Move From Performance Monitor Register
XFX	7C0002A6		O	B	mfspr	Move From Special Purpose Register
VX	10000604			V	mfvscr	Move From VSCR
X	7C0001DC		H	E.PC	msgclr	Message Clear
X	7C00019C		H	E.PC	msgsnd	Message Send
XFX	7C000120			B	mtcrf	Move To Condition Register Fields
XFX	7C000386		P	E.DC	mtdcr	Move To Device Control Register
X	7C000346			E.DC	mtdcrux	Move To Device Control Register User-mode Indexed
X	7C000306		P	E.DC	mtdcrx	Move To Device Control Register Indexed
X	7C000124		P	E	mtmsr	Move To Machine State Register
XFX	7C100120			B	mtocrf	Move To One Condition Register Field
XFX	7C00039C		O	E.PM	mtpmr	Move To Performance Monitor Register
XFX	7C0003A6		O	B	mtspr	Move To Special Purpose Register
VX	10000644			V	mtvscr	Move To VSCR
X	10000150	SR		LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
X	10000110	SR		LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
XO	7C000092	SR		64	mulhd[.]	Multiply High Doubleword
XO	7C000012	SR		64	mulhdu[.]	Multiply High Doubleword Unsigned
X	10000050	SR		LMA	mulhhw[.]	Multiply High Halfword to Word Signed
X	10000010	SR		LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
XO	7C000096	SR		B	mulhw[.]	Multiply High Word
XO	7C000016	SR		B	mulhwu[.]	Multiply High Word Unsigned

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
XO	7C0001D2	SR		64	mulld[o][.]	Multiply Low Doubleword
X	10000350	SR		LMA	mullhw[.]	Multiply Low Halfword to Word Signed
X	10000310	SR		LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
XO	7C0001D6	SR		B	mullw[o][.]	Multiply Low Word
X	7C0003B8	SR		B	nand[.]	NAND
XO	7C0000D0	SR		B	neg[o][.]	Negate
XO	1000015C	SR		LMA	nmacchw[o][.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	100001DC	SR		LMA	nmacchws[o][.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	1000005C	SR		LMA	nmachhw[o][.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
XO	100000DC	SR		LMA	nmachhws[o][.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
XO	1000035C	SR		LMA	nmaclhw[o][.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
XO	100003DC	SR		LMA	nmaclhws[o][.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
X	7C0000F8	SR		B	nor[.]	NOR
X	7C000378	SR		B	or[.]	OR
X	7C000338	SR		B	orc[.]	OR with Complement
X	7C0000F4			B	popcntb	Population Count Bytes
RR	0400----			VLE	se_add	Add Short Form
OIM5	2000----			VLE	se_addi	Add Immediate Short Form
RR	4600----	SR		VLE	se_and[.]	AND Short Form
RR	4500----			VLE	se_andc	AND with Complement Short Form
IM5	2E00----			VLE	se_andi	AND Immediate Short Form
BD8	E800----			VLE	se_b[l]	Branch [and Link]
BD8	E000----			VLE	se_bc	Branch Conditional Short Form
IM5	6000----			VLE	se_bclri	Bit Clear Immediate
C	0006----			VLE	se_bctr[l]	Branch to Count Register [and Link]
IM5	6200----			VLE	se_bgeni	Bit Generate Immediate
C	0004----			VLE	se_blr[l]	Branch to Link Register [and Link]
IM5	2C00----			VLE	se_bmaski	Bit Mask Generate Immediate
IM5	6400----			VLE	se_bseti	Bit Set Immediate
IM5	6600----			VLE	se_btsti	Bit Test Immediate
RR	0C00----			VLE	se_cmp	Compare Word
RR	0E00----			VLE	se_cmph	Compare Halfword Short Form
RR	0F00----			VLE	se_cmphl	Compare Halfword Logical Short Form
IM5	2A00----			VLE	se_cmpi	Compare Immediate Word Short Form
RR	0D00----			VLE	se_cmpl	Compare Logical Word
OIM5	2200----			VLE	se_cmpli	Compare Logical Immediate Word
R	00D0----			VLE	se_extsb	Extend Sign Byte Short Form
R	00F0----			VLE	se_extsh	Extend Sign Halfword Short Form
R	00C0----			VLE	se_extzb	Extend Zero Byte
R	00E0----			VLE	se_extzh	Extend Zero Halfword
C	0000----			VLE	se_illegal	Illegal
C	0001----			VLE	se_isync	Instruction Synchronize
SD4	8000----			VLE	se_lbz	Load Byte and Zero Short Form
SD4	A000----			VLE	se_lhz	Load Halfword and Zero Short Form
IM7	4800----			VLE	se_li	Load Immediate Short Form
SD4	C000----			VLE	se_lwz	Load Word and Zero Short Form
RR	0300----			VLE	se_mfar	Move from Alternate Register
R	00A0----			VLE	se_mfctr	Move From Count Register
R	0080----			VLE	se_mflr	Move From Link Register
RR	0100----			VLE	se_mr	Move Register
RR	0200----			VLE	se_mtar	Move To Alternate Register
R	00B0----			VLE	se_mtctr	Move To Count Register
R	0090----			VLE	se_mtlr	Move To Link Register
RR	0500----			VLE	se_mullw	Multiply Low Word Short Form
R	0030----			VLE	se_neg	Negate Short Form

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
R	0020----			VLE	se_not	NOT Short Form
RR	4400----			VLE	se_or	OR Short Form
C	0009----		H	VLE	se_rfci	Return From Critical Interrupt
C	000A----		H	VLE	se_rfdi	Return From Debug Interrupt
C	000C----		P	VLE, E.HV	se_rfgi	Return From Guest Interrupt
C	0008----		P	VLE	se_rfi	Return From Interrupt
C	000B----		H	VLE	se_rfmci	Return From Machine Check Interrupt
C	0002----			VLE	se_sc	System Call
RR	4200----			VLE	se_slw	Shift Left Word
IM5	6C00----			VLE	se_slwi	Shift Left Word Immediate Short Form
RR	4100----			VLE	se_sraw	Shift Right Algebraic Word
IM5	6A00----			VLE	se_srawi	Shift Right Algebraic Immediate
RR	4000----			VLE	se_srw	Shift Right Word
IM5	6800----			VLE	se_srwi	Shift Right Word Immediate Short Form
SD4	9000----			VLE	se_stb	Store Byte Short Form
SD4	B000----			VLE	se_sth	Store Halfword Short Form
SD4	D000----			VLE	se_stw	Store Word Short Form
RR	0600----			VLE	se_sub	Subtract
RR	0700----			VLE	se_subf	Subtract From Short Form
OIM5	2400----	SR		VLE	se_subi[.]	Subtract Immediate
X	7C000036	SR		64	sld[.]	Shift Left Doubleword
X	7C000030	SR		B	slw[.]	Shift Left Word
X	7C000634	SR		64	srad[.]	Shift Right Algebraic Doubleword
XS	7C000674	SR		64	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	7C000630	SR		B	sraw[.]	Shift Right Algebraic Word
X	7C000670	SR		B	srawi[.]	Shift Right Algebraic Word Immediate
X	7C000436	SR		64	srd[.]	Shift Right Doubleword
X	7C000430	SR		B	srw[.]	Shift Right Word
X	7C00056D			B	stbcx.	Store Byte Conditional Indexed
X	7C000506			DS	stbdx	Store Byte with Decoration Indexed
X	7C0001BE		P	E.PD	stbepx	Store Byte by External Process ID Indexed
X	7C0001EE			B	stbux	Store Byte with Update Indexed
X	7C0001AE			B	stbx	Store Byte Indexed
X	7C0001AD			64	stdcx.	Store Doubleword Conditional Indexed
X	7C0005C6			DS	stddx	Store Doubleword with Decoration Indexed
X	7C00013A		P	E.PD;64	stdepx	Store Doubleword by External Process ID Indexed
X	7C00016A			64	stdux	Store Doubleword with Update Indexed
X	7C00012A			64	stdx	Store Doubleword Indexed
X	7C000746			DS	stfddx	Store Floating Doubleword with Decoration Indexed
X	7C0005BE		P	E.PD	stfdepx	Store Floating-Point Double by External Process ID Indexed
X	7C00072C			B	sthbrx	Store Halfword Byte-Reverse Indexed
X	7C0005AD			B	sthcx.	Store Halfword Conditional Indexed
X	7C000546			DS	sthdx	Store Halfword with Decoration Indexed
X	7C00033E		P	E.PD	sthepx	Store Halfword by External Process ID Indexed
X	7C00036E			B	sthux	Store Halfword with Update Indexed
X	7C00032E			B	sthx	Store Halfword Indexed
X	7C0005AA			MA	stswi	Store String Word Immediate
X	7C00052A			MA	stswx	Store String Word Indexed
X	7C00010E			V	stvebx	Store Vector Element Byte Indexed
X	7C00014E			V	stvehx	Store Vector Element Halfword Indexed
X	7C00064E		P	E.PD	stvepx	Store Vector by External Process ID Indexed
X	7C00060E		P	E.PD	stvepxl	Store Vector by External Process ID Indexed LRU
X	7C00018E			V	stvewx	Store Vector Element Word Indexed
X	7C0001CE			V	stvx	Store Vector Indexed
X	7C0003CE			V	stvxl	Store Vector Indexed LRU
X	7C00052C			B	stwbrx	Store Word Byte-Reverse Indexed
X	7C00012D			B	stwcx.	Store Word Conditional Indexed
X	7C000586			DS	stwdx	Store Word with Decoration Indexed
X	7C00013E		P	E.PD	stwepx	Store Word by External Process ID Indexed

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C00016E			B	stwux	Store Word with Update Indexed
X	7C00012E			B	stwx	Store Word Indexed
XO	7C000050	SR		B	subf[o][.]	Subtract From
XO	7C000010	SR		B	subfc[o][.]	Subtract From Carrying
XO	7C000110	SR		B	subfe[o][.]	Subtract From Extended
XO	7C0001D0	SR		B	subfme[o][.]	Subtract From Minus One Extended
XO	7C000190	SR		B	subfze[o][.]	Subtract From Zero Extended
X	7C0004AC			B	sync	Synchronize
X	7C000088			64	td	Trap Doubleword
X	7C000624		H	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	7C000764		H	E	tlbre	TLB Read Entry
X	7C000724		H	E	tlbsx	TLB Search Indexed
X	7C00046C		H	E	tlbsync	TLB Synchronize
X	7C0007A4		H	E	tlbwe	TLB Write Entry
X	7C000008			B	tw	Trap Word
VX	10000180			V	vaddcuw	Vector Add and write Carry-out Unsigned Word
VX	1000000A			V	vaddfp	Vector Add Single-Precision
VX	10000300			V	vaddsbs	Vector Add Signed Byte Saturate
VX	10000340			V	vaddshs	Vector Add Signed Halfword Saturate
VX	10000380			V	vaddsws	Vector Add Signed Word Saturate
VX	10000000			V	vaddubm	Vector Add Unsigned Byte Modulo
VX	10000200			V	vaddubs	Vector Add Unsigned Byte Saturate
VX	10000040			V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	10000240			V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	10000080			V	vadduwm	Vector Add Unsigned Word Modulo
VX	10000280			V	vadduws	Vector Add Unsigned Word Saturate
VX	10000404			V	vand	Vector Logical AND
VX	10000444			V	vandc	Vector Logical AND with Complement
VX	10000502			V	vavgsb	Vector Average Signed Byte
VX	10000542			V	vavgsh	Vector Average Signed Halfword
VX	10000582			V	vavgsw	Vector Average Signed Word
VX	10000402			V	vavgub	Vector Average Unsigned Byte
VX	10000442			V	vavguh	Vector Average Unsigned Halfword
VX	10000482			V	vavguw	Vector Average Unsigned Word
VX	1000034A			V	vcfsx	Vector Convert From Signed Fixed-Point Word
VX	1000030A			V	vcfux	Vector Convert From
VC	100003C6			V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VC	100000C6			V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VC	10000006			V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VC	10000046			V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VC	10000086			V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VC	100001C6			V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VC	100002C6			V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VC	10000306			V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
VC	10000346			V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VC	10000386			V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VC	10000206			V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
VC	10000246			V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VC	10000286			V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
VX	100003CA			V	vctsxs	Vector Convert To Signed Fixed-Point Word Saturate
VX	1000038A			V	vctuxs	Vector Convert To Unsigned Fixed-Point Word Saturate
VX	1000018A			V	vexptefp	Vector 2 Raised to the Exponent Estimate Floating-Point
VX	100001CA			V	vlogefp	Vector Log Base 2 Estimate Floating-Point
VA	1000002E			V	vmaddfp	Vector Multiply-Add Single-Precision
VX	1000040A			V	vmaxfp	Vector Maximum Single-Precision
VX	10000102			V	vmaxsb	Vector Maximum Signed Byte
VX	10000142			V	vmaxsh	Vector Maximum Signed Halfword
VX	10000182			V	vmaxsw	Vector Maximum Signed Word
VX	10000002			V	vmaxub	Vector Maximum Unsigned Byte
VX	10000042			V	vmaxuh	Vector Maximum Unsigned Halfword

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VX	10000082			V	vmaxuw	Vector Maximum Unsigned Word
VA	10000020			V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	10000021			V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VX	1000044A			V	vminfp	Vector Minimum Single-Precision
VX	10000302			V	vminsb	Vector Minimum Signed Byte
VX	10000342			V	vminsh	Vector Minimum Signed Halfword
VX	10000382			V	vminsw	Vector Minimum Signed Word
VX	10000202			V	vminub	Vector Minimum Unsigned Byte
VX	10000242			V	vminuh	Vector Minimum Unsigned Halfword
VX	10000282			V	vminuw	Vector Minimum Unsigned Word
VA	10000022			V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VX	1000000C			V	vmrghb	Vector Merge High Byte
VX	1000004C			V	vmrghh	Vector Merge High Halfword
VX	1000008C			V	vmrghw	Vector Merge High Word
VX	1000010C			V	vmrglb	Vector Merge Low Byte
VX	1000014C			V	vmrglh	Vector Merge Low Halfword
VX	1000018C			V	vmrglw	Vector Merge Low Word
VA	10000025			V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	10000028			V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	10000029			V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	10000024			V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	10000026			V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	10000027			V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VX	10000308			V	vmulesb	Vector Multiply Even Signed Byte
VX	10000348			V	vmulesh	Vector Multiply Even Signed Halfword
VX	10000208			V	vmuleub	Vector Multiply Even Unsigned Byte
VX	10000248			V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	10000108			V	vmulosb	Vector Multiply Odd Signed Byte
VX	10000148			V	vmulosh	Vector Multiply Odd Signed Halfword
VX	10000008			V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	10000048			V	vmulouh	Vector Multiply Odd Unsigned Halfword
VA	1000002F			V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	10000504			V	vnor	Vector Logical NOR
VX	10000484			V	vor	Vector Logical OR
VA	1000002B			V	vperm	Vector Permute
VX	1000030E			V	vpkpx	Vector Pack Pixel
VX	1000018E			V	vpkshss	Vector Pack Signed Halfword Signed Saturate
VX	1000010E			V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	100001CE			V	vpkswss	Vector Pack Signed Word Signed Saturate
VX	1000014E			V	vpkswus	Vector Pack Signed Word Unsigned Saturate
VX	1000000E			V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VX	1000008E			V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
VX	1000004E			V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
VX	100000CE			V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
VX	1000010A			V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	100002CA			V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
VX	1000020A			V	vrfin	Vector Round to Single-Precision Integer Nearest
VX	1000028A			V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
VX	1000024A			V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	10000004			V	vrlb	Vector Rotate Left Byte
VX	10000044			V	vrlh	Vector Rotate Left Halfword
VX	10000084			V	vrlw	Vector Rotate Left Word
VX	1000014A			V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VA	1000002A			V	vsel	Vector Select
VX	100001C4			V	vsl	Vector Shift Left
VX	10000104			V	vslb	Vector Shift Left Byte
VA	1000002C			V	vsldoi	Vector Shift Left Double by Octet Immediate
VX	10000144			V	vslh	Vector Shift Left Halfword
VX	1000040C			V	vslo	Vector Shift Left by Octet
VX	10000184			V	vslw	Vector Shift Left Word

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VX	1000020C			V	vspltb	Vector Splat Byte
VX	1000024C			V	vsplth	Vector Splat Halfword
VX	1000030C			V	vspltisb	Vector Splat Immediate Signed Byte
VX	1000034C			V	vspltish	Vector Splat Immediate Signed Halfword
VX	1000038C			V	vspltisw	Vector Splat Immediate Signed Word
VX	1000028C			V	vspltw	Vector Splat Word
VX	100002C4			V	vsr	Vector Shift Right
VX	10000304			V	vsrab	Vector Shift Right Algebraic Word
VX	10000344			V	vsrah	Vector Shift Right Algebraic Halfword
VX	10000384			V	vsraw	Vector Shift Right Algebraic Word
VX	10000204			V	vsrb	Vector Shift Right Byte
VX	10000244			V	vsrh	Vector Shift Right Halfword
VX	1000044C			V	vsro	Vector Shift Right by Octet
VX	10000284			V	vsrw	Vector Shift Right Word
VX	10000580			V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
VX	1000004A			V	vsubfp	Vector Subtract Single-Precision
VX	10000700			V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	10000740			V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	10000780			V	vsubsws	Vector Subtract Signed Word Saturate
VX	10000400			V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	10000600			V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	10000440			V	vsubuhm	Vector Subtract Unsigned Halfword Modulo
VX	10000640			V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	10000480			V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	10000680			V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	10000688			V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	10000708			V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	10000648			V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	10000608			V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	10000788			V	vsumsws	Vector Sum across Signed Word Saturate
VX	1000034E			V	vupkhpx	Vector Unpack High Pixel
VX	1000020E			V	vupkhsb	Vector Unpack High Signed Byte
VX	1000024E			V	vupkhsh	Vector Unpack High Signed Halfword
VX	100003CE			V	vupklpx	Vector Unpack Low Pixel
VX	1000028E			V	vupklsb	Vector Unpack Low Signed Byte
VX	100002CE			V	vupklsh	Vector Unpack Low Signed Halfword
VX	100004C4			V	vxor	Vector Logical XOR
X	7C00007C			WT	wait	Wait
X	7C000106		P	E	wrtee	Write MSR External Enable
X	7C000146		P	E	wrteei	Write MSR External Enable Immediate
X	7C000278	SR		B	xor[.]	XOR

[1] See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.

[2] For 16-bit instructions, the “Opcode” column represents the 16-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits; dashes are used following the opcode to indicate the form is a 16-bit instruction. For 32-bit instructions, the “Opcode” column represents the 32-bit hexadecimal instruction encoding with the opcode, extended opcode, and other fields with fixed values in the corresponding fields in the instruction, and with 0...s in bit positions which are not opcode, extended opcode or fixed value bits.

Appendix B. VLE Instruction Set Sorted by Opcode

This appendix lists all the instructions available in VLE mode in the Power ISA , in order by opcode. Opcodes that are not defined below are treated as illegal by category VLE.

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
C	0000----			VLE	se_illegal	Illegal
C	0001----			VLE	se_isync	Instruction Synchronize
C	0002----			VLE	se_sc	System Call
C	0004----			VLE	se_blr[l]	Branch to Link Register [and Link]
C	0006----			VLE	se_bctr[l]	Branch to Count Register [and Link]
C	0008----		P	VLE	se_rfi	Return From Interrupt
C	0009----		H	VLE	se_rfci	Return From Critical Interrupt
C	000A----		H	VLE	se_rfdi	Return From Debug Interrupt
C	000B----		H	VLE	se_rfmci	Return From Machine Check Interrupt
C	000C----		P	VLE, E.HV	se_rfgi	Return From Guest Interrupt
R	0020----			VLE	se_not	NOT Short Form
R	0030----			VLE	se_neg	Negate Short Form
R	0080----			VLE	se_mflr	Move From Link Register
R	0090----			VLE	se_mtlr	Move To Link Register
R	00A0----			VLE	se_mfctr	Move From Count Register
R	00B0----			VLE	se_mtctr	Move To Count Register
R	00C0----			VLE	se_extzb	Extend Zero Byte
R	00D0----			VLE	se_extsb	Extend Sign Byte Short Form
R	00E0----			VLE	se_extzh	Extend Zero Halfword
R	00F0----			VLE	se_extsh	Extend Sign Halfword Short Form
RR	0100----			VLE	se_mr	Move Register
RR	0200----			VLE	se_mtar	Move To Alternate Register
RR	0300----			VLE	se_mfar	Move from Alternate Register
RR	0400----			VLE	se_add	Add Short Form
RR	0500----			VLE	se_mullw	Multiply Low Word Short Form
RR	0600----			VLE	se_sub	Subtract
RR	0700----			VLE	se_subf	Subtract From Short Form
RR	0C00----			VLE	se_cmp	Compare Word
RR	0D00----			VLE	se_cmpl	Compare Logical Word
RR	0E00----			VLE	se_cmph	Compare Halfword Short Form
RR	0F00----			VLE	se_cmphl	Compare Halfword Logical Short Form
VX	10000000			V	vaddubm	Vector Add Unsigned Byte Modulo
VX	10000002			V	vmaxub	Vector Maximum Unsigned Byte
VX	10000004			V	vrlb	Vector Rotate Left Byte
VC	10000006			V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VX	10000008			V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	1000000A			V	vaddfp	Vector Add Single-Precision
VX	1000000C			V	vmrghb	Vector Merge High Byte
VX	1000000E			V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
X	10000010	SR		LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
XO	10000018	SR		LMA	machhwu[o][.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
VA	10000020			V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VA	10000021			V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VA	10000022			V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VA	10000024			V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	10000025			V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	10000026			V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	10000027			V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VA	10000028			V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	10000029			V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	1000002A			V	vsel	Vector Select
VA	1000002B			V	vperm	Vector Permute
VA	1000002C			V	vsldoi	Vector Shift Left Double by Octet Immediate
VA	1000002E			V	vmaddfp	Vector Multiply-Add Single-Precision
VA	1000002F			V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	10000040			V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	10000042			V	vmaxuh	Vector Maximum Unsigned Halfword
VX	10000044			V	vrlh	Vector Rotate Left Halfword
VC	10000046			V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VX	10000048			V	vmulouh	Vector Multiply Odd Unsigned Halfword
VX	1000004A			V	vsubfp	Vector Subtract Single-Precision
VX	1000004C			V	vmrghh	Vector Merge High Halfword
VX	1000004E			V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
X	10000050	SR		LMA	mulhhw[.]	Multiply High Halfword to Word Signed
XO	10000058	SR		LMA	machhw[o][.]	Multiply Accumulate High Halfword to Word Modulo Signed
XO	1000005C	SR		LMA	nmachhw[o][.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
VX	10000080			V	vadduwm	Vector Add Unsigned Word Modulo
VX	10000082			V	vmaxuw	Vector Maximum Unsigned Word
VX	10000084			V	vrlw	Vector Rotate Left Word
VC	10000086			V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VX	1000008C			V	vmrghw	Vector Merge High Word
VX	1000008E			V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
XO	10000098	SR		LMA	machhwsu[o][.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
VC	100000C6			V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VX	100000CE			V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
XO	100000D8	SR		LMA	machhws[o][.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	100000DC	SR		LMA	nmachhws[o][.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
VX	10000102			V	vmaxsb	Vector Maximum Signed Byte
VX	10000104			V	vslb	Vector Shift Left Byte
VX	10000108			V	vmulosb	Vector Multiply Odd Signed Byte
VX	1000010A			V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	1000010C			V	vmrglb	Vector Merge Low Byte
VX	1000010E			V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
X	10000110	SR		LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
XO	10000118	SR		LMA	macchwu[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
VX	10000142			V	vmaxsh	Vector Maximum Signed Halfword
VX	10000144			V	vslh	Vector Shift Left Halfword
VX	10000148			V	vmulosh	Vector Multiply Odd Signed Halfword
VX	1000014A			V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VX	1000014C			V	vmrglh	Vector Merge Low Halfword
VX	1000014E			V	vpkswus	Vector Pack Signed Word Unsigned Saturate
X	10000150	SR		LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
XO	10000158	SR		LMA	macchw[o][.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	1000015C	SR		LMA	nmacchw[o][.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VX	10000180			V	vaddcuw	Vector Add and write Carry-out Unsigned Word
VX	10000182			V	vmaxsw	Vector Maximum Signed Word
VX	10000184			V	vslw	Vector Shift Left Word
VX	1000018A			V	vexptefp	Vector 2 Raised to the Exponent Estimate Floating-Point
VX	1000018C			V	vmrglw	Vector Merge Low Word
VX	1000018E			V	vpkshss	Vector Pack Signed Halfword Signed Saturate
XO	10000198	SR		LMA	macchwsu[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
VX	100001C4			V	vsl	Vector Shift Left
VC	100001C6			V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VX	100001CA			V	vlogefp	Vector Log Base 2 Estimate Floating-Point
VX	100001CE			V	vpkswss	Vector Pack Signed Word Signed Saturate
XO	100001D8	SR		LMA	macchws[o][.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	100001DC	SR		LMA	nmacchws[o][.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
EVX	10000200			SP	evaddw	Vector Add Word
VX	10000200			V	vaddubs	Vector Add Unsigned Byte Saturate
EVX	10000202			SP	evaddiw	Vector Add Immediate Word
VX	10000202			V	vminub	Vector Minimum Unsigned Byte
EVX	10000204			SP	evsubfw	Vector Subtract from Word
VX	10000204			V	vsrb	Vector Shift Right Byte
EVX	10000206			SP	evsubifw	Vector Subtract Immediate from Word
VC	10000206			V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
EVX	10000208			SP	evabs	Vector Absolute Value
VX	10000208			V	vmuleub	Vector Multiply Even Unsigned Byte
EVX	10000209			SP	evneg	Vector Negate
EVX	1000020A			SP	evextsb	Vector Extend Sign Byte
VX	1000020A			V	vrfin	Vector Round to Single-Precision Integer Nearest
EVX	1000020B			SP	evextsh	Vector Extend Sign Halfword
EVX	1000020C			SP	evrndw	Vector Round Word
VX	1000020C			V	vspltb	Vector Splat Byte
EVX	1000020D			SP	evcntlzw	Vector Count Leading Zeros Word
EVX	1000020E			SP	evcntlsw	Vector Count Leading Signed Bits Word
VX	1000020E			V	vupkhsb	Vector Unpack High Signed Byte
EVX	1000020F			SP	brinc	Bit Reversed Increment
EVX	10000211			SP	evand	Vector AND
EVX	10000212			SP	evandc	Vector AND with Complement
EVX	10000216			SP	evxor	Vector XOR
EVX	10000217			SP	evor	Vector OR
EVX	10000218			SP	evnor	Vector NOR
EVX	10000219			SP	eveqv	Vector Equivalent
EVX	1000021B			SP	evorc	Vector OR with Complement
EVX	1000021E			SP	evnand	Vector NAND
EVX	10000220			SP	evsrwu	Vector Shift Right Word Unsigned
EVX	10000221			SP	evsrws	Vector Shift Right Word Signed
EVX	10000222			SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	10000223			SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	10000224			SP	evslw	Vector Shift Left Word
EVX	10000226			SP	evslwi	Vector Shift Left Word Immediate
EVX	10000228			SP	evrlw	Vector Rotate Left Word
EVX	10000229			SP	evsplati	Vector Splat Immediate
EVX	1000022A			SP	evrlwi	Vector Rotate Left Word Immediate
EVX	1000022B			SP	evsplatfi	Vector Splat Fractional Immediate
EVX	1000022C			SP	evmergehi	Vector Merge High
EVX	1000022D			SP	evmergelo	Vector Merge Low
EVX	1000022E			SP	evmergehilo	Vector Merge High/Low
EVX	1000022F			SP	evmergelohi	Vector Merge Low/High
EVX	10000230			SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	10000231			SP	evcmpgts	Vector Compare Greater Than Signed

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000232			SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	10000233			SP	evcmplts	Vector Compare Less Than Signed
EVX	10000234			SP	evcmpeq	Vector Compare Equal
VX	10000240			V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	10000242			V	vminuh	Vector Minimum Unsigned Halfword
VX	10000244			V	vsrh	Vector Shift Right Halfword
VC	10000246			V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VX	10000248			V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	1000024A			V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	1000024C			V	vsplth	Vector Splat Halfword
VX	1000024E			V	vupkhsh	Vector Unpack High Signed Halfword
EVS	10000278			SP	evsel	Vector Select
EVX	10000280			SP.FV	evfsadd	Vector Floating-Point Single-Precision Add
VX	10000280			V	vadduws	Vector Add Unsigned Word Saturate
EVX	10000281			SP.FV	evfssub	Vector Floating-Point Single-Precision Subtract
VX	10000282			V	vminuw	Vector Minimum Unsigned Word
EVX	10000284			SP.FV	evfsabs	Vector Floating-Point Single-Precision Absolute Value
VX	10000284			V	vsrw	Vector Shift Right Word
EVX	10000285			SP.FV	evfsnabs	Vector Floating-Point Single-Precision Negative Absolute Value
EVX	10000286			SP.FV	evfsneg	Vector Floating-Point Single-Precision Negate
VC	10000286			V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
EVX	10000288			SP.FV	evfsmul	Vector Floating-Point Single-Precision Multiply
EVX	10000289			SP.FV	evfsdiv	Vector Floating-Point Single-Precision Divide
VX	1000028A			V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
EVX	1000028C			SP.FV	evfscmpgt	Vector Floating-Point Single-Precision Compare Greater Than
VX	1000028C			V	vspltw	Vector Splat Word
EVX	1000028D			SP.FV	evfscmplt	Vector Floating-Point Single-Precision Compare Less Than
EVX	1000028E			SP.FV	evfscmpeq	Vector Floating-Point Single-Precision Compare Equal
VX	1000028E			V	vupklsb	Vector Unpack Low Signed Byte
EVX	10000290			SP.FV	evfscfui	Vector Convert Floating-Point Single-Precision from Unsigned Integer
EVX	10000291			SP.FV	evfscfsi	Vector Convert Floating-Point Single-Precision from Signed Integer
EVX	10000292			SP.FV	evfscfuf	Vector Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	10000293			SP.FV	evfscfsf	Vector Convert Floating-Point Single-Precision from Signed Fraction
EVX	10000294			SP.FV	evfsctui	Vector Convert Floating-Point Single-Precision to Unsigned Integer
EVX	10000295			SP.FV	evfsctsi	Vector Convert Floating-Point Single-Precision to Signed Integer
EVX	10000296			SP.FV	evfsctuf	Vector Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	10000297			SP.FV	evfsctsf	Vector Convert Floating-Point Single-Precision to Signed Fraction
EVX	10000298			SP.FV	evfsctuiz	Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero
EVX	1000029A			SP.FV	evfsctsiz	Vector Convert Floating-Point Single-Precision to Signed Integer with Round Toward Zero
EVX	1000029C			SP.FV	evfststgt	Vector Floating-Point Single-Precision Test Greater Than
EVX	1000029D			SP.FV	evfststlt	Vector Floating-Point Single-Precision Test Less Than
EVX	1000029E			SP.FV	evfststeq	Vector Floating-Point Single-Precision Test Equal
EVX	100002C0			SP.FS	efsadd	Floating-Point Single-Precision Add
EVX	100002C1			SP.FS	efssub	Floating-Point Single-Precision Subtract
EVX	100002C4			SP.FS	efsabs	Floating-Point Single-Precision Absolute Value
VX	100002C4			V	vsr	Vector Shift Right
EVX	100002C5			SP.FS	efsnabs	Floating-Point Single-Precision Negative Absolute Value

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002C6			SP.FS	efsneg	Floating-Point Single-Precision Negate
VC	100002C6			V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
EVX	100002C8			SP.FS	efsmul	Floating-Point Single-Precision Multiply
EVX	100002C9			SP.FS	efsdiv	Floating-Point Single-Precision Divide
VX	100002CA			V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
EVX	100002CC			SP.FS	efscmpgt	Floating-Point Single-Precision Compare Greater Than
EVX	100002CD			SP.FS	efscmplt	Floating-Point Single-Precision Compare Less Than
EVX	100002CE			SP.FS	efscmpeq	Floating-Point Single-Precision Compare Equal
VX	100002CE			V	vupklsh	Vector Unpack Low Signed Halfword
EVX	100002CF			SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	100002D0			SP.FS	efscfui	Convert Floating-Point Single-Precision from Unsigned Integer
EVX	100002D1			SP.FS	efscfsi	Convert Floating-Point Single-Precision from Signed Integer
EVX	100002D2			SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	100002D3			SP.FS	efscfsf	Convert Floating-Point Single-Precision from Signed Fraction
EVX	100002D4			SP.FS	efsctui	Convert Floating-Point Single-Precision to Unsigned Integer
EVX	100002D5			SP.FS	efsctsi	Convert Floating-Point Single-Precision to Signed Integer
EVX	100002D6			SP.FS	efsctuf	Convert Floating-Point Single-Precision to Unsigned Fraction
EVX	100002D7			SP.FS	efsctsf	Convert Floating-Point Single-Precision to Signed Fraction
EVX	100002D8			SP.FS	efsctuiz	Convert Floating-Point Single-Precision to Unsigned Integer with Round Towards Zero
EVX	100002DA			SP.FS	efsctsiz	Convert Floating-Point Single-Precision to Signed Integer with Round Towards Zero
EVX	100002DC			SP.FS	efststgt	Floating-Point Single-Precision Test Greater Than
EVX	100002DD			SP.FS	efststlt	Floating-Point Single-Precision Test Less Than
EVX	100002DE			SP.FS	efststeq	Floating-Point Single-Precision Test Equal
EVX	100002E0			SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	100002E1			SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	100002E2			SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	100002E3			SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	100002E4			SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	100002E5			SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	100002E6			SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	100002E8			SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	100002E9			SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	100002EA			SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	100002EB			SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	100002ED			SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	100002EC			SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	100002EE			SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	100002EF			SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	100002F0			SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	100002F1			SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	100002F2			SP.FS	efscfuf	Convert Floating-Point Single-Precision from Unsigned Fraction
EVX	100002F3			SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100002F4			SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	100002F5			SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	100002F6			SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	100002F7			SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	100002F8			SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	100002FA			SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	100002FC			SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	100002FD			SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	100002FE			SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	10000300			SP	evlddx	Vector Load Double Word into Double Word Indexed
VX	10000300			V	vaddsbs	Vector Add Signed Byte Saturate
EVX	10000301			SP	evldd	Vector Load Double Word into Double Word
EVX	10000302			SP	evldwx	Vector Load Double Word into Two Words Indexed
VX	10000302			V	vminsb	Vector Minimum Signed Byte
EVX	10000303			SP	evldw	Vector Load Double Word into Two Words
EVX	10000304			SP	evldhx	Vector Load Double Word into Four Halfwords Indexed
VX	10000304			V	vsrab	Vector Shift Right Algebraic Word
EVX	10000305			SP	evldh	Vector Load Double Word into Four Halfwords
VC	10000306			V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
EVX	10000308			SP	evlhhesplatx	Vector Load Halfword into Halfwords Even and Splat Indexed
VX	10000308			V	vmulesb	Vector Multiply Even Signed Byte
EVX	10000309			SP	evlhhesplat	Vector Load Halfword into Halfwords Even and Splat
VX	1000030A			V	vcfux	Vector Convert From
EVX	1000030C			SP	evlhhousplatx	Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed
VX	1000030C			V	vspltisb	Vector Splat Immediate Signed Byte
EVX	1000030D			SP	evlhhousplat	Vector Load Halfword into Halfword Odd Unsigned and Splat
EVX	1000030E			SP	evlhhossplatx	Vector Load Halfword into Halfword Odd Signed and Splat Indexed
VX	1000030E			V	vpkpx	Vector Pack Pixel
EVX	1000030F			SP	evlhhossplat	Vector Load Halfword into Halfword Odd and Splat
X	10000310	SR		LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
EVX	10000311			SP	evlwhe	Vector Load Word into Two Halfwords Even
EVX	10000314			SP	evlwhoux	Vector Load Word into Two Halfwords Odd Unsigned Indexed (zero-extended)
EVX	10000315			SP	evlwhou	Vector Load Word into Two Halfwords Odd Unsigned (zero-extended)
EVX	10000316			SP	evlwhosx	Vector Load Word into Two Halfwords Odd Signed Indexed (with sign extension)
EVX	10000317			SP	evlwhos	Vector Load Word into Two Halfwords Odd Signed (with sign extension)
EVX	10000318			SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
XO	10000318	SR		LMA	maclhwu[o][.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
EVX	10000319			SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	1000031C			SP	evlwhsplatx	Vector Load Word into Two Halfwords and Splat Indexed
EVX	1000031D			SP	evlwhsplat	Vector Load Word into Two Halfwords and Splat
EVX	10000320			SP	evstddx	Vector Store Doubleword of Doubleword Indexed
EVX	10000321			SP	evstdd	Vector Store Double of Double
EVX	10000322			SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	10000323			SP	evstdw	Vector Store Double of Two Words
EVX	10000324			SP	evstdhx	Vector Store Double of Four Halfwords Indexed
EVX	10000325			SP	evstdh	Vector Store Double of Four Halfwords

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000330			SP	evstwhex	Vector Store Word of Two Halfwords from Even Indexed
EVX	10000331			SP	evstwhe	Vector Store Word of Two Halfwords from Even
EVX	10000334			SP	evstwhox	Vector Store Word of Two Halfwords from Odd Indexed
EVX	10000335			SP	evstwho	Vector Store Word of Two Halfwords from Odd
EVX	10000338			SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	10000339			SP	evstwwe	Vector Store Word of Word from Even
EVX	1000033C			SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	1000033D			SP	evstwwo	Vector Store Word of Word from Odd
VX	10000340			V	vaddshs	Vector Add Signed Halfword Saturate
VX	10000342			V	vminsh	Vector Minimum Signed Halfword
VX	10000344			V	vsrah	Vector Shift Right Algebraic Halfword
VC	10000346			V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VX	10000348			V	vmulesh	Vector Multiply Even Signed Halfword
VX	1000034A			V	vcfsx	Vector Convert From Signed Fixed-Point Word
VX	1000034C			V	vspltish	Vector Splat Immediate Signed Halfword
VX	1000034E			V	vupkhpx	Vector Unpack High Pixel
X	10000350	SR		LMA	mullhw[.]	Multiply Low Halfword to Word Signed
XO	10000358	SR		LMA	maclhw[o][.]	Multiply Accumulate Low Halfword to Word Modulo Signed
XO	1000035C	SR		LMA	nmaclhw[o][.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
VX	10000380			V	vaddsws	Vector Add Signed Word Saturate
VX	10000382			V	vminsw	Vector Minimum Signed Word
VX	10000384			V	vsraw	Vector Shift Right Algebraic Word
VC	10000386			V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VX	1000038A			V	vctuxs	Vector Convert To Unsigned Fixed-Point Word Saturate
VX	1000038C			V	vspltisw	Vector Splat Immediate Signed Word
XO	10000398	SR		LMA	maclhwsu[o][.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
VC	100003C6			V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VX	100003CA			V	vctsxs	Vector Convert To Signed Fixed-Point Word Saturate
VX	100003CE			V	vupklpx	Vector Unpack Low Pixel
XO	100003D8	SR		LMA	maclhws[o][.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	100003DC	SR		LMA	nmaclhws[o][.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
VX	10000400			V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	10000402			V	vavgub	Vector Average Unsigned Byte
EVX	10000403			SP	evmhessf	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional
VX	10000404			V	vand	Vector Logical AND
EVX	10000407			SP	evmhossf	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional
EVX	10000408			SP	evmheumi	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer
EVX	10000409			SP	evmhesmi	Vector Multiply Halfwords, Even, Signed, Modulo, Integer
VX	1000040A			V	vmaxfp	Vector Maximum Single-Precision
EVX	1000040B			SP	evmhesmf	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional
EVX	1000040C			SP	evmhoumi	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer
VX	1000040C			V	vslo	Vector Shift Left by Octet
EVX	1000040D			SP	evmhosmi	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer
EVX	1000040F			SP	evmhosmf	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional
EVX	10000423			SP	evmhessfa	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional to Accumulator
EVX	10000427			SP	evmhossfa	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional to Accumulator

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	10000428			SP	evmheumia	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer to Accumulator
EVX	10000429			SP	evmhesmia	Vector Multiply Halfwords, Even, Signed, Modulo, Integer to Accumulator
EVX	1000042B			SP	evmhesmfa	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional to Accumulator
EVX	1000042C			SP	evmhoumia	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	1000042D			SP	evmhosmia	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer to Accumulator
EVX	1000042F			SP	evmhosmfa	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional to Accumulator
VX	10000440			V	vsubuhm	Vector Subtract Unsigned Halfword Modulo
VX	10000442			V	vavguh	Vector Average Unsigned Halfword
VX	10000444			V	vandc	Vector Logical AND with Complement
EVX	10000447			SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	10000448			SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
VX	1000044A			V	vminfp	Vector Minimum Single-Precision
EVX	1000044C			SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
VX	1000044C			V	vsro	Vector Shift Right by Octet
EVX	1000044D			SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	1000044F			SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	10000453			SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	10000458			SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	10000459			SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	1000045B			SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	10000467			SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	10000468			SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	1000046C			SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	1000046D			SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	1000046F			SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	10000473			SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	10000478			SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	10000479			SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	1000047B			SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
VX	10000480			V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	10000482			V	vavguw	Vector Average Unsigned Word
VX	10000484			V	vor	Vector Logical OR
EVX	100004C0			SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	100004C1			SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	100004C2			SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	100004C3			SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	100004C4			SP	evmra	Initialize Accumulator
VX	100004C4			V	vxor	Vector Logical XOR
EVX	100004C6			SP	evdivws	Vector Divide Word Signed
EVX	100004C7			SP	evdivwu	Vector Divide Word Unsigned
EVX	100004C8			SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	100004C9			SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	100004CA			SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	100004CB			SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	10000500			SP	evmheusiaaw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate into Words
EVX	10000501			SP	evmhessiaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate into Words
VX	10000502			V	vavgsb	Vector Average Signed Byte
EVX	10000503			SP	evmhessfaaw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000504			SP	evmhousiaaw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate into Words
VX	10000504			V	vnor	Vector Logical NOR
EVX	10000505			SP	evmhossiaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	10000507			SP	evmhossfaaw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	10000508			SP	evmheumiaaw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	10000509			SP	evmhesmiaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	1000050B			SP	evmhesmfaaw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	1000050C			SP	evmhoumiaaw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	1000050D			SP	evmhosmiaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	1000050F			SP	evmhosmfaaw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	10000528			SP	evmhegumiaa	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	10000529			SP	evmhegsmiaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	1000052B			SP	evmhegsmfaa	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	1000052C			SP	evmhogumiaa	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	1000052D			SP	evmhogsmiaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer, and Accumulate
EVX	1000052F			SP	evmhogsmfaa	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	10000540			SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate into Words
EVX	10000541			SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate into Words
VX	10000542			V	vavgsh	Vector Average Signed Halfword
EVX	10000548			SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate into Words
EVX	10000549			SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate into Words
EVX	10000553			SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	10000558			SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	10000559			SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
EVX	1000055B			SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	10000580			SP	evmheusianw	Vector Multiply Halfwords, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
VX	10000580			V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
EVX	10000581			SP	evmhessianw	Vector Multiply Halfwords, Even, Signed, Saturate, Integer and Accumulate Negative into Words
VX	10000582			V	vavgsw	Vector Average Signed Word
EVX	10000583			SP	evmhessfanw	Vector Multiply Halfwords, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000584			SP	evmhousianw	Vector Multiply Halfwords, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	10000585			SP	evmhossianw	Vector Multiply Halfwords, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	10000587			SP	evmhossfanw	Vector Multiply Halfwords, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	10000588			SP	evmheumianw	Vector Multiply Halfwords, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	10000589			SP	evmhesmianw	Vector Multiply Halfwords, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	1000058B			SP	evmhesmfanw	Vector Multiply Halfwords, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	1000058C			SP	evmhoumianw	Vector Multiply Halfwords, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	1000058D			SP	evmhosmianw	Vector Multiply Halfwords, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	1000058F			SP	evmhosmfanw	Vector Multiply Halfwords, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	100005A8			SP	evmhegumian	Vector Multiply Halfwords, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	100005A9			SP	evmhegsmian	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	100005AB			SP	evmhegsmfan	Vector Multiply Halfwords, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	100005AC			SP	evmhogumian	Vector Multiply Halfwords, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	100005AD			SP	evmhogsmian	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	100005AF			SP	evmhogsmfan	Vector Multiply Halfwords, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	100005C0			SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	100005C1			SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
EVX	100005C8			SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words
EVX	100005C9			SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
EVX	100005D3			SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	100005D8			SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	100005D9			SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	100005DB			SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
VX	10000600			V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	10000604			V	mfvscr	Move From VSCR
VX	10000608			V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate

Form	Opcode (hexadecimal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
VX	10000640			V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	10000644			V	mtvscr	Move To VSCR
VX	10000648			V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	10000680			V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	10000688			V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	10000700			V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	10000708			V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	10000740			V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	10000780			V	vsubsws	Vector Subtract Signed Word Saturate
VX	10000788			V	vsumsws	Vector Sum across Signed Word Saturate
D8	18000000			VLE	e_lbzu	Load Byte and Zero with Update
D8	18000100			VLE	e_lhzu	Load Halfword and Zero with Update
D8	18000200			VLE	e_lwzu	Load Word and Zero with Update
D8	18000300			VLE	e_lhau	Load Halfword Algebraic with Update
D8	18000400			VLE	e_stbu	Store Byte with Update
D8	18000500			VLE	e_sthu	Store Halfword with Update
D8	18000600			VLE	e_stwu	Store Word with Update
D8	18000800			VLE	e_lmw	Load Multiple Word
D8	18000900			VLE	e_stmw	Store Multiple Word
SCI8	18008000	SR		VLE	e_addi[.]	Add Scaled Immediate
SCI8	18009000	SR		VLE	e_addic[.]	Add Scaled Immediate Carrying
SCI8	1800A000			VLE	e_mulli	Multiply Low Scaled Immediate
SCI8	1800A800			VLE	e_cmpi	Compare Scaled Immediate Word
SCI8	1800B000	SR		VLE	e_subfic[.]	Subtract From Scaled Immediate Carrying
SCI8	1800C000	SR		VLE	e_andi[.]	AND Scaled Immediate
SCI8	1800D000	SR		VLE	e_ori[.]	OR Scaled Immediate
SCI8	1800E000	SR		VLE	e_xori[.]	XOR Scaled Immediate
SCI8	1880A800			VLE	e_cmpli	Compare Logical Scaled Immediate Word
D	1C000000			VLE	e_add16i	Add Immediate
OIM5	2000----			VLE	se_addi	Add Immediate Short Form
OIM5	2200----			VLE	se_cmpli	Compare Logical Immediate Word
OIM5	2400----	SR		VLE	se_subi[.]	Subtract Immediate
IM5	2A00----			VLE	se_cmpi	Compare Immediate Word Short Form
IM5	2C00----			VLE	se_bmaski	Bit Mask Generate Immediate
IM5	2E00----			VLE	se_andi	AND Immediate Short Form
D	30000000			VLE	e_lbz	Load Byte and Zero
D	34000000			VLE	e_stb	Store Byte
D	38000000			VLE	e_lha	Load Halfword Algebraic
RR	4000----			VLE	se_srw	Shift Right Word
RR	4100----			VLE	se_sraw	Shift Right Algebraic Word
RR	4200----			VLE	se_slw	Shift Left Word
RR	4400----			VLE	se_or	OR Short Form
RR	4500----			VLE	se_andc	AND with Complement Short Form
RR	4600----	SR		VLE	se_and[.]	AND Short Form
IM7	4800----			VLE	se_li	Load Immediate Short Form
D	50000000			VLE	e_lwz	Load Word and Zero
D	54000000			VLE	e_stw	Store Word
D	58000000			VLE	e_lhz	Load Halfword and Zero
D	5C000000			VLE	e_sth	Store Halfword
IM5	6000----			VLE	se_bclri	Bit Clear Immediate
IM5	6200----			VLE	se_bgeni	Bit Generate Immediate
IM5	6400----			VLE	se_bseti	Bit Set Immediate
IM5	6600----			VLE	se_btsti	Bit Test Immediate
IM5	6800----			VLE	se_srwi	Shift Right Word Immediate Short Form
IM5	6A00----			VLE	se_srawi	Shift Right Algebraic Word Immediate
IM5	6C00----			VLE	se_slwi	Shift Left Word Immediate Short Form
LI20	70000000			VLE	e_li	Load Immediate
I16A	70008800	SR		VLE	e_add2i.	Add (2 operand) Immediate and Record
I16A	70009000			VLE	e_add2is	Add (2 operand) Immediate Shifted
I16A	70009800			VLE	e_cmp16i	Compare Immediate Word
I16A	7000A000			VLE	e_mull2i	Multiply (2 operand) Low Immediate

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
I16A	7000A800			VLE	e_cmpl16i	Compare Logical Immediate Word
I16A	7000B000			VLE	e_cmph16i	Compare Halfword Immediate
I16A	7000B800			VLE	e_cmphl16i	Compare Halfword Logical Immediate
I16L	7000C000			VLE	e_or2i	OR (two operand) Immediate
I16L	7000C800	SR		VLE	e_and2i.	AND (two operand) Immediate
I16L	7000D000			VLE	e_or2is	OR (2 operand) Immediate Shifted
I16L	7000E000			VLE	e_lis	Load Immediate Shifted
I16L	7000E800	SR		VLE	e_and2is.	AND (2 operand) Immediate Shifted
M	74000000			VLE	e_rlwimi	Rotate Left Word Immediate then Mask Insert
M	74000001			VLE	e_rlwinm	Rotate Left Word Immediate then AND with Mask
BD24	78000000			VLE	e_b[l]	Branch [and Link]
BD15	7A000000	CT		VLE	e_bc[l]	Branch Conditional [and Link]
X	7C000000			B	cmp	Compare
X	7C000008			B	tw	Trap Word
X	7C00000C			V	lvsl	Load Vector for Shift Left Indexed
X	7C00000E			V	lvebx	Load Vector Element Byte Indexed
XO	7C000010	SR		B	subfc[o][.]	Subtract From Carrying
XO	7C000012	SR		64	mulhdu[.]	Multiply High Doubleword Unsigned
XO	7C000014	SR		B	addc[o][.]	Add Carrying
XO	7C000016	SR		B	mulhwu[.]	Multiply High Word Unsigned
X	7C00001C			VLE	e_cmph	Compare Halfword
A	7C00001E			B	isel	Integer Select
XL	7C000020			VLE	e_mcrf	Move CR Field
XFX	7C000026			B	mfcr	Move From Condition Register
X	7C000028			B	lwarx	Load Word And Reserve Indexed
X	7C00002A			64	ldx	Load Doubleword Indexed
X	7C00002C			B	icbt	Instruction Cache Block Touch
X	7C00002E			B	lwzx	Load Word and Zero Indexed
X	7C000030	SR		B	slw[.]	Shift Left Word
X	7C000034	SR		B	cntlzw[.]	Count Leading Zeros Word
X	7C000036	SR		64	sld[.]	Shift Left Doubleword
X	7C000038	SR		B	and[.]	AND
X	7C00003A		P	E.PD;64	ldepx	Load Doubleword by External Process ID Indexed
X	7C00003E		P	E.PD	lwepx	Load Word by External Process ID Indexed
X	7C000040			B	cmpl	Compare Logical
XL	7C000042			VLE	e_crnor	Condition Register NOR
ESC	7C000048			VLE, E.HV	e_sc	System Call
X	7C00004C			V	lvsr	Load Vector for Shift Right Indexed
X	7C00004E			V	lvehx	Load Vector Element Halfword Indexed
XO	7C000050	SR		B	subf[o][.]	Subtract From
X	7C00005C			VLE	e_cmphl	Compare Halfword Logical
X	7C000068			B	lbarx	Load Byte and Reserve Indexed
X	7C00006A			64	ldux	Load Doubleword with Update Indexed
X	7C00006C			B	dcbst	Data Cache Block Store
X	7C00006E			B	lwzux	Load Word and Zero with Update Indexed
X	7C000070	SR		VLE	e_slwi[.]	Shift Left Word Immediate
X	7C000074	SR		64	cntlzd[.]	Count Leading Zeros Doubleword
X	7C000078	SR		B	andc[.]	AND with Complement
X	7C00007C			WT	wait	Wait
X	7C00007E			E.PD	dcbstep	Data Cache Block Store by External PID
X	7C000088			64	td	Trap Doubleword
X	7C00008E			V	lvewx	Load Vector Element Word Indexed
XO	7C000092	SR		64	mulhd[.]	Multiply High Doubleword
XO	7C000096	SR		B	mulhw[.]	Multiply High Word
X	7C00009C	SR		LMV	dlmzb[.]	Determine Leftmost Zero Byte
X	7C0000A6		P	B	mfmsr	Move From Machine State Register
X	7C0000A8			64	ldarx	Load Doubleword And Reserve Indexed
X	7C0000AC			B	dcbf	Data Cache Block Flush
X	7C0000AE			B	lbzx	Load Byte and Zero Indexed
X	7C0000BE		P	E.PD	lbepx	Load Byte by External Process ID Indexed

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C0000CE			V	lvx	Load Vector Indexed
XO	7C0000D0	SR		B	neg[o][.]	Negate
X	7C0000E8			B	lharx	Load Halfword and Reserve Indexed
X	7C0000EE			B	lbzux	Load Byte and Zero with Update Indexed
X	7C0000F4			B	popcntb	Population Count Bytes
X	7C0000F8	SR		B	nor[.]	NOR
X	7C0000FE		P	E.PD	dcbfep	Data Cache Block Flush by External PID
XL	7C000102			VLE	e_crandc	Condition Register AND with Complement
X	7C000106		P	E	wrtee	Write MSR External Enable
X	7C00010C		M	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	7C00010E			V	stvebx	Store Vector Element Byte Indexed
XO	7C000110	SR		B	subfe[o][.]	Subtract From Extended
XO	7C000114	SR		B	adde[o][.]	Add Extended
XFX	7C000120			B	mtcrf	Move To Condition Register Fields
X	7C000124		P	E	mtmsr	Move To Machine State Register
X	7C00012A			64	stdx	Store Doubleword Indexed
X	7C00012D			B	stwcx.	Store Word Conditional Indexed
X	7C00012E			B	stwx	Store Word Indexed
X	7C00013A		P	E.PD;64	stdepx	Store Doubleword by External Process ID Indexed
X	7C00013E		P	E.PD	stwepx	Store Word by External Process ID Indexed
X	7C000146		P	E	wrteei	Write MSR External Enable Immediate
X	7C00014C		M	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	7C00014E			V	stvehx	Store Vector Element Halfword Indexed
X	7C00016A			64	stdux	Store Doubleword with Update Indexed
X	7C00016E			B	stwux	Store Word with Update Indexed
XL	7C000182			VLE	e_crxor	Condition Register XOR
X	7C00018D		M	ECL	icblq.	Instruction Cache Block Lock Query
X	7C00018E			V	stvewx	Store Vector Element Word Indexed
XO	7C000190	SR		B	subfze[o][.]	Subtract From Zero Extended
XO	7C000194	SR		B	addze[o][.]	Add to Zero Extended
X	7C00019C		H	E.PC	msgsnd	Message Send
X	7C0001AD			64	stdcx.	Store Doubleword Conditional Indexed
X	7C0001AE			B	stbx	Store Byte Indexed
X	7C0001BE		P	E.PD	stbepx	Store Byte by External Process ID Indexed
XL	7C0001C2			VLE	e_crnand	Condition Register NAND
X	7C0001CC		M	ECL	icblc	Instruction Cache Block Lock Clear
X	7C0001CE			V	stvx	Store Vector Indexed
XO	7C0001D0	SR		B	subfme[o][.]	Subtract From Minus One Extended
XO	7C0001D2	SR		64	mulld[o][.]	Multiply Low Doubleword
XO	7C0001D4	SR		B	addme[o][.]	Add to Minus One Extended
XO	7C0001D6	SR		B	mullw[o][.]	Multiply Low Word
X	7C0001DC		H	E.PC	msgclr	Message Clear
X	7C0001EC			B	dcbtst	Data Cache Block Touch for Store
X	7C0001EE			B	stbux	Store Byte with Update Indexed
X	7C0001FE		P	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID
XL	7C000202			VLE	e_crand	Condition Register AND
X	7C000206		P	E.DC	mfdcrx	Move From Device Control Register Indexed
X	7C00020E		P	E.PD	lvepxl	Load Vector by External Process ID Indexed LRU
XO	7C000214	SR		B	add[o][.]	Add
XL	7C00021C			E.HV	ehpriv	Embedded Hypervisor Privilege
X	7C00022C			B	dcbt	Data Cache Block Touch
X	7C00022E			B	lhzx	Load Halfword and Zero Indexed
X	7C000230	SR		VLE	e_rlw[.]	Rotate Left Word
X	7C000238	SR		B	eqv[.]	Equivalent
X	7C00023E		P	E.PD	lhepx	Load Halfword by External Process ID Indexed
XL	7C000242			VLE	e_creqv	Condition Register Equivalent
X	7C000246			E.DC	mfdcrux	Move From Device Control Register User-mode Indexed
X	7C00024E		P	E.PD	lvepx	Load Vector by External Process ID Indexed
X	7C00026E			B	lhzux	Load Halfword and Zero with Update Indexed
X	7C000270	SR		VLE	e_rlwi[.]	Rotate Left Word Immediate
X	7C000278	SR		B	xor[.]	XOR

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C00027E		P	E.PD	dcbtep	Data Cache Block Touch by External PID
XFX	7C000286		P	E.DC	mfdcr	Move From Device Control Register
X	7C00028C		P	E.CD	dcread	Data Cache Read
XFX	7C00029C		O	E.PM	mfpmr	Move From Performance Monitor Register
XFX	7C0002A6		O	B	mfspr	Move From Special Purpose Register
X	7C0002AA			64	lwax	Load Word Algebraic Indexed
X	7C0002AE			B	lhax	Load Halfword Algebraic Indexed
X	7C0002CE			V	lvxl	Load Vector Indexed LRU
X	7C0002EA			64	lwaux	Load Word Algebraic with Update Indexed
X	7C0002EE			B	lhaux	Load Halfword Algebraic with Update Indexed
X	7C000306		P	E.DC	mtdcrx	Move To Device Control Register Indexed
X	7C00030C		M	ECL	dcblc	Data Cache Block Lock Clear
X	7C00032E			B	sthx	Store Halfword Indexed
X	7C000338	SR		B	orc[.]	OR with Complement
X	7C00033E		P	E.PD	sthepx	Store Halfword by External Process ID Indexed
XL	7C000342			VLE	e_crorc	Condition Register OR with Complement
X	7C000346			E.DC	mtdcrux	Move To Device Control Register User-mode Indexed
X	7C00034D		M	ECL	dcblq.	Data Cache Block Lock Query
X	7C00036E			B	sthux	Store Halfword with Update Indexed
X	7C000378	SR		B	or[.]	OR
XL	7C000382			VLE	e_cror	Condition Register OR
XFX	7C000386		P	E.DC	mtdcr	Move To Device Control Register
X	7C00038C		H	E.CI	dci	Data Cache Invalidate
XO	7C000392	SR		64	divdu[o][.]	Divide Doubleword Unsigned
XO	7C000396	SR		B	divwu[o][.]	Divide Word Unsigned
XFX	7C00039C		O	E.PM	mtpmr	Move To Performance Monitor Register
XFX	7C0003A6		O	B	mtspr	Move To Special Purpose Register
X	7C0003AC		P	E	dcbi	Data Cache Block Invalidate
X	7C0003C6			DS	dsn	Decorated Storage Notify
X	7C0003CC		M	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	7C0003CC		H	E.CD	dcread	Data Cache Read [Alternative Encoding]
X	7C0003CE			V	stvxl	Store Vector Indexed LRU
XO	7C0003D2	SR		64	divd[o][.]	Divide Doubleword
XO	7C0003D6	SR		B	divw[o][.]	Divide Word
X	7C000400			E	mcrxr	Move To Condition Register from XER
X	7C000406			DS	lbdx	Load Byte with Decoration Indexed
X	7C00042A			MA	lswx	Load String Word Indexed
X	7C00042C			B	lwbrx	Load Word Byte-Reverse Indexed
X	7C000430	SR		B	srw[.]	Shift Right Word
X	7C000436	SR		64	srd[.]	Shift Right Doubleword
X	7C000446			DS	lhdx	Load Halfword with Decoration Indexed
X	7C00046C		H	E	tlbsync	TLB Synchronize
X	7C000470	SR		VLE	e_srwi[.]	Shift Right Word Immediate
X	7C000486			DS	lwdx	Load Word with Decoration Indexed
X	7C0004AA			MA	lswi	Load String Word Immediate
X	7C0004AC			B	sync	Synchronize
X	7C0004BE		P	E.PD	lfdepx	Load Floating-Point Double by External Process ID Indexed
X	7C0004C6			DS	lddx	Load Doubleword with Decoration Indexed
X	7C000506			DS	stbdx	Store Byte with Decoration Indexed
X	7C00052A			MA	stswx	Store String Word Indexed
X	7C00052C			B	stwbrx	Store Word Byte-Reverse Indexed
X	7C000546			DS	sthdx	Store Halfword with Decoration Indexed
X	7C00056D			B	stbcx.	Store Byte Conditional Indexed
X	7C000586			DS	stwdx	Store Word with Decoration Indexed
X	7C0005AA			MA	stswi	Store String Word Immediate
X	7C0005AD			B	sthcx.	Store Halfword Conditional Indexed
X	7C0005BE		P	E.PD	stfdepx	Store Floating-Point Double by External Process ID Indexed
X	7C0005C6			DS	stddx	Store Doubleword with Decoration Indexed
X	7C0005EC			E	dcba	Data Cache Block Allocate

Form	Opcode (hexadeci-mal)[2]	Mode Dep.[1]	Priv[1]	Cat[1]	Mnemonic	Instruction
X	7C00060E		P	E.PD	stvepxl	Store Vector by External Process ID Indexed LRU
X	7C000624		H	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	7C00062C			B	lhbrx	Load Halfword Byte-Reverse Indexed
X	7C000630	SR		B	sraw[.]	Shift Right Algebraic Word
X	7C000634	SR		64	srad[.]	Shift Right Algebraic Doubleword
EVX	7C00063E		P	E.PD	evlddepx	Vector Load Doubleword into Doubleword by External Process ID Indexed
X	7C000646			DS	lfddx	Load Floating Doubleword with Decoration Indexed
X	7C00064E		P	E.PD	stvepx	Store Vector by External Process ID Indexed
X	7C000670	SR		B	srawi[.]	Shift Right Algebraic Word Immediate
XS	7C000674	SR		64	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	7C0006AC			E	mbar	Memory Barrier
X	7C000724		H	E	tlbsx	TLB Search Indexed
X	7C00072C			B	sthbrx	Store Halfword Byte-Reverse Indexed
X	7C000734	SR		B	extsh[.]	Extend Sign Halfword
EVX	7C00073E		P	E.PD	evstddepx	Vector Store Doubleword into Doubleword by External Process ID Indexed
X	7C000746			DS	stfddx	Store Floating Doubleword with Decoration Indexed
X	7C000764		H	E	tlbre	TLB Read Entry
X	7C000774	SR		B	extsb[.]	Extend Sign Byte
X	7C00078C		H	E.CI	ici	Instruction Cache Invalidate
X	7C0007A4		H	E	tlbwe	TLB Write Entry
X	7C0007AC			B	icbi	Instruction Cache Block Invalidate
X	7C0007B4	SR		64	extsw[.]	Extend Sign Word
X	7C0007BE		P	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	7C0007CC		H	E.CD	icread	Instruction Cache Read
X	7C0007EC			B	dcbz	Data Cache Block set to Zero
X	7C0007FE		P	E.PD	dcbzep	Data Cache Block set to Zero by External PID
XFX	7C100026			B	mfocrf	Move From One Condition Register Field
XFX	7C100120			B	mtocrf	Move To One Condition Register Field
SD4	8000----			VLE	se_lbz	Load Byte and Zero Short Form
SD4	9000----			VLE	se_stb	Store Byte Short Form
SD4	A000----			VLE	se_lhz	Load Halfword and Zero Short Form
SD4	B000----			VLE	se_sth	Store Halfword Short Form
SD4	C000----			VLE	se_lwz	Load Word and Zero Short Form
SD4	D000----			VLE	se_stw	Store Word Short Form
BD8	E000----			VLE	se_bc	Branch Conditional Short Form
BD8	E800----			VLE	se_b[l]	Branch [and Link]

[1] See the key to the mode dependency and privilege column below and the key to the category column in Section 1.3.5 of Book I.

[2]For 16-bit instructions, the “Opcode” column represents the 16-bit hexadecimal instruction encoding with the opcode and extended opcode in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode bits; dashes are used following the opcode to indicate the form is a 16-bit instruction. For 32-bit instructions, the "Opcode" column represents the 32-bit hexadecimal instruction encoding with the opcode, extended opcode, and other fields with fixed values in the corresponding fields in the instruction, and with 0’s in bit positions which are not opcode, extended opcode or fixed value bits."

Mode Dependency and Privilege Abbreviations

Except as described below and in Section 1.10.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

Mode Dep.	Description
CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
SR	The setting of status registers (such as XER and CR0) is mode-dependent.

Mode Dep.	Description
32	The instruction must be executed only in 32-bit mode.
64	The instruction must be executed only in 64-bit mode.

Key to Privilege Column

Priv.	Description
P	Denotes a privileged instruction.
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for ***mtspr***), depending on the SPR or PMR number.
M	Denotes an instruction that is treated as privileged or nonprivileged, depending on the value of the UCLE bit of the MSR.
H	Denotes an instruction that can be executed only in hypervisor state.

Appendices:

Power ISA Book I-III Appendices

Appendix A. Incompatibilities with the POWER Architecture

This appendix identifies the known incompatibilities that must be managed in the migration from the POWER Architecture to the Power ISA. Some of the incompatibilities can, at least in principle, be detected by the processor, which could trap and let software simulate the POWER operation. Others cannot be detected by the processor even in principle.

In general, the incompatibilities identified here are those that affect a POWER application program. Incompatibilities for instructions that can be used only by POWER operating system programs are not necessarily discussed. Discussion of incompatibilities that pertain only to operating system programs assumes the Server environment (because there is no need for POWER operating system programs to run in the Embedded environment).

A.1 New Instructions, Formerly Privileged Instructions

Instructions new to Power ISA typically use opcode values (including extended opcode) that are illegal in POWER. A few instructions that are privileged in POWER (e.g., ***dclz***, called ***dcbz*** in Power ISA) have been made nonprivileged in Power ISA. Any POWER program that executes one of these now-valid or now-nonprivileged instructions, expecting to cause the system illegal instruction error handler or the system privileged instruction error handler to be invoked, will not execute correctly on Power ISA.

A.2 Newly Privileged Instructions

The following instructions are nonprivileged in POWER but privileged in Power ISA.

mfmsr
mfsr

A.3 Reserved Fields in Instructions

These fields are shown with "/"s in the instruction layouts. In both POWER and Power ISA these fields are ignored by the processor. The Power ISA states that these fields must contain zero. The POWER Architecture lacks such a statement, but it is expected that essentially all POWER programs contain zero in these fields.

In several cases the Power ISA assumes that reserved fields in POWER instructions indeed contain zero. The cases include the following.

- ***bclr***[***l***] and ***bcctr***[***l***] assume that bits 19:20 in the POWER instructions contain zero.
- ***cmpi***, ***cmp***, ***cmpli***, and ***cmpl*** assume that bit 10 in the POWER instructions contains zero.
- ***mtspr*** and ***mfspr*** assume that bits 16:20 in the POWER instructions contain zero.
- ***mtcrf*** and ***mfcr*** assume that bit 11 in the POWER instructions is contains zero.
- *Synchronize* assumes that bits 9:10 in the POWER instruction (***dcs***) contain zero. (This assumption provides compatibility for application programs, but not necessarily for operating system programs; see Section A.22.)
- ***mtmsr*** assumes that bit 15 in the POWER instruction contains zero.

A.4 Reserved Bits in Registers

Both POWER and Power ISA permit software to write any value to these bits. However in POWER reading such a bit always returns 0, while in Power ISA reading it may return either 0 or the value that was last written to it.

A.5 Alignment Check

The POWER MSR AL bit (bit 24) is no longer supported; the corresponding Power ISA MSR bit, bit 56, is reserved. The low-order bits of the EA are always used. (Notice that the value 0 — the normal value for a reserved bit —- means "ignore the low-order EA bits" in

POWER, and the value 1 means “use the low-order EA bits”.) POWER-compatible operating system code will probably write the value 1 to this bit.

A.6 Condition Register

The following instructions specify a field in the CR explicitly (via the BF field) and also, in POWER, use bit 31 as the Record bit. In Power ISA, bit 31 is a reserved field for these instructions and is ignored by the processor. In POWER, if bit 31 contains 1 the instructions execute normally (i.e., as if the bit contained 0) except as follows:

cmp	CR0 is undefined if Rc=1 and BF≠0
cmpl	CR0 is undefined if Rc=1 and BF≠0
mcrxr	CR0 is undefined if Rc=1 and BF≠0
fcmpu	CR1 is undefined if Rc=1
fcmpo	CR1 is undefined if Rc=1
mcrfs	CR1 is undefined if Rc=1 and BF≠1

A.7 LK and Rc Bits

For the instructions listed below, if bit 31 (LK or Rc bit in POWER) contains 1, in POWER the instruction executes as if the bit contained 0 except as follows: if LK=1, the Link Register is set (to an undefined value, except for ***svc***); if Rc=1, Condition Register Field 0 or 1 is set to an undefined value. In Power ISA, bit 31 is a reserved field for these instructions and is ignored by the processor.

Power ISA instructions for which bit 31 is the LK bit in POWER:

sc (***svc*** in POWER)
the *Condition Register Logical* instructions
mcrf
isync (***ics*** in POWER)

Power ISA instructions for which bit 31 is the Rc bit in POWER:

fixed-point X-form *Load* and *Store* instructions
fixed-point X-form *Compare* instructions
the X-form *Trap* instruction
mtspr, ***mfspr***, ***mtcrf***, ***mcrxr***, ***mfcr***, ***mtocrf***, ***mfocrf***
floating-point X-form *Load* and *Store* instructions
floating-point *Compare* instructions
mcrfs
dcbz (***dclz*** in POWER)

A.8 BO Field

POWER shows certain bits in the BO field — used by *Branch Conditional* instructions — as “x”. Although the POWER Architecture does not say how these bits are to be interpreted, they are in fact ignored by the processor.

Power ISA shows these bits as “z”, “a”, or “t”. The “z” bits are ignored, as in POWER. However, the “a” and “t” bits can be used by software to provide a hint about how the branch is likely to behave. If a POWER program has the “wrong” value for these bits, the program will produce the same results as on POWER but performance may be affected.

A.9 BH Field

Bits 19:20 of the *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions are reserved in POWER but are defined as a branch hint (BH) field in Power ISA. Because these bits are hints, they may affect performance but do not affect the results of executing the instruction.

A.10 Branch Conditional to Count Register

For the case in which the Count Register is decremented and tested (i.e., the case in which BO_2=0), POWER specifies only that the branch target address is undefined, with the implication that the Count Register, and the Link Register if LK=1, are updated in the normal way. Power ISA specifies that this instruction form is invalid.

A.11 System Call

There are several respects in which Power ISA is incompatible with POWER for *System Call* instructions — which in POWER are called *Supervisor Call* instructions.

- POWER provides a version of the *Supervisor Call* instruction (bit 30 = 0) that allows instruction fetching to continue at any one of 128 locations. It is used for “fast SVCs”. Power ISA provides no such version: if bit 30 of the instruction is 0 the instruction form is invalid.
- POWER provides a version of the *Supervisor Call* instruction (bits 30:31 = 0b11) that resumes instruction fetching at one location and sets the Link Register to the address of the next instruction. Power ISA provides no such version: bit 31 is a reserved field.
- For POWER, information from the MSR is saved in the Count Register. For Power ISA this information is saved in SRR1.
- In POWER bits 16:19 and 27:29 of the instruction comprise defined instruction fields or a portion thereof, while in Power ISA these bits comprise reserved fields.

- In POWER bits 20:26 of the instruction comprise a portion of the SV field, while in Power ISA these bits comprise the LEV field.
- POWER saves the low-order 16 bits of the instruction, in the Count Register. Power ISA does not save them.
- The settings of MSR bits by the associated interrupt differ between POWER and Power ISA; see *POWER Processor Architecture* and Book III.

A.12 Fixed-Point Exception Register (XER)

Bits 48:55 of the XER are reserved in Power ISA, while in POWER the corresponding bits (16:23) are defined and contain the comparison byte for the ***lscbx*** instruction (which Power ISA lacks).

A.13 Update Forms of Storage Access Instructions

Power ISA requires that RA not be equal to either RT (fixed-point *Load* only) or 0. If the restriction is violated the instruction form is invalid. POWER permits these cases, and simply avoids saving the EA.

A.14 Multiple Register Loads

Power ISArequires that RA, and RB if present in the instruction format, not be in the range of registers to be loaded, while POWER permits this and does not alter RA or RB in this case. (The Power ISA restriction applies even if RA=0, although there is no obvious benefit to the restriction in this case since RA is not used to compute the effective address if RA=0.) If the Power ISA restriction is violated, either the system illegal instruction error handler is invoked or the results are boundedly undefined. The instructions affected are:

lmw (***lm*** in POWER)
lswi (***lsi*** in POWER)
lswx (***lsx*** in POWER)

For example, an ***lmw*** instruction that loads all 32 registers is valid in POWER but is an invalid form in Power ISA.

A.15 Load/Store Multiple Instructions

There are two respects in which Power ISA is incompatible with POWER for *Load Multiple* and *Store Multiple* instructions.

- If the EA is not word-aligned, in Power ISA either an Alignment exception occurs or the addressed bytes are loaded, while in POWER an Alignment interrupt occurs if MSR_{AL}=1 (the low-order two bits of the EA are ignored if MSR_{AL}=0).
- In Power ISA the instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

A.16 Move Assist Instructions

There are several respects in which Power ISA is incompatible with POWER for *Move Assist* instructions.

- In Power ISA an ***lswx*** instruction with zero length leaves the contents of RT undefined (if RT≠RA and RT≠RB) or is an invalid instruction form (if RT=RA or RT=RB), while in POWER the corresponding instruction (***lsx***) is a no-op in these cases.

- In Power ISA a *Move Assist* instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

A.17 Move To/From SPR

There are several respects in which Power ISA is incompatible with POWER for *Move To/From Special Purpose Register* instructions.

- The SPR field is ten bits long in Power ISA, but only five in POWER (see also Section A.3, “Reserved Fields in Instructions”).
- ***mfspr*** can be used to read the Decrementer in problem state in POWER, but only in privileged state in Power ISA.
- If the SPR value specified in the instruction is not one of the defined values, POWER behaves as follows.
  - If the instruction is executed in problem state and SPR_0=1, a Privileged Instruction type Program interrupt occurs. No architected registers are altered except those set by the interrupt.
  - Otherwise no architected registers are altered.

  In this same case, Power ISA behaves as follows.
  - If the instruction is executed in problem state, a Hypervisor Emulation Assistance interrupt occurs if spr_0=0 and a Privileged Instruction type Program interrupt occurs if spr_0=1. No architected registers are altered except those set by the interrupt.
  - If the instruction is executed in privileged state, a Hypervisor Emulation Assistance interrupt occurs if the SPR value is 0 or, for ***mfspr*** only, if the SPR value is 4, 5, or 6. In

these cases no architected registers are altered except those set by the interrupt. Otherwise no operation is performed. (See Section 4.4.4, “Move To/From System Register Instructions” in Book III-S.)

A.18 Effects of Exceptions on FPSCR Bits FR and FI

For the following cases, POWER does not specify how FR and FI are set, while Power ISA preserves them for Invalid Operation Exception caused by a *Compare* instruction, sets FI to 1 and FR to an undefined value for disabled Overflow Exception, and clears them otherwise.

- Invalid Operation Exception (enabled or disabled)
- Zero Divide Exception (enabled or disabled)
- Disabled Overflow Exception

A.19 Store Floating-Point Single Instructions

There are several respects in which Power ISA is incompatible with POWER for *Store Floating-Point Single* instructions.

- POWER uses $FPSCR_{UE}$ to help determine whether denormalization should be done, while Power ISA does not. Using $FPSCR_{UE}$ is in fact incorrect: if $FPSCR_{UE}$=1 and a denormalized single-precision number is copied from one storage location to another by means of ***lfs*** followed by ***stfs***, the two “copies” may not be the same.
- For an operand having an exponent that is less than 874 (unbiased exponent less than -149), POWER stores a zero (if $FPSCR_{UE}$=0) while Power ISA stores an undefined value.

A.20 Move From FPSCR

POWER defines the high-order 32 bits of the result of ***mffs*** to be 0xFFFF_FFFF, while Power ISA copies the high-order 32-bits of the FPSCR.

A.21 Zeroing Bytes in the Data Cache

The ***dclz*** instruction of POWER and the ***dcbz*** instruction of Power ISA have the same opcode. However, the functions differ in the following respects.

- ***dclz*** clears a line while ***dcbz*** clears a block.
- ***dclz*** saves the EA in RA (if RA≠0) while ***dcbz*** does not.
- ***dclz*** is privileged while ***dcbz*** is not.

A.22 Synchronization

The *Synchronize* instruction (called ***dcs*** in POWER) and the ***isync*** instruction (called ***ics*** in POWER) cause more pervasive synchronization in Power ISA than in POWER. However, unlike ***dcs***, *Synchronize* does not wait until data cache block writes caused by preceding instructions have been performed in main storage. Also, *Synchronize* has an L field while ***dcs*** does not, and some uses of the instruction by the operating system require L=2<S>. (The L field corresponds to reserved bits in ***dcs*** and hence is expected to be zero in POWER programs; see Section A.3.)

A.23 Move To Machine State Register Instruction

The ***mtmsr*** instruction has an L field in Power ISA but not in POWER. The function of the variant of ***mtmsr*** with L=1 differs from the function of the instruction in the POWER architecture in the following ways.

- In Power ISA, this variant of ***mtmsr*** modifies only the EE and RI bits of the MSR, while in the POWER ***mtmsr*** modifies all bits of the MSR.
- This variant of ***mtmsr*** is execution synchronizing in Power ISA but is context synchronizing in POWER. (The POWER architecture lacks Power ISA’s distinction between execution synchronization and context synchronization. The statement in the POWER architecture specification that ***mtmsr*** is “synchronizing” is equivalent to stating that the instruction is context synchronizing.)

Also, ***mtmsr*** is optional in Power ISA but required in POWER.

A.24 Direct-Store Segments

POWER’s direct-store segments are not supported in Power ISA.

A.25 Segment Register Manipulation Instructions

The definitions of the four *Segment Register Manipulation* instructions ***mtsr***, ***mtsrin***, ***mfsr***, and ***mfsrin*** differ in two respects between POWER and Power ISA. Instructions similar to ***mtsrin*** and ***mfsrin*** are called ***mtsri*** and ***mfsri*** in POWER.

privilege: ***mfsr*** and ***mfsri*** are problem state instructions in POWER, while ***mfsr*** and ***mfsrin*** are privileged in Power ISA.

function: the “indirect” instructions (***mtsri*** and ***mfsri***) in POWER use an RA register in computing the Segment Register number, and the computed EA is stored into RA (if

RA≠0 and RA≠RT), while in Power ISA ***mtsrin*** and ***mfsrin*** have no RA field and the EA is not stored.

mtsr, ***mtsrin*** (***mtsri***), and ***mfsr*** have the same opcodes in Power ISA as in POWER. ***mfsri*** (POWER) and ***mfsrin*** (Power ISA) have different opcodes.

Also, the *Segment Register Manipulation* instructions are required in POWER whereas they are optional in Power ISA.

A.26 TLB Entry Invalidation

The ***tlbi*** instruction of POWER and the ***tlbie*** instruction of Power ISA have the same opcode. However, the functions differ in the following respects.

- ***tlbi*** computes the EA as (RAl0) + (RB), while ***tlbie*** lacks an RA field and computes the EA and related information as (RB).
- ***tlbi*** saves the EA in RA (if RA≠0), while ***tlbie*** lacks an RA field and does not save the EA.
- For ***tlbi*** the high-order 36 bits of RB are used in computing the EA, while for ***tlbie*** these bits contain additional information that is not directly related to the EA.
- For ***tlbi*** has no RS operand, while for ***tlbie*** the (RS) is an LPID value used to qualify the TLB invalidation.

Also, ***tlbi*** is required in POWER whereas ***tlbie*** is optional in Power ISA.

A.27 Alignment Interrupts

Placing information about the interrupting instruction into the DSISR and the DAR when an Alignment interrupt occurs is optional in Power ISA but required in POWER.

A.28 Floating-Point Interrupts

POWER uses MSR bit 20 to control the generation of interrupts for floating-point enabled exceptions, and Power ISA uses the corresponding MSR bit, bit 52, for the same purpose. However, in Power ISA this bit is part of a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER this bit is used independently to control the occurrence of the interrupt (in POWER all floating-point interrupts are precise).

A.29 Timing Facilities

A.29.1 Real-Time Clock

The POWER Real-Time Clock is not supported in Power ISA. Instead, Power ISA provides a Time Base. Both the RTC and the TB are 64-bit Special Purpose Registers, but they differ in the following respects.

- The RTC counts seconds and nanoseconds, while the TB counts “ticks”. The ticking rate of the TB is implementation-dependent.
- The RTC increments discontinuously: 1 is added to RTCU when the value in RTCL passes 999_999_999. The TB increments continuously: 1 is added to TBU when the value in TBL passes 0xFFFF_FFFF.
- The RTC is written and read by the ***mtspr*** and ***mfspr*** instructions, using SPR numbers that denote the RTCU and RTCL. The TB is written and read by the same instructions using different SPR numbers.
- The SPR numbers that denote POWER’s RTCL and RTCU are invalid in Power ISA.
- The RTC is guaranteed to increment at least once in the time required to execute ten *Add Immediate* instructions. No analogous guarantee is made for the TB.
- Not all bits of RTCL need be implemented, while all bits of the TB must be implemented.

A.29.2 Decrementer

The Power ISA Decrementer differs from the POWER Decrementer in the following respects.

- The Power ISA DEC decrements at the same rate that the TB increments, while the POWER DEC decrements every nanosecond (which is the same rate that the RTC increments).
- Not all bits of the POWER DEC need be implemented, while all bits of the Power ISA DEC must be implemented.
- The interrupt caused by the DEC has its own interrupt vector location in Power ISA, but is considered an External interrupt in POWER.

A.30 Deleted Instructions

The following instructions are part of the POWER Architecture but have been dropped from the Power ISA.

abs	Absolute
clcs	Cache Line Compute Size
clf	Cache Line Flush
cli (*)	Cache Line Invalidate
dclst	Data Cache Line Store
div	Divide
divs	Divide Short
doz	Difference Or Zero
dozi	Difference Or Zero Immediate
lscbx	Load String And Compare Byte Indexed
maskg	Mask Generate
maskir	Mask Insert From Register
mfsri	Move From Segment Register Indirect
mul	Multiply
nabs	Negative Absolute
rac (*)	Real Address Compute
rfi (*)	Return From Interrupt
rfsvc	Return From SVC
rlmi	Rotate Left Then Mask Insert
rrib	Rotate Right And Insert Bit
sle	Shift Left Extended
sleq	Shift Left Extended With MQ
sliq	Shift Left Immediate With MQ
slliq	Shift Left Long Immediate With MQ
sllq	Shift Left Long With MQ
slq	Shift Left With MQ
sraiq	Shift Right Algebraic Immediate With MQ
sraq	Shift Right Algebraic With MQ
sre	Shift Right Extended
srea	Shift Right Extended Algebraic
sreq	Shift Right Extended With MQ
sriq	Shift Right Immediate With MQ
srliq	Shift Right Long Immediate With MQ
srlq	Shift Right Long With MQ
srq	Shift Right With MQ

(*) This instruction is privileged.

Note: Many of these instructions use the MQ register. The MQ is not defined in the Power ISA.

A.31 Discontinued Opcodes

The opcodes listed below are defined in the POWER Architecture but have been dropped from the Power ISA. The list contains the POWER mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The corresponding instructions are reserved in Power ISA.

MNEM	PRI	XOP
abs	31	360
clcs	31	531
clf	31	118
cli (*)	31	502
dclst	31	630
div	31	331
divs	31	363
doz	31	264
dozi	09	-
lscbx	31	277
maskg	31	29
maskir	31	541
mfsri	31	627
mul	31	107
nabs	31	488
rac (*)	31	818
rfi (*)	19	50
rfsvc	19	82
rlmi	22	-
rrib	31	537
sle	31	153
sleq	31	217
sliq	31	184
slliq	31	248
sllq	31	216
slq	31	152
sraiq	31	952
sraq	31	920
sre	31	665
srea	31	921
sreq	31	729
sriq	31	696
srliq	31	760
srlq	31	728
srq	31	664

(*) This instruction is privileged.

Assembler Note

It might be helpful to current software writers for the Assembler to flag the discontinued POWER instructions.

A.32 POWER2 Compatibility

The POWER2 instruction set is a superset of the POWER instruction set. Some of the instructions added for POWER2 are included in the Power ISA. Those that have been renamed in the Power ISA are listed in this section, as are the new POWER2 instructions that are not included in the Power ISA.

Other incompatibilities are also listed.

A.32.1 Cross-Reference for Changed POWER2 Mnemonics

The following table lists the new POWER2 instruction mnemonics that have been changed in the Power ISA User Instruction Set Architecture, sorted by POWER2 mnemonic.

To determine the Power ISA mnemonic for one of these POWER2 mnemonics, find the POWER2 mnemonic in the second column of the table: the remainder of the line gives the Power ISA mnemonic and the page on which the instruction is described, as well as the instruction names.

POWER2 mnemonics that have not changed are not listed.

Page	POWER2		Power ISA	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
153	fcir[.]	Floating Convert Double to Integer with Round	fctiw[.]	Floating Convert To Integer Word
154	fcirz[.]	Floating Convert Double to Integer with Round to Zero	fctiwz[.]	Floating Convert To Integer Word with round toward Zero

A.32.2 Load/Store Floating-Point Double

Several of the opcodes for the *Load/Store Floating-Point Quad* instructions of the POWER2 architecture have been reclaimed by the *Load/Store Foating-Point Double [Indexed]* instructions (entries with a ***'-'*** in the Power ISA column have not been reclaimed):

MNEMONIC			
POWER2	**POWER ISA**	**PRI**	**XOP**
lfq	***lq***	56	-
lfqu	***lfdp***	57	0
lfqux	***-***	31	823
lfqx	***lfdpx***	31	791
stfq	***-***	60	-
stfqu	***stfdp***	61	-
stfqux	***-***	31	951
stfqx	***stfdpx***	31	919

Differences between the ***l/stfdp[x]*** instructions and the POWER2 ***l***/***stfq***[***u***][***x***] instructions include the following.

- The storage operand for the ***l/stfdp[x]*** instructions must be quadword aligned for optimal performance.
- The register pairs for the ***l/stfdp[x]*** instructions must be even-odd pairs, instead of any consecutive pair.
- The ***l/stfdp[x]*** instructions do not have update forms.

A.32.3 Floating-Point Conversion to Integer

The ***fcir*** and ***fcirz*** instructions of POWER2 have the same opcodes as do the ***fctiw*** and ***fctiwz*** instructions, respectively, of Power ISA. However, the functions differ in the following respects.

- ***fcir*** and ***fcirz*** set the high-order 32 bits of the target FPR to 0xFFFF_FFFF, while ***fctiw*** and ***fctiwz*** set them to an undefined value.
- Except for enabled Invalid Operation Exceptions, ***fcir*** and ***fcirz*** set the FPRF field of the FPSCR based on the result, while ***fctiw*** and ***fctiwz*** set it to an undefined value.
- ***fcir*** and ***fcirz*** do not affect the VXSNAN bit of the FPSCR, while ***fctiw*** and ***fctiwz*** do.
- ***fcir*** and ***fcirz*** set $FPSCR_{XX}$ to 1 for certain cases of “Large Operands” (i.e., operands that are too large to be represented as a 32-bit signed fixed-point integer), while ***fctiw*** and ***fctiwz*** do not alter it for any case of “Large Operand”. (The IEEE standard requires not altering it for “Large Operands”.)

A.32.4 Floating-Point Interrupts

POWER2 uses MSR bits 20 and 23 to control the generation of interrupts for floating-point enabled exceptions, and Power ISA uses the corresponding MSR bits, bits 52 and 55, for the same purpose. However, in Power ISA these bits comprise a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER2 these bits are used independently to control the occurrence (bit 20) and the precision (bit 23) of the interrupt. Moreover, in Power ISA all floating-point interrupts are considered Program interrupts, while in POWER2 imprecise floating-point interrupts have their own interrupt vector location.

A.32.5 Trace

The Trace interrupt vector location differs between the two architectures, and there are many other differences.

A.33 Deleted Instructions

The following instructions are new in POWER2 implementations of the POWER Architecture but have been dropped from the Power ISA.

lfq	Load Floating-Point Quad
lfqu	Load Floating-Point Quad with Update
lfqux	Load Floating-Point Quad with Update Indexed
lfqx	Load Floating-Point Quad Indexed
stfq	Store Floating-Point Quad
stfqu	Store Floating-Point Quad with Update
stfqux	Store Floating-Point Quad with Update Indexed
stfqx	Store Floating-Point Quad Indexed

A.33.1 Discontinued Opcodes

The opcodes listed below are new in POWER2 implementations of the POWER Architecture but have been dropped from the Power ISA. The list contains the POWER2 mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The instructions are either illegal or reserved in Power ISA; see Appendix D.

MNEM	PRI	XOP
lfq	56	-
lfqx	31	791
stfqx	31	919

Appendix B. Platform Support Requirements

As described in Chapter 1 of Book I, the architecture is structured as a collection of categories. Each category is comprised of facilities and/or instructions that together provide a unit of functionality. The Server and Embedded categories are referred to as “special” because all implementations must support at least one of these categories. Each special category, when taken together with the Base category, is referred to as an “environment”, and provides the minimum functionality required to develop operating systems and applications.

Every processor implementation supports at least one of the environments, and may also support a set of categories chosen based on the target market for the implementation. However, a Server implementation supports only those categories designated as part of the Server platform in Figure 1. To facilitate the development of operating systems and applications for a well-defined purpose or customer set, usually embodied in a unique hardware platform, this appendix documents the association between a platform and the set of categories it requires.

Adding a new platform may permit cost-performance optimization by clearly identifying a unique set of categories. However, this has the potential to fragment the application base. As a result, new platforms will be added only when the optimization benefit clearly outweighs the loss due to fragmentation.

The platform support requirements are documented in Figure 1. An “x” in a column indicates that the category is required. A “+” in a column indicates that the requirement is being phased in.

Category	Server Platform	Embedded Platform
Base	x	x
Server	x	
Embedded		x
Alternate Time Base		
Cache Specification		
Decimal Floating-Point	x	
Decorated Storage	x	
Elemental Memory Barriers		
Embedded.Cache Debug		
Embedded.Cache Initialization		
Embedded.Device Control		
Embedded.Enhanced Debug		
Embedded.External PID		
Embedded.Hypervisor		
Embedded.Hypervisor.LRAT		
Embedded.Little-Endian		
Embedded.Page Table		
Embedded.Performance Monitor		
Embedded.Processor Control		
Embedded Cache Locking		
Embedded Multi-Threading Embedded Multi-Threading.Thread Management		
Embedded.TLB Write Conditional		
External Control		
External Proxy		
Floating-Point Floating-Point.Record	x x	
Legacy Move Assist		
Legacy Integer Multiply-Accumulate		
Load/Store Quadword	x^2	
Memory Coherence	x	
Move Assist	x	
Processor Compatibility	x	
Signal Processing Engine SPE.Embedded Float Scalar Double SPE.Embedded Float Scalar Single SPE.Embedded Float Vector		
Store Conditional Page Mobility	x	
Stream	x	
Strong Access Order	x	
Trace	x	
Transactional Memory	x	

Figure 1. Platform Support Requirements (Sheet 1 of 2)

Category	Server Platform	Embedded Platform
Variable Length Encoding		
Vector Vector.Little-Endian	+ +[1]	
Wait		
64-Bit	x	

1. If the Vector category is supported, Vector.Little-Endian is required on the Server platform.
2. Optional for the Server Platform.

Figure 1. Platform Support Requirements (Sheet 2 of 2)

Appendix C. Complete SPR List

This appendix lists all the Special Purpose Registers in the Power ISA , ordered by SPR number.

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
1	00000 00001	XER	no	no	64	B
3	00000 00011	DSCR	no	no	64	STM
8	00000 01000	LR	no	no	64	B
9	00000 01001	CTR	no	no	64	B
13	00000 01101	AMR	no[9]	no	64	S
17	00000 10001	DSCR	yes	yes	64	STM
18	00000 10010	DSISR	yes	yes	32	S
19	00000 10011	DAR	yes	yes	64	S
22	00000 10110	DEC	yes[13]	yes[13]	32	B
25	00000 11001	SDR1	hypv[3]	hypv[3]	64	S
26	00000 11010	SRR0	yes[13]	yes[13]	64	B
27	00000 11011	SRR1	yes[13]	yes[13]	64	B
28	00000 11100	CFAR	yes	yes	64	S
29	00000 11101	AMR	yes[9]	yes	64	S
48	00001 10000	PID	yes	yes	32	E
53	00001 10101	GDECAR	hypv[3]	no	32	E.HV
54	00001 10110	DECAR	hypv[12]	-	32	E
55	00001 10111	MCIVPR	hypv[12]	hypv[12]	64	E
56	00001 11000	LPER	hypv[12]	hypv[12]	64	E.HV; E.PT
57	00001 11001	LPERU	hypv[12]	hypv[12]	32	E.HV; E.PT
58	00001 11010	CSRR0	hypv[12]	hypv[12]	64	E
59	00001 11011	CSRR1	hypv[12]	hypv[12]	32	E
60	00001 11100	GTSRWR	hypv[3]	no	32	E.HV
61	00001 11101	IAMR	yes[8]	yes	64	S
61	00001 11101	DEAR	yes[13]	yes[13]	64	E
62	00001 11110	ESR	yes[13]	yes[13]	32	E
63	00001 11111	IVPR	hypv[12]	hypv[12]	64	E
128	00100 00000	TFHAR	no	no	64	TM
129	00100 00001	TFIAR	no	no	64	TM
130	00100 00010	TEXASR	no	no	64	TM
131	00100 00011	TEXASRU	no	no	32	TM
136	00100 01000	CTRL	-	no	32	S
152	00100 11000	CTRL	yes	-	32	S
153	00100 11001	FSCR	yes	yes	64	S
157	00100 11101	UAMOR	yes[10]	yes	64	S
159	00100 11111	PSPB	yes	yes	32	S
176	00101 10000	DPDES	hypv[3]	yes	64	S.PC
177	00101 10001	DHDES	hypv[3]	hypv[3]	64	S.PC
180	00101 10100	DAWR0	hypv[3]	hypv[3]	64	S

Figure 2. SPR Numbers (Sheet 1 of 5)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
186	00101 11010	RPR	hypv[3]	hypv[3]	64	S
187	00101 11011	CIABR	hypv[3]	hypv[3]	64	S
188	00101 11100	DAWRX0	hypv[3]	hypv[3]	32	S
190	00101 11110	HFSCR	hypv[3]	hypv[3]	64	S
256	01000 00000	VRSAVE	no	no	32	B
259	01000 00011	SPRG3	-	no	64	B
260-263	01000 001xx	SPRG[4-7]	-	no	64	E
268	01000 01100	TB	-	no	64	B
269	01000 01101	TBU	-	no	32	B
272-275	01000 100xx	SPRG[0-3]	yes[13]	yes[13]	64	B
276-279	01000 101xx	SPRG[4-7]	yes	yes	64	E
282	01000 11010	EAR	hypv[4]	hypv[4]	32	EC
283	01000 11011	CIR	-	hypv[4]	32	E
283	01000 11011	CIR	-	yes	32	S
284	01000 11100	TBL	hypv[4]	-	32	B
285	01000 11101	TBU	hypv[4]	-	32	B
286	01000 11110	TBU40	hypv	-	64	S
286	01000 11110	PIR	hypv[12]	yes[13]	32	E
287	01000 11111	PVR	-	yes	32	B
304	01001 10000	HSPRG0	hypv[3]	hypv[3]	64	S
304	01001 10000	DBSR	hypv[5,12]	hypv[12]	32	E
305	01001 10001	HSPRG1	hypv[3]	hypv[3]	64	S
306	01001 10010	HDSISR	hypv[3]	hypv[3]	32	S
306	01001 10010	DBSRWR	hypv[3]	-	32	E.HV
307	01001 10011	HDAR	hypv[3]	hypv[3]	64	S
307	01001 10011	EPCR	hypv[3]	hypv[3]	32	E.HV, (E;64)
308	01001 10100	SPURR	hypv[3]	yes	64	S
308	01001 10100	DBCR0	hypv[12]	hypv[12]	32	E
309	01001 10101	PURR	hypv[3]	yes	64	S
309	01001 10101	DBCR1	hypv[12]	hypv[12]	32	E
310	01001 10110	HDEC	hypv[3]	hypv[3]	32	S
310	01001 10110	DBCR2	hypv[12]	hypv[12]	32	E
311	01001 10111	MSRP	hypv[3]	hypv[3]	32	E.HV
312	01001 11000	RMOR	hypv[3]	hypv[3]	64	S
312	01001 11000	IAC1	hypv[12]	hypv[12]	64	E
313	01001 11001	HRMOR	hypv[3]	hypv[3]	64	S
313	01001 11001	IAC2	hypv[12]	hypv[12]	64	E
314	01001 11010	HSRR0	hypv[3]	hypv[3]	64	S
314	01001 11010	IAC3	hypv[12]	hypv[12]	64	E
315	01001 11011	HSRR1	hypv[3]	hypv[3]	64	S
315	01001 11011	IAC4	hypv[12]	hypv[12]	64	E
316	01001 11100	DAC1	hypv[12]	hypv[12]	64	E
317	01001 11101	DAC2	hypv[12]	hypv[12]	64	E
318	01001 11110	LPCR	hypv[3]	hypv[3]	64	S
319	01001 11111	LPIDR	hypv[3]	hypv[3]	32	S
336	01010 10000	TSR	yes[5,13]	yes[13]	32	E
336	01010 10000	HMER	hypv[3,8]	hypv[3]	64	S
337	01010 10001	HMEER	hypv[3]	hypv[3]	64	S
338	01010 10010	PCR	hypv[3]	hypv[3]	64	S
338	01010 10010	LPIDR	hypv[3]	hypv[3]	32	E.HV
339	01010 10011	HEIR	hypv[3]	hypv[3]	32	S
339	01010 10011	MAS5	hypv[3]	hypv[3]	32	E.HV
340	01010 10100	TCR	yes[13]	yes[13]	32	E
341	01010 10101	MAS8	hypv[3]	hypv[3]	32	E.HV

Figure 2. SPR Numbers (Sheet 2 of 5)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
342	01010 10110	LRATCFG	-	hypv[3]	32	E.HV.LRAT
343	01010 10111	LRATPS	-	hypv[3]	32	E.HV.LRAT
344-347	01010 110xx	TLB[0-3]PS	-	hypv[3]	32	E.HV
348	01010 11100	MAS5||MAS6	hypv[3]	hypv[3]	64	E.HV; 64
349	01010 11101	MAS8||MAS1	hypv[3]	hypv[3]	64	E.HV; 64
349	01010 11101	AMOR	hypv[3]	hypv[3]	64	S
350	01010 11110	EPTCFG	hypv[9]	hypv[9]	32	E.PT
368-371	01011 100xx	GSPRG0-3	yes	yes	64	E.HV
372	01011 10100	MAS7||MAS3	yes	yes	64	E; 64
373	01011 10101	MAS0||MAS1	yes	yes	64	E; 64
374	01011 10110	GDEC	yes	yes	32	E.HV
375	01011 10111	GTCR	yes	yes	32	E.HV
376	01011 11000	GTSR	yes	yes	32	E.HV
378	01011 11010	GSRR0	yes	yes	64	E.HV
379	01011 11011	GSRR1	yes	yes	32	E.HV
380	01011 11100	GEPR	yes	yes	32	E.HV;EXP
381	01011 11101	GDEAR	yes	yes	64	E.HV
382	01011 11110	GPIR	hypv[3]	yes	32	E.HV
383	01011 11111	GESR	yes	yes	32	E.HV
400-415	01100 1xxxx	IVOR[0-15]	hypv[12]	hypv[12]	32	E
432-435	01101 100xx	IVOR38-41	hypv[12]	hypv[12]	32	E.HV
436	01101 10100	IVOR42	hypv[12]	hypv[12]	32	E.HV.LRAT
437	01101 10101	TENSR	-	hypv[12]	64	E.MT
438	01101 10110	TENS	hypv[12]	hypv[12]	64	E.MT
439	01101 10111	TENC	hypv[12]	hypv[12]	64	E.MT
440-441	01101 1100x	GIVOR2-3	hypv[3]	yes	32	E.HV
442	01101 11010	GIVOR4	hypv[3]	yes	32	E.HV
443	01101 11011	GIVOR8	hypv[3]	yes	32	E.HV
444	01101 11100	GIVOR13	hypv[3]	yes	32	E.HV
445	01101 11101	GIVOR14	hypv[3]	yes	32	E.HV
446	01101 11110	TIR	-	hypv[12]	64	E.MT
446	01101 11110	TIR	-	yes	64	S.PC
447	01101 11111	GIVPR	hypv[3]	yes	64	E.HV
464	01110 10000	GIVOR35	hypv[3]	yes	32	E.HV;E.PM
474	11010 01110	GIVOR10	hypv[3]	yes	32	E.HV
475	11011 01110	GIVOR11	hypv[3]	yes	32	E.HV
476	11100 01110	GIVOR12	hypv[3]	yes	32	E.HV
512	10000 00000	SPEFSCR	no	no	32	SP
526	10000 01110	ATB/ATBL	-	no	64	ATB
527	10000 01111	ATBU	-	no	32	ATB
528	10000 10000	IVOR32	hypv[12]	hypv[12]	32	SP
529	10000 10001	IVOR33	hypv[12]	hypv[12]	32	SP
530	10000 10010	IVOR34	hypv[12]	hypv[12]	32	SP
531	10000 10011	IVOR35	hypv[12]	hypv[12]	32	E.PM
532	10000 10100	IVOR36	hypv[12]	hypv[12]	32	E.PC
533	10000 10101	IVOR37	hypv[12]	hypv[12]	32	E.PC
570	10001 11010	MCSRR0	hypv[12]	hypv[12]	64	E
571	10001 11011	MCSRR1	hypv[12]	hypv[12]	32	E
572	10001 11100	MCSR	hypv[12]	hypv[12]	64	E
574	10001 11110	DSRR0	yes	yes	64	E.ED
575	10001 11111	DSRR1	yes	yes	32	E.ED
604	10010 11100	SPRG8	hypv[12]	hypv[12]	64	E
605	10010 11101	SPRG9	yes	yes	64	E.ED
624	10011 10000	MAS0	yes	yes	32	E

Figure 2. SPR Numbers (Sheet 3 of 5)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
625	10011 10001	MAS1	yes	yes	32	E
626	10011 10010	MAS2	yes	yes	64	E
627	10011 10011	MAS3	yes	yes	32	E
628	10011 10100	MAS4	yes	yes	32	E
630	10011 10110	MAS6	yes	yes	32	E
631	10011 10111	MAS2U	yes	yes	32	E
688-691	10101 100xx	TLB[0-3]CFG	-	hypv[12]	32	E
702	10101 11110	EPR	-	yes[13]	32	EXP
768	11000 00000	SIER	-	yes	64	S
769	11000 00001	MMCR2	no[14]	no[14]	64	S
770	11000 00010	MMCRA	no[14]	no[14]	64	S
771	11000 00011	PMC1	no[14]	no[14]	32	S
772	11000 00100	PMC2	no[14]	no[14]	32	S
773	11000 00101	PMC3	no[14]	no[14]	32	S
774	11000 00110	PMC4	no[14]	no[14]	32	S
775	11000 00111	PMC5	no[14]	no[14]	32	S
776	11000 01000	PMC6	no[14]	no[14]	32	S
779	11000 01011	MMCR0	no[14]	no[14]	64	S
780	11000 01100	SIAR	-	no[14]	64	S
781	11000 01101	SDAR	-	no[14]	64	S
782	11000 01110	MMCR1	-	no[14]	64	S
784	11000 10000	SIER	yes	yes	64	S
785	11000 10001	MMCR2	yes	yes	64	S
786	11000 10010	MMCRA	yes	yes	64	S
787	11000 10011	PMC1	yes	yes	32	S
788	11000 10100	PMC2	yes	yes	32	S
789	11000 10101	PMC3	yes	yes	32	S
790	11000 10110	PMC4	yes	yes	32	S
791	11000 10111	PMC5	yes	yes	32	S
792	11000 11000	PMC6	yes	yes	32	S
795	11000 11011	MMCR0	yes	yes	64	S
796	11000 11100	SIAR	yes	yes	64	S
797	11000 11101	SDAR	yes	yes	64	S
798	11000 11110	MMCR1	yes	yes	64	S
800	11001 00000	BESCRS	no	no	64	S
801	11001 00001	BESCRSU	no	no	32	S
802	11001 00010	BESCRR	no	no	64	S
803	11001 00011	BESCRRU	no	no	32	S
804	11001 00100	EBBHR	no	no	64	S
805	11001 00101	EBBRR	no	no	64	S
806	11001 00110	BESCR	no	no	64	S
808	11001 01000	reserved[15]	no	no	na	B
809	11001 01001	reserved[15]	no	no	na	B
810	11001 01010	reserved[15]	no	no	na	B
811	11001 01011	reserved[15]	no	no	na	B
815	11001 01111	TAR	no	no	64	S
848	11010 10000	IC	hypv[3]	yes	64	S
849	11010 10001	VTB	hypv[3]	yes	64	S
896	11100 00000	PPR	no	no	64	S
898	11100 00010	PPR32	no	no	32	B
924	11100 11100	DCDBTRL	-[6]	hypv[12]	32	E.CD
925	11100 11101	DCDBTRH	-[6]	hypv[12]	32	E.CD
926	11100 11110	ICDBTRL	-[7]	hypv[12]	32	E.CD
927	11100 11111	ICDBTRH	-[7]	hypv[12]	32	E.CD

Figure 2. SPR Numbers (Sheet 4 of 5)

decimal	SPR[1]		Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$	$spr_{0:4}$		mtspr	mfspr		
944	11101	10000	MAS7	yes	yes	32	E
947	11101	10011	EPLC	yes	yes	32	E.PD
948	11101	10100	EPSC	yes	yes	32	E.PD
979	11110	10011	ICDBDR	-[7]	hypv[12]	32	E.CD
1012	11111	10100	MMUCSR0	hypv[12]	hypv[12]	32	E
1015	11111	10111	MMUCFG	-	hypv[12]	32	E
1023	11111	11111	PIR	-	yes	32	S

- This register is not defined for this instruction.

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.

[2] See Section 1.3.5 of Book I. If multiple categories are listed separated by a semicolon, all the listed categories must be implemented in order for the other columns of the line to apply. A comma separates two alternatives, and takes precedence over a semicolon; e.g., the EPCR (E.HV,E;64) must be implemented if either (a) category E.HV is implemented or (b) the processor is an Embedded processor that implements the 64-Bit category.

[3] This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-S or Chapter 2 of Book III-E as appropriate).

[4] <S>This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-S).
<E>If the Embedded.Hypervisor category is supported, this register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E). Otherwise the register is privileged.

[5] This register cannot be directly written. Instead, bits in the register corresponding to 1 bits in (RS) can be cleared using *mtspr SPR,RS*.

[6] The register can be written by the ***dcread*** instruction.

[7] The register can be written by the ***icread*** instruction.

[8] This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using *mtspr SPR,RS*.

[9] The value specified in register RS may be masked by the contents of the [U]AMOR before being placed into the AMR; see the ***mtspr*** instruction description in Book III-S.

[10] The value specified in register RS may be ANDed with the contents of the AMOR before being placed into the UAMOR; see the ***mtspr*** instruction description in Book III-S.

[11] The register is Category: Phased-in.

[12] If the Embedded.Hypervisor category is supported, this register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2 of Book III-E). Otherwise the register is privileged for Embedded.

[13] If the Embedded.Hypervisor category is supported, this register is a hypervisor resource and can be accessed by this instruction only in hypervisor state, and guest references to the register are redirected to the corresponding guest register (see Chapter 2 of Book III-E). Otherwise the register is privileged.

[14] $MMCR0_{PMCC}$ controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 9.4 for details.

[15] Accesses to these SPRs are noops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I..

SPR numbers 777-778, 783, 793-794, and 799 are reserved for the Performance Monitor. All other SPR numbers that are not shown above and are not implementation-specific are reserved.

Figure 2. SPR Numbers (Sheet 5 of 5)

Appendix D. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

1, 5, 6

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix F of Book Appendices. All unused extended opcodes are illegal.

4, 19, 30, 31, 56, 5 , 58, 59, 60, 62, 63

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture.

Appendix E. Reserved Instructions

The instructions in this class are allocated to specific purposes that are outside the scope of the Power ISA.

The following types of instruction are included in this class.

1. The instruction having primary opcode 0, except the instruction consisting entirely of binary 0s (which is an illegal instruction; see Section 1.7.2, “Illegal Instruction Class” on page 21) and the extended opcode shown below.

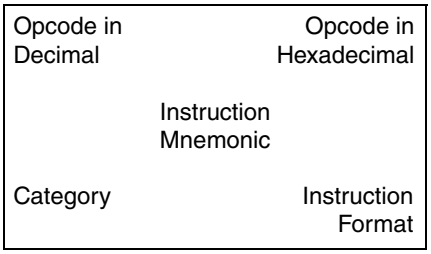
   256 Service Processor “Attention”

2. Instructions for the POWER Architecture that have not been included in the Power ISA. These are listed in Section A.31, “Discontinued Opcodes” and Section A.33.1, “Discontinued Opcodes”.
3. Implementation-specific instructions used to conform to the Power ISA specification.
4. Any other implementation-dependent instructions that are not defined in the Power ISA.

Appendix F. Opcode Maps

This appendix contains tables showing the opcodes and extended opcodes.

For the primary opcode table (Table 1 on page 1372), each cell is in the following format.



The category abbreviations are shown on Section 1.3.5 of Book I. However, the categories "Phased-In", "Phased-Out", and floating-point "Record" are not listed in the opcode tables.

The extended opcode tables show the extended opcode in decimal, the instruction mnemonic, the category, and the instruction format. These tables appear in order of primary opcode within three groups. The first group consists of the primary opcodes that have small extended opcode fields (2-4 bits), namely 30, 58, and 62. The second group consists of primary opcodes that have 11-bit extended opcode fields. The third group consists of primary opcodes that have 10-bit extended opcode fields. The tables for the second and third groups are rotated.

In the extended opcode tables several special markings are used.

- A prime (') following an instruction mnemonic denotes an additional cell, after the lowest-numbered one, used by the instruction. For example, ***subfc*** occupies cells 8 and 520 of primary opcode 31, with the former corresponding to OE=0 and the latter to OE=1. Similarly, ***sradi*** occupies cells 826 and 827, with the former corresponding to sh_5=0 and the latter to sh_5=1 (the 9-bit extended opcode 413, shown on page 100, excludes the sh_5 bit).
- Two vertical bars (||) are used instead of primed mnemonics when an instruction occupies an entire column of a table. The instruction mnemonic is repeated in the last cell of the column.
- For primary opcode 31, an asterisk (*) in a cell that would otherwise be empty means that the cell is reserved because it is "overlaid", by a fixed-point or *Storage Access* instruction having only a primary opcode, by an instruction having an extended opcode in primary opcode 30, 58, or 62, or by a potential instruction in any of the categories just mentioned. The overlaying instruction, if any, is also shown. A cell thus reserved should not be assigned to an instruction having primary opcode 31. (The overlaying is a consequence of opcode decoding for fixed-point instructions: the primary opcode, and the extended opcode if any, are mapped internally to a 10-bit "compressed opcode" for ease of subsequent decoding on some implementations that complied with previous versions of the architecture.)
- Parentheses around the opcode or extended opcode mean that the instruction was defined in earlier versions of the Power ISA but is no longer defined in the Power ISA.
- Curly brackets around the opcode or extended opcode mean that the instruction will be defined in future versions of the Power ISA.
- ***long*** is used as filler for mnemonics that are longer than a table cell.

An empty cell, a cell containing only an asterisk, or a cell in which the opcode or extended opcode is parenthesized, corresponds to an illegal instruction.

The instruction consisting entirely of binary 0s causes the system illegal instruction error handler to be invoked for all members of the POWER family, and this is likely to remain true in future models (it is guaranteed in the Power ISA). An instruction having primary opcode 0 but not consisting entirely of binary 0s is reserved except for the following extended opcode (instruction bits 21:30).

256 Service Processor "Attention" (Power ISA only)

Table 1: Primary opcodes				
0 00 Illegal, Reserved	1 01	2 02 tdi 64 D	3 03 twi B D	See primary opcode 0 extensions on page 1371 Trap Doubleword Immediate Trap Word Immediate
4 04 Vector, LMA, SP V, LMA, SP	5 05	6 06	7 07 mulli BD	See Table 8 and Table 7 Multiply Low Immediate
8 08 subfic B D	9 09	10 0A cmpli B D	11 0B cmpi B D	Subtract From Immediate Carrying Compare Logical Immediate Compare Immediate
12 0C addic B D	13 0D addic. B D	14 0E addi B D	15 0F addis B D	Add Immediate Carrying Add Immediate Carrying and Record Add Immediate Add Immediate Shifted
16 10 bc B B	17 11 sc B SC	18 12 b B I	19 13 CR ops, etc. XL	Branch Conditional System Call Branch See Table 10 on page 1385
20 14 rlwimi B M	21 15 rlwinm B M	22 16	23 17 rlwnm B M	Rotate Left Word Imm. then Mask Insert Rotate Left Word Imm. then AND with Mask Rotate Left Word then AND with Mask
24 18 ori B D	25 19 oris B D	26 1A xori B D	27 1B xoris B D	OR Immediate OR Immediate Shifted XOR Immediate XOR Immediate Shifted
28 1C andi. B D	29 1D andis. B D	30 1E FX Dwd Rot MD[S]	31 1F FX Extended Ops	AND Immediate AND Immediate Shifted See Table 2 on page 1374 See Table 10 on page 1385
32 20 lwz B D	33 21 lwzu B D	34 22 lbz B D	35 23 lbzu B D	Load Word and Zero Load Word and Zero with Update Load Byte and Zero Load Byte and Zero with Update
36 24 stw B D	37 25 stwu B D	38 26 stb B D	39 27 stbu B D	Store Word Store Word with Update Store Byte Store Byte with Update
40 28 lhz B D	41 29 lhzu B D	42 2A lha B D	43 2B lhau B D	Load Half and Zero Load Half and Zero with Update Load Half Algebraic Load Half Algebraic with Update
44 2C sth B D	45 2D sthu B D	46 2E lmw B D	47 2F stmw B D	Store Half Store Half with Update Load Multiple Word Store Multiple Word
48 30 lfs FP D	49 31 lfsu FP D	50 32 lfd FP D	51 33 lfdu FP D	Load Floating-Point Single Load Floating-Point Single with Update Load Floating-Point Double Load Floating-Point Double with Update
52 34 stfs FP D	53 35 stfsu FP D	54 36 stfd FP D	55 37 stfdu FP D	Store Floating-Point Single Store Floating-Point Single with Update Store Floating-Point Double Store Floating-Point Double with Update
56 38 lq LSQ DQ	57 39	58 3A FX DS-form Loads DS	59 3B FP Single & DFP Ops	Load Quadword See Table 3 on page 1374 See Table 4 on page 1374 See Table 16 on page 1389

Table 1: Primary opcodes				
60 3C VSX Extended Ops	61 3D stfdp FP DS	62 3E FX DS-form Stores DS	63 3F FP Double &DFP Ops	Store Floating-Point Double Pair See Table 6 on page 1374 See Table 17 on page 1391 See Table 18 on page 1393

Table 2: Extended opcodes for primary opcode 30 (instruction bits 27:30)

	00	01	10	11
00	0 ***rldicl*** 64 MD	1 ***rldicl'*** MD	2 ***rldicr*** 64 MD	3 ***rldicr'*** MD
01	4 ***rldic*** 64 MD	5 ***rldic'*** MD	6 ***rldimi*** 64 MD	7 ***rldimi'*** MD
10	8 ***rldcl*** 64 MDS	9 ***rldcr*** 64 MDS		
11				

Table 3: Extended opcodes for primary opcode 57 (instruction bits 30:31)

	0	1
0	0 ***lfdp*** FP DS	
1		

Table 4: Extended opcodes for primary opcode 58 (instruction bits 30:31)

	0	1
0	0 ***ld*** 64 DS	1 ***ldu*** 64 DS
1	2 ***lwa*** 64 DS	

Table 5: Extended opcodes for primary opcode 61 (instruction bits 30:31)

	0	1
0	0 ***stfdp*** FP DS	
1		

Table 6: Extended opcodes for primary opcode 62 (instruction bits 30:31)

	0	1
0	0 ***std*** 64 DS	1 ***stdu*** 64 DS
1	2 ***stq*** LSQ DS	

Table 7: (Left) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)

	000000	000001	000010	000011	000100	000101	000110	000111	001000	001001	001010	001011	001100	001101	001110	001111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000	512 ***evaddw*** SP EVX		514 ***evaddiw*** SP EVX		516 ***evsubfw*** SP EVX		518 ***evsubifw*** SP EVX		520 ***evabs*** SP EVX	521 ***evneg*** SP EVX	522 ***evextsb*** SP EVX	523 ***evextsh*** SP EVX	524 ***evrndw*** SP EVX	525 ***evcntlzw*** SP EVX	526 ***evcntlsw*** SP EVX	527 ***brinc*** SP EVX
01001																
01010	640 ***evfsadd*** sp.fv EVX	641 ***evfssub*** sp.fv EVX			644 ***evfsabs*** sp.fv EVX	645 ***evfsnabs*** sp.fv EVX	646 ***evfsneg*** sp.fv EVX		648 ***evfsmul*** sp.fv EVX	649 ***evfsdiv*** sp.fv EVX			652 ***long*** sp.fv EVX	653 ***evfscmplt*** sp.fv EVX	654 ***long*** sp.fv EVX	
01011	704 ***efsadd*** sp.fs EVX	705 ***efssub*** sp.fs EVX			708 ***efsabs*** sp.fs EVX	709 ***efsnabs*** sp.fs EVX	710 ***efsneg*** sp.fs EVX		712 ***efsmul*** sp.fs EVX	713 ***efsdiv*** sp.fs EVX			716 ***efscmpgt*** sp.fs EVX	717 ***efscmplt*** sp.fs EVX	718 ***efscmpeq*** sp.fs EVX	719 ***efscfd*** sp.fd EVX
01100	768 ***evlddx*** SP EVX	769 ***evldd*** SP EVX	770 ***evldwx*** SP EVX	771 ***evldw*** SP EVX	772 ***evldhx*** SP EVX	773 ***evldh*** SP EVX			776 ***long*** SP EVX	777 ***long*** SP EVX			780 ***long*** SP EVX	781 ***long*** SP EVX	782 ***long*** SP EVX	783 ***long*** SP EVX
01101																
01110																
01111																
10000				1027 ***evmhessf*** SP EVX				1031 ***evmhossf*** SP EVX	1032 ***long*** SP EVX	1033 ***long*** SP EVX		1035 ***long*** SP EVX	1036 ***long*** SP EVX	1037 ***long*** SP EVX		1039 ***long*** SP EVX
10001								1095 ***long*** SP EVX	1096 ***long*** SP EVX				1100 ***long*** SP EVX	1101 ***long*** SP EVX		1103 ***long*** SP EVX
10010																
10011	1216 ***long*** SP EVX	1217 ***long*** SP EVX	1218 ***long*** SP EVX	1219 ***long*** SP EVX	1220 ***evmra*** SP EVX		1222 ***evdivws*** SP EVX	1223 ***evdivwu*** SP EVX	1224 ***long*** SP EVX	1225 ***long*** SP EVX	1226 ***long*** SP EVX	1227 ***long*** SP EVX				
10100	1280 ***long*** SP EVX	1281 ***long*** SP EVX		1283 ***long*** SP EVX	1284 ***long*** SP EVX	1285 ***long*** SP EVX		1287 ***long*** SP EVX	1288 ***long*** SP EVX	1289 ***long*** SP EVX		1291 ***long*** SP EVX	1292 ***long*** SP EVX	1293 ***long*** SP EVX		1295 ***long*** SP EVX
10101	1344 ***long*** SP EVX	1345 ***long*** SP EVX							1352 ***long*** SP EVX	1353 ***long*** SP EVX						
10110	1408 ***long*** SP EVX	1409 ***long*** SP EVX		1411 ***long*** SP EVX	1412 ***long*** SP EVX	1413 ***long*** SP EVX		1415 ***long*** SP EVX	1416 ***long*** SP EVX	1417 ***long*** SP EVX		1419 ***long*** SP EVX	1420 ***long*** SP EVX	1421 ***long*** SP EVX		1423 ***long*** SP EVX
10111	1472 ***long*** SP EVX	1473 ***long*** SP EVX							1480 ***long*** SP EVX	1481 ***long*** SP EVX						
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 7 (Left-Center) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)

	010000	010001	010010	010011	010100	010101	010110	010111	011000	011001	011010	011011	011100	011101	011110	011111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000		529 ***evand*** SP EVX	530 ***evandc*** SP EVX				534 ***evxor*** SP EVX	535 ***evor*** SP EVX	536 ***evnor*** SP EVX	537 ***eveqv*** SP EVX		539 ***evorc*** SP EVX			542 ***evnand*** SP EVX	
01001																
01010	656 ***evfscfui*** sp.fv EVX	657 ***evfscfsi*** sp.fv EVX	658 ***evfscfuf*** sp.fv EVX	659 ***evfscfsf*** sp.fv EVX	660 ***evfsctui*** sp.fv EVX	661 ***evfsctsi*** sp.fv EVX	662 ***evfsctuf*** sp.fv EVX	663 ***evfsctsf*** sp.fv EVX	664 ***evfsctuiz*** sp.fv EVX		666 ***evfsctsiz*** sp.fv EVX		668 ***evfststgt*** sp.fv EVX	669 ***evfststlt*** sp.fv EVX	670 ***evfststeq*** sp.fv EVX	
01011	720 ***efscfui*** sp.fs EVX	721 ***efscfsi*** sp.fs EVX	722 ***efscfuf*** sp.fs EVX	723 ***efscfsf*** sp.fs EVX	724 ***efsctui*** sp.fs EVX	725 ***efsctsi*** sp.fs EVX	726 ***efsctuf*** sp.fs EVX	727 ***efsctsf*** sp.fs EVX	728 ***efsctuiz*** sp.fs EVX		730 ***efsctsiz*** sp.fs EVX		732 ***efststgt*** sp.fs EVX	733 ***efststlt*** sp.fs EVX	734 ***efststeq*** sp.fs EVX	
01100	784 ***evlwhex*** SP EVX	785 ***evlwhe*** SP EVX			788 ***evlwhoux*** SP EVX	789 ***evlwhou*** SP EVX	790 ***evlwhosx*** SP EVX	791 ***evlwhos*** SP EVX	792 ***long*** SP EVX	793 ***long*** SP EVX			796 ***long*** SP EVX	797 ***long*** SP EVX		
01101																
01110																
01111																
10000																
10001				1107 ***evmwssf*** SP EVX					1112 ***long*** SP EVX	1113 ***long*** SP EVX		1115 ***long*** SP EVX				
10010																
10011																
10100																
10101				1363 ***long*** SP EVX					1368 ***long*** SP EVX	1369 ***long*** SP EVX		1371 ***long*** SP EVX				
10110																
10111				1491 ***long*** SP EVX					1496 ***long*** SP EVX	1497 ***long*** SP EVX		1499 ***long*** SP EVX				
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 7 (Right-Center) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)

	100000	100001	100010	100011	100100	100101	100110	100111	101000	101001	101010	101011	101100	101101	101110	101111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000	544 ***evsrwu*** SP EVX	545 ***evsrws*** SP EVX	546 ***evsrwiu*** SP EVX	547 ***evsrwis*** SP EVX	548 ***evslw*** SP EVX		550 ***evslwi*** SP EVX		552 ***evrlw*** SP EVX	553 ***evsplati*** SP EVX	554 ***evrlwi*** SP EVX	555 ***evsplatfi*** SP EVX	556 *long* SP EVX	557 long SP EVX	558 long SP EVX	559 ***long*** SP EVX
01001																
01010																
01011	736 ***efdadd*** sp.fdEVX	737 ***efdsub*** sp.fdEVX	738 ***efdcfuid*** sp.fdEVX	739 ***efdcfsid*** sp.fdEVX	740 ***efdabs*** sp.fdEVX	741 ***efdnabs*** sp.fdEVX	742 ***efdneg*** sp.fdEVX		744 ***efdmul*** sp.fdEVX	745 ***efddiv*** sp.fdEVX	746 ***efdctuidz*** sp.fdEVX	747 ***efdctsidz*** sp.fdEVX	748 ***efdcmpgt*** sp.fdEVX	749 ***efdcmplt*** sp.fdEVX	750 ***efdcmpeq*** sp.fdEVX	751 ***efdcfs*** sp.fdEVX
01100	800 ***evstddx*** SP EVX	801 ***evstdd*** SP EVX	802 ***evstdwx*** SP EVX	803 ***evstdw*** SP EVX	804 ***evstdhx*** SP EVX	805 ***evstdh*** SP EVX										
01101																
01110																
01111																
10000				1059 ***long*** SP EVX				1063 ***long*** SP EVX	1064 ***long*** SP EVX	1065 ***long*** SP EVX		1067 ***long*** SP EVX	1068 ***long*** SP EVX	1069 ***long*** SP EVX		1071 ***long*** SP EVX
10001								1127 ***long*** SP EVX	1128 ***long*** SP EVX				1132 ***long*** SP EVX	1133 ***long*** SP EVX		1135 ***long*** SP EVX
10010																
10011																
10100									1320 ***long*** SP EVX	1321 ***long*** SP EVX		1323 ***long*** SP EVX	1324 ***long*** SP EVX	1325 ***long*** SP EVX		1327 ***long*** SP EVX
10101																
10110									1448 ***long*** SP EVX	1449 ***long*** SP EVX		1451 ***long*** SP EVX	1452 ***long*** SP EVX	1453 ***long*** SP EVX		1455 ***long*** SP EVX
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 7 (Right) Extended opcodes for primary opcode 4 [Category: SP.*] (instruction bits 21:31)

	110000	110001	110010	110011	110100	110101	110110	110111	111000	111001	111010	111011	111100	111101	111110	111111
00000																
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000	560 ***evcmpgtu*** SP EVX	561 ***evcmpgts*** SP EVX	562 ***evcmpltu*** SP EVX	563 ***evcmplts*** SP EVX	564 ***evcmpeq*** SP EVX											
01001									632 ***evsel*** SP EVS	633 ***evsel'*** SP EVS	634 ***evsel'*** SP EVS	635 ***evsel'*** SP EVS	636 ***evsel'*** SP EVS	637 ***evsel'*** SP EVS	638 ***evsel'*** SP EVS	639 ***evsel'*** SP EVS
01010																
01011	752 ***efdcfui*** sp.fdEVX	753 ***efdcfsi*** sp.fdEVX	754 ***efdcfuf*** sp.fdEVX	755 ***efdcfsf*** sp.fdEVX	756 ***efdctui*** sp.fdEVX	757 ***efdctsi*** sp.fdEVX	758 ***efdctuf*** sp.fdEVX	759 ***efdctsf*** sp.fdEVX	760 ***efdctuiz*** sp.fdEVX		762 ***efdctsiz*** sp.fdEVX		764 ***efdtstgt*** sp.fdEVX	765 ***efdtstlt*** sp.fdEVX	766 ***efdtsteq*** sp.fdEVX	
01100	816 ***evstwhex*** SP EVX	817 ***evstwhe*** SP EVX			820 ***evstwhox*** SP EVX	821 ***evstwho*** SP EVX			824 ***evstwwex*** SP EVX	825 ***evstwwe*** SP EVX			828 ***evstwwox*** SP EVX	829 ***evstwwo*** SP EVX		
01101																
01110																
01111																
10000																
10001				1139 ***long*** SP EVX					1144 ***long*** SP EVX	1145 ***long*** SP EVX		1147 ***long*** SP EVX				
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 8: (Left) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

	000000	000001	000010	000011	000100	000101	000110	000111	001000	001001	001010	001011	001100	001101	001110	001111
00000	0 **vaddubm** V VX		2 **vmaxub** V VX		4 **vrlb** V VX		6 *vcmpequb* V VC		8 **vmuloub** V VX		10 **vaddfp** V VX		12 **vmrghb** V VX		14 **vpkuhum** V VX	
00001	64 *vadduhm* V VX		66 **vmaxuh** V VX		68 **vrlh** V VX		70 *vcmpequh* V VC		72 **vmulouh** V VX		74 **vsubfp** V VX		76 **vmrghh** V VX		78 *vpkuwum* V VX	
00010	128 *vadduwm* V VX		130 **vmaxuw** V VX		132 **vrlw** V VX		134 *vcmpequw* V VC		136 **vmulouw** V VC	137 **vmuluwm** V VC			140 **vmrghw** V VX		142 **vpkuhus** V VX	
00011	192 *vaddudm* V VX		194 *vmaxud* V VX		196 *vrld* V VX		198 *vcmpeqfp* V VC	199 *vcmpequd* V VC							206 **vpkuwus** V VX	
00100	256 *vadduqm* V VX		258 **vmaxsb** V VX		260 **vslb** V VX				264 **vmulosb** V VX		266 **vrefp** V VX		268 **vmrglb** V VX		270 **vpkshus** V VX	
00101	320 *vaddcuq* V VX		322 **vmaxsh** V VX		324 **vslh** V VX				328 **vmulosh** V VX		330 **vrsqrtefp** V VX		332 **vmrglh** V VX		334 **vpkswus** V VX	
00110	384 **vaddcuw** V VX		386 **vmaxsw** V VX		388 **vslw** V VX				392 **vmulosw** V VC		394 **vexptefp** V VX		396 **vmrglw** V VX		398 **vpkshss** V VX	
00111			450 *vmaxsd* V VX		452 **vsl** V VX		454 *vcmpgefp* V VC				458 **vlogefp** V VX				462 **vpkswss** V VX	
01000	512 **vaddubs** V VX		514 **vminub** V VX		516 **vsrb** V VX		518 *vcmpgtub* V VC		520 **vmuleub** V VX		522 **vrfin** V VX		524 **vspltb** V VX		526 **vupkhsb** V VX	
01001	576 **vadduhs** V VX		578 **vminuh** V VX		580 **vsrh** V VX		582 *vcmpgtuh* V VC		584 **vmuleuh** V VX		586 **vrfiz** V VX		588 **vsplth** V VX		590 **vupkhsh** V VX	
01010	640 **vadduws** V VX		642 **vminuw** V VX		644 **vsrw** V VX		646 *vcmpgtuw* V VC		648 **vmuleuw** V VC		650 **vrfip** V VX		652 **vspltw** V VX		654 **vupklsb** V VX	
01011			706 *vminud* V VX		708 **vsr** V VX		710 *vcmpgtfp* V VC	711 *vcmpgtud* V VC			714 **vrfim** V VX				718 **vupklsh** V VX	
01100	768 **vaddsbs** V VX		770 **vminsb** V VX		772 **vsrab** V VX		774 *vcmpgtsb* V VC		776 **vmulesb** V VX		778 **vcfux** V VX		780 **vspltisb** V VX		782 **vpkpx** V VX	
01101	832 **vaddshs** V VX		834 **vminsh** V VX		836 **vsrah** V VX		838 *vcmpgtsh* V VC		840 **vmulesh** V VX		842 **vcfsx** V VX		844 **vspltish** V VX		846 **vupkhpx** V VX	
01110	896 **vaddsws** V VX		898 **vminsw** V VX		900 **vsraw** V VX		902 *vcmpgtsw* V VC		904 **vmulesw** V VC		906 *vctuxs* V VX		908 **vspltisw** V VX			
01111			962 *vminsd* V VX		964 *vsrad* V VX		966 *vcmpbfp* V VC	967 *vcmpgtsd* V VC			970 **vctsxs** V VX				974 **vupklpx** V VX	
10000	1024 **vsububm** V VX	1025 **bcdadd.** V VX	1026 **vavguh** V VX		1028 **vand** V VX		1030 *vcmpequb.* V VC		1032 **vpmsumb** V VX		1034 **vmaxfp** V VX		1036 **vslo** V VX			
10001	1088 *vsubuhm* V VX	1089 **bcdsub.** V VX	1090 **vavguh** V VX		1092 **vandc** V VX		1094 *vcmpequh.* V VC		1096 **vpmsumh** V VX		1098 **vminfp** V VX		1100 **vsro** V VX		1102 *vpkudum* V VX	
10010	1152 *vsubuwm* V VX		1154 **vavguw** V VX		1156 **vor** V VX		1158 *vcmpequw.* V VC		1160 **vpmsumw** V VX							
10011	1216 *vsubudm* V VX				1220 **vxor** V VX		1222 *vcmpeqfp.* V VC	1223 *vcmpequd.* V VC	1224 **vpmsumd** V VX						1230 *vpkudus* V VX	
10100	1280 *vsubuqm* V VX		1282 **vavgsb** V VX		1284 **vnor** V VX				1288 **vcipher** V.Cry VX	1289 **vcipherlast** V.Cry VX			1292 *vgbbd* V VX			
10101	1344 *vsubcuq* V VX		1346 **vavgsh** V VX		1348 *vorc* V VX				1352 **vncipher** V.Cry VX	1353 **vncipherlast** V.Cry VX			1356 *vbpermq* V VX		1358 *vpksdus* V VX	
10110	1408 **vsubcuw** V VX		1410 **vavgsw** V VX		1412 *vnand* V VX											
10111					1476 *vsld* V VX		1478 *vcmpgefp.* V VC		1480 **vsbox** V.Cry VX						1486 *vpksdss* V VX	
11000	1536 **vsububs** V VX	1537 **bcdadd.** V VX			1540 **mfvscr** V VX		1542 *vcmpgtub.* V VC		1544 *vsum4ubs* V VX							
11001	1600 **vsubuhs** V VX	1601 **bcdsub.** V VX			1604 **mtvscr** V VX		1606 *vcmpgtuh.* V VC		1608 *vsum4shs* V VX						1614 *vupkhsw* V VX	
11010	1664 **vsubuws** V VX		1666 **vshasigmaw** V.Cry VX		1668 *veqv* V VX		1670 *vcmpgtuw.* V VC		1672 *vsum2sws* V VX				1676 *vmrgeo* V VX			
11011			1666 **vshasigmad** V.Cry VX		1732 *vsrd* V VX		1734 *vcmpgtfp.* V VC	1735 *vcmpgtud.* V VC							1742 *vupklsw* V VX	
11100	1792 **vsubsbs** V VX		1794 *vclzb* V VX	1795 *vpopcntb* V VX			1798 *vcmpgtsb.* V VC		1800 *vsum4sbs* V VX							
11101	1856 **vsubshs** V VX		1858 *vclzh* V VX	1859 *vpopcnth* V VX			1862 *vcmpgtsh.* V VC									
11110	1920 **vsubsws** V VX		1922 *vclzw* V VX	1923 *vpopcntw* V VX			1926 *vcmpgtsw.* V VC		1928 *vsumsws* V VX				1932 *vmrgew* V VX			
11111			1986 *vclzd* V VX	1987 *vpopcntd* V VX			1990 *vcmpbfp.* V VC	1991 *vcmpgtsd.* V VC								

Table 8 (Left-Center) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

	010000	010001	010010	010011	010100	010101	010110	010111	011000	011001	011010	011011	011100	011101	011110	011111
00000	16 ***mulhhwu*** LMA X	17 ***mulhhwu.*** LMA X							24 ***machhwu*** LMA XO	24 ***long*** LMA XO						
00001	80 ***mulhhw*** LMA X	81 ***mulhhw.*** LMA X							88 ***machhw*** LMA XO	89 ***machhw.*** LMA XO			92 ***nmachhw*** LMA XO	93 ***long*** LMA XO		
00010									152 ***long*** LMA XO	153 ***long*** LMA XO						
00011									216 ***machhws*** LMA XO	217 ***long*** LMA XO			220 ***long*** LMA XO	220 ***long*** LMA XO		
00100	272 ***mulchwu*** LMA X	273 ***mulchwu.*** LMA X							280 ***macchwu*** LMA XO	281 ***long*** LMA XO						
00101	336 ***mulchw*** LMA X	337 ***mulchw.*** LMA X							344 ***macchw*** LMA XO	345 ***macchw.*** LMA XO			348 ***nmacchw*** LMA XO	349 ***long*** LMA XO		
00110									408 ***long*** LMA XO	409 ***long*** LMA XO						
00111									472 ***macchws*** LMA XO	473 ***long*** LMA XO			476 ***long*** LMA XO	477 ***long*** LMA XO		
01000																
01001																
01010																
01011																
01100	784 ***mullhwu*** LMA X	784 ***mullhwu.*** LMA X							792 ***maclhwu*** LMA XO	793 ***maclhwu.*** LMA XO						
01101	848 ***mullhw*** LMA X	849 ***mullhw.*** LMA X							856 ***maclhw*** LMA XO	857 ***maclhw.*** LMA XO			860 ***nmaclhw*** LMA XO	861 ***nmaclhw.*** LMA XO		
01110									920 ***long*** LMA XO	921 ***long*** LMA XO						
01111									984 ***maclhws*** LMA XO	985 ***maclhws.*** LMA XO			988 ***long*** LMA XO	989 ***long*** LMA XO		
10000									1048 ***long*** LMA XO	1049 ***long*** LMA XO						
10001									1112 ***machhw'*** LMA XO	1113 ***long*** LMA XO			1116 ***long*** LMA XO	1117 ***long*** LMA XO		
10010									1176 ***long*** LMA XO	1177 ***long*** LMA XO						
10011									1240 ***long*** LMA XO	1241 ***long*** LMA XO			1244 ***long*** LMA XO	1245 ***long*** LMA XO		
10100									1304 ***long*** LMA XO	1305 ***long*** LMA XO						
10101									1368 ***macchw'*** LMA XO	1369 ***long*** LMA XO			1372 ***long*** LMA XO	1373 ***long*** LMA XO		
10110									1432 ***long*** LMA XO	1433 ***long*** LMA XO						
10111									1496 ***long*** LMA XO	1497 ***long*** LMA XO			1500 ***long*** LMA XO	1501 ***long*** LMA XO		
11000																
11001																
11010																
11011																
11100									1816 ***long*** LMA XO	1817 ***long*** LMA XO						
11101									1880 ***maclhw'*** LMA XO	1881 ***maclhw'*** LMA XO			1884 ***long*** LMA XO	1885 ***long*** LMA XO		
11110									1944 ***long*** LMA XO	1946 ***long*** LMA XO						
11111									2008 ***long*** LMA XO	2009 ***long*** LMA XO			2012 ***long*** LMA XO	2013 ***long*** LMA XO		

Table 8 (Right-Center) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

	100000	100001	100010	100011	100100	100101	100110	100111	101000	101001	101010	101011	101100	101101	101110	101111
00000	32 *vmhaddshs* V VA	32 *vmhraddshs* V VA	34 *vmladduhm* V VA		36 *vmsumubm* V VA	37 *vmsummbm* V VA	38 *vmsumuhm* V VA	39 *vmsumuhs* V VA	40 *vmsumshm* V VA	41 *vmsumshs* V VA	42 ***vsel*** V VA	43 ***vperm*** V VA	44 ***vsldoi*** V VA	45 ***vpermxor*** V.RAID VA	46 ***vmaddfp*** V VA	47 ***vnmsubfp*** V VA
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111	*vmhaddshs*	*vmhraddshs*	*vmladduhm*		*vmsumubm*	*vmsummbm*	*vmsumuhm*	*vmsumuhs*	*vmsumshm*	*vmsumshs*	***vsel***	***vperm***	***vsdoi***	***vpermxor***	***vmaddfp***	***vnmsubfp***

Table 8 (Right) Extended opcodes for primary opcode 4 [Category: V & LMA] (instruction bits 21:31)

	110000	110001	110010	110011	110100	110101	110110	110111	111000	111001	111010	111011	111100	111101	111110	111111
00000													60 *vaddeuqm* V VA	61 *vaddecuq* V VA	62 *vsubeuqm* V VA	63 *vsubecuq* V VA
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 9: (Left) Extended opcodes for primary opcode 19 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000	0 ***mcrf*** B XL															
00001		33 ***crnor*** B XL					38 ***rfmci*** E XL	39 ***rfdi*** E.ED X								
00010																
00011							102 ***rfgi*** E XL									
00100		129 ***crandc*** B XL														
00101																
00110		193 ***crxor*** B XL					198 ***dnh*** E.EDXFX									
00111		225 ***crnand*** B XL														
01000		257 ***crand*** B XL														
01001		289 ***creqv*** B XL														
01010																
01011																
01100																
01101		417 ***crorc*** B XL														
01110		449 ***cror*** B XL														
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 9. (Right) Extended opcodes for primary opcode 19 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	16 ***bclr*** B XL		18 ***rfid*** S XL													
00001			50 ***rfi*** E XL	51 ***rfci*** E XL												
00010			(82) ***rfsvc*** XL													
00011																
00100			146 ***rfebb*** S XL				150 ***isync*** B XL									
00101																
00110																
00111																
01000			274 ***hrfid*** S XL													
01001																
01010																
01011																
01100			402 ***doze*** S XL													
01101			434 ***nap*** S XL													
01110			466 ***sleep*** S XL													
01111			498 ***rvwinkle*** S XL													
10000	528 ***bcctr*** B XL															
10001	560 ***bctar[l]*** B XL															
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111																

Table 10:(Left) Extended opcodes for primary opcode 31 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000	0 ***cmp*** B X				4 ***tw*** B X		6 ***lvsl*** V X	7 ***lvebx*** V X	8 ***subfc*** B XO	9 ***mulhdu*** 64 XO	10 ***addc*** B XO	11 ***mulhwu*** B XO	12 ***lxsiwzx*** VSX XX		14 Res'd VLE	15 See Table 15
00001	32 ***cmpl*** B X	33 Res'd VLE					38 ***lvsr*** V X	39 ***lvehx*** V X	40 ***subf*** B XO						46 Res'd VLE	
00010					68 ***td*** 64 X			71 ***lvewx*** V X		73 ***mulhd*** 64 XO	74 ***addg6s'*** BCDA XO	75 ***mulhw*** B XO	76 ***lxsiwax*** VSX XX		78 ***dlmzb*** LMV X	
00011								103 ***lvx*** V X	104 ***neg*** B XO							
00100		129 Res'd VLE		131 ***wrtee*** E X			134 ***dcbtstls*** ECL X	135 ***stvebx*** V X	136 ***subfe*** B XO		138 ***adde*** B XO		138 ***stxsiwx*** VSX XX		140 ***msgsndp*** S.PC X	
00101				163 ***wrteei*** E X			166 ***dcbtls*** ECL X	167 ***stvehx*** V X							174 ***msgclrp*** S.PC X	
00110		193 Res'd VLE					198 ***icblq.*** ECL X	199 ***stvewx*** V X	200 ***subfze*** B XO		202 ***addze*** B XO				206 ***msgsnd*** E/S.PC X	
00111		225 Res'd VLE					230 ***icblc*** ECL X	231 ***stvx*** V X	232 ***subfme*** B XO	233 ***mulld*** 64 XO	234 ***addme*** B XO	235 ***mullw*** B XO			238 ***msgclr*** E/S.PC X	
01000		257 Res'd VLE		259 ***mfdcrx*** E.DC X			262 Res'd AP	263 ***lvepxl*** E.PD X			266 ***add*** B XO				270 ***ehpriv*** E.HV XL	
01001		289 Res'd VLE		291 ***mfdcrux*** E.DC X				295 ***lvepx*** E.PD X							302 ***mfbhrbe*** S X	
01010				323 ***mfdcr*** E.DC XFX			326 ***dcread*** E.CD X						332 ***lxvdsx*** VSX XX		334 ***mfpmr*** E.PM XFX	
01011								359 ***lvxl*** V X							{366} ***mftmr***	
01100				387 ***mtdcrx*** E.DC X			390 ***dcblc*** ECL X			393 ***divdeu'*** 64 XO		395 ***divweu*** B XO			398 ***mvptas***	
01101		417 Res'd VLE		419 ***mtdcrux*** E.DC X			422 ***dcblq.*** ECL X			425 ***divde*** 64 XO		427 ***divwe*** B XO			430 ***clrbhrb*** S X	
01110		449 Res'd VLE		451 ***mtdcr*** E.DC XFX			454 ***dci*** E.CI X			457 ***divdu*** 64 XO		459 ***divwu*** B XO			462 ***mtpmr*** E.PM XFX	
01111				483 ***dsn*** DS X			486 Res'd AP	487 ***stvxl*** V X		489 ***divd*** 64 XO		491 ***divw*** B XO			494 ***mttmr*** EMT XFX	
10000	512 ***mcrxr*** E X			515 *lbdx* DS X				519 Res'd V X	520 ***subfc'*** B XO	521 ***mulhdu'*** 64XO	522 ***addc'*** B XO	523 ***mulhwu'*** B XO	524 ***lxsspx*** VSX XX			
10001				547 *lhdx* DS X				551 Res'd V X	552 ***subf'*** B XO							
10010				579 *lwdx* DS X						585 ***mulhd'*** 64 XO	586 ***addg6s*** BCDA XO	587 ***mulhw'*** B XO	588 ***lxsdx*** VSX XX			
10011				611 *lddx* DS X					616 ***neg'*** B XO							
10100				643 *stbdx* DS X				647 Res'd V X	648 ***subfe'*** B XO		650 ***adde'*** B XO		652 ***stxsspx*** VSX XX		654 ***tbegin.*** TM X	
10101				675 *sthdx* DS X				679 Res'd V X							686 ***tend.*** TM X	
10110				707 *stwdx* DS X					712 ***subfze'*** B XO		714 ***addze'*** B XO		716 ***stxsdx*** VSX XX		718 ***tcheck*** TM X	
10111				739 *stddx* DS X					744 ***subfme'*** B XO	745 ***mulld'*** 64 XO	746 ***addme'*** B XO	747 ***mullw'*** B XO			750 ***tsr.*** TM X	
11000								775 ***stvepxl*** E.PD X			778 ***add'*** B XO		780 ***lxvw4x*** VSX XX		782 ***tabortwc.*** TM X	
11001				803 *lfddx* DS X				807 ***stvepx*** E.PD X							814 ***tabortdc.*** TM X	
11010													844 ***lxvd2x*** VSX XX		846 ***tabortwci.*** TM X	
11011															878 ***tabortdci.*** TM X	
11100								903 Res'd V X		905 ***divdeu'*** 64 XO		907 ***divweu'*** B XO	908 ***stxvw4x*** VSX XX		910 ***tabort.*** TM X	
11101				931 ***stfddx*** DS X				935 Res'd V X		937 ***divde'*** 64 XO		939 ***divwe'*** 64 XO			942 ***treclaim.*** TM X	
11110							966 ***ici*** E.CI X			969 ***divdu'*** 64XO		971 ***divwu'*** 64XO	972 ***stxvd2x*** VSX XX			
11111							998 ***icread*** E.CD X			1001 ***divd'*** 64 XO		1003 ***divw'*** B XO			1006 ***trechkpt.*** TM X	See Table 15

Table 10. (Right) Extended opcodes for primary opcode 31 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	16 Res'd VLE		18 ***tlbilx*** E X	19 ***mfcr*** B XFX	20 ***lwarx*** B X	21 ***ldx*** 64 X	22 ***icbt*** B X	23 ***lwzx*** B X	24 ***slw*** B X		26 ***cntlzw*** B X	27 ***sld*** 64 X	28 ***and*** B X	29 ***ldepx*** E.PD X	30 See Table 11	31 ***lwepx*** E.PD X
00001				51 ***mfvsrd*** VSX XX	52 ***lbarx*** B X	53 ***ldux*** 64 X	54 ***dcbst*** B X	55 ***lwzux*** B X	56 Res'd VLE		58 ***cntlzd*** 64 X		60 ***andc*** B X		62 See Table 12	
00010			(82) ***mtsrd*** X	83 ***mfmsr*** B X	84 ***ldarx*** 64 X		86 ***dcbf*** B X	87 ***lbzx*** B X							94 ***rldicr**** 64 MD	95 ***lbepx*** E.PD X
00011			(114) ***mtsrdin*** X	115 ***mfvsrwz*** VSX XX	116 ***lharx*** B X		(118) ***clf*** X	119 ***lbzux*** B X			122 ***popcntb*** B X		124 ***nor*** B X		126 ***rldicr**** 64 MD	127 ***dcbfep*** E.PD X
00100	144 ***mtcrf*** B XFX		146 ***mtmsr*** B X	147 ***mtsle*** S X		149 ***stdx*** 64 X	150 ***stwcx.*** B X	151 ***stwx*** B X			154 ***prtyw*** B X			157 ***stdepx*** E.PD;64 X	158 ***rldic**** 64 MD	159 See Table 14
00101			178 ***mtmsrd*** S X	179 ***mtvsrd*** VSX XX		181 ***stdux*** 64 X	182 ***stqcx.*** LSQ X	183 ***stwux*** B X			186 ***prtyd*** 64 X				190 ***rldic**** 64 MD	191 ***rlwinm**** B M
00110			210 ***mtsr*** S X	211 ***mtvsrwa*** VSX XX		214 ***stdcx.*** 64 X	215 ***stbx*** B X								222 ***rldimi**** 64 MD	223 ***stbepx*** E.PD X
00111			242 ***mtsrin*** S X	242 ***mtvsrwz*** VSX XX		246 ***dcbtst*** B X	247 ***stbux*** B X				250 Res'd		252 ***bpermd*** 64 X		254 ***rldimi**** 64 MD	255 See Table 14
01000			274 ***tlbiel*** S X		276 ***lqarx*** LSQ X		278 ***dcbt*** B X	279 ***lhzx*** B X	280 Res'd VLE		282 ***cdtbcd*** BCDA X		284 ***eqv*** B X		286 ***rldcl**** 64 MDS	286 See Table 14
01001			306 ***tlbie*** S X		308 Res'd		310 ***eciwx*** EC X	311 ***lhzux*** B X	312 Res'd VLE		314 ***cbcdtd*** BCDA X		316 ***xor*** B X		318 ***rldcr**** 64 MDS	319 See Table 14
01010				339 ***mfspr*** B XFX		341 ***lwax*** 64 X	342 Res'd AP	343 ***lhax*** B X							350 *	351 ***xori**** B D
01011			370 ***tlbia*** S X	371 ***mftb*** S XFX		373 ***lwaux*** 64 X	374 Res'd AP	375 ***lhaux*** B X			378 ***popcntw*** B X				382 *	383 ***xoris**** B D
01100			402 ***slbmte*** S X					407 ***sthx*** B X					412 ***orc*** B X		414 *	415 See Table 14
01101			434 ***slbie*** S X				438 ***ecowx*** EC X	439 ***sthux*** B X					444 ***or*** B X		446 *	447 ***andis.**** B D
01110				467 ***mtspr*** B XFX		469 *	470 ***dcbi*** E X	471 ***lmw**** All D					476 ***nand*** B X		478 *	
01111			498 ***slbia*** S X			501 *		503 ***stmw**** All D			506 ***popcntd*** 64 X		508 ***cmpb*** BX		510 *	
10000			(530) no-op		532 ***ldbrx*** 64 X	533 ***lswx*** MA X	534 ***lwbrx*** B X	535 ***lfsx*** FP X	536 ***srw*** B X			539 ***srd*** 64 X				
10001			(562) no-op				566 ***tlbsync*** B X	567 ***lfsux*** FP X	568 Res'd VLE							
10010			(594) no-op	595 ***mfsr*** S X		597 ***lswi*** MA X	598 ***sync*** B X	599 ***lfdx*** FP X								607 ***lfdepx*** E.PD X
10011			(626) no-op					631 ***lfdux*** FP X								
10100			(658) no-op	659 ***mfsrin*** S X	660 ***stdbrx*** 64 X	661 ***stswx*** MA X	662 ***stwbrx*** B X	663 ***stfsx*** FP X								
10101			(690) no-op				694 ***stbcx.*** B X	695 ***stfsux*** FP X								
10110			(722) no-op			725 ***stswi*** MA X	726 ***sthcx.*** B X	727 ***stfdx*** FP X								735 ***stfdepx*** E.PD X
10111			(754) no-op				758 ***dcba*** E X	759 ***stfdux*** FP X								
11000			786 ***tlbivax*** E X			789 ***lwzcix*** S X	790 ***lhbrx*** B X	791 ***lfdpx*** FP X	792 ***sraw*** B X		794 ***srad*** 64 X					799 ***evlddepx*** E.PD evx
11001			(818) ***rac*** X			821 ***lhzcix*** S X	822 Res'd	823 Res'd	824 ***srawi*** B X		826 ***sradi*** 64 XS	827 ***sradi'*** 64 XS				
11010			850 ***tlbsrx.*** E.TWC X	851 ***slbmfev*** S X		853 ***lbzcix*** S X	854 See Table 13	855 ***lfiwax*** FP X								
11011						885 ***ldcix*** S X		887 ***lfiwzx*** FP X								
11100			914 ***tlbsx*** E X	915 ***slbmfee*** S X		917 ***stwcix*** S X	918 ***sthbrx*** B X	919 ***stfdpx*** FP X			922 ***extsh*** B X					927 ***evstddepx*** E.PD evx
11101			946 ***tlbre*** E X			949 ***sthcix*** S X		951 Res'd AP			954 ***extsb*** B X					
11110			978 ***tlbwe*** E X	979 ***slbfee.*** S X		981 ***stbcix*** S X	982 ***icbi*** B X	983 ***stfiwx*** FP X			986 ***extsw*** 64 X					991 ***icbiep*** E.PD X
11111			1010 Res'd			1013 ***stdcix*** S X	1014 ***dcbz*** B X									1023 ***dcbzep*** E.PD X

Table 11:Opcode: 31, Extended Opcode: 30

0	11110	
00000	30 ***rldicl**** 64 MD	

Table 12:Opcode: 31, Extended Opcode: 62

0	11110	
00001	62 ***rldicl**** 64 MD	62 ***wait*** WT X

Table 13:Opcode: 31, Extended Opcode: 854

	10110	
11010	854 ***eieio*** S X	854 ***mbar*** E X

Table 14:Opcode: 31, Extended Opcode: 159

	11111	
00100	159 ***rlwimi**** B M	159 ***stwepx*** E.PD X
00101	191 ***rlwinm**** B M	
00110		223 ***stbepx*** E.PD X
00111	255 ***rlwnm**** B M	
01000	287 ***ori**** B D	287 ***lhepx*** E.PD X
01001	319 ***oris**** B D	319 ***dcbtep*** E.PD X
01010	351 ***xori**** B D	
01011	383 ***xoris**** B D	
01100	415 ***andi.**** B D	415 ***sthepx*** E.PD X

Table 15:Opcode: 31, Extended Opcode: 15

	01111	
00000		15 ***isel*** B A
00001	47 *	
00010	79 ***tdi**** 64 D	
00011	111 ***twi**** B D	
00100	143 *	
00101	175 *	
00110	207 *	
00111	239 ***mulli**** B D	
01000	271 ***subfic**** B D	
01001		
01010	335 ***cmpli**** B D	
01011	367 ***cmpi**** B D	
01100	399 ***addic**** B D	
01101	431 ***addic.**** B D	
01110	463 ***addi**** B D	
01111	495 ***addis**** B D	
10000		
10001		
10010		
10011		
10100		
10101		
10110		
10111		
11000		
11001		
11010		
11011		
11100		
11101		
11110		
11111		***isel***

Table 16:(Left) Extended opcodes for primary opcode 59 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000			2 ***dadd*** DFP X	3 ***dqua*** DFP Z												
00001			34 ***dmul*** DFP X	35 ***drrnd*** DFP Z												
00010			66 ***dscli*** DFP Z22	67 ***dquai*** DFP Z												
00011			98 ***dscri*** DFP Z	99 ***drintx*** DFP Z23												
00100			130 ***dcmpo*** DFP X													
00101			162 ***dtstex*** DFP X													
00110			194 ***dtstdc*** DFP Z23													
00111			226 ***dtstdg*** DFP Z23	227 ***drintn*** DFP Z23												
01000			258 ***dctdps*** DFP X	259 ***dqua'*** DFP Z												
01001			290 ***dctfix*** DFP X	291 ***drrnd'*** DFP Z												
01010			322 ***ddedpd*** DFP X	323 ***dquai'*** DFP Z												
01011			354 ***dxex*** DFP X	355 ***drintx'*** DFP Z23												
01100																
01101																
01110																
01111				227 ***drintn'*** DFP Z23												
10000			514 ***dsub*** DFP X	515 ***dqua'*** DFP Z												
10001			546 ***ddiv*** DFP X	547 ***drrnd'*** DFP Z												
10010			578 ***dscli'*** DFP Z22	579 ***dquai'*** DFP Z												
10011			610 ***dscri'*** DFP Z22	611 ***drintx'*** DFP Z23												
10100			642 ***dcmpu*** DFP X													
10101			674 ***dtstsf*** DFP X													
10110			706 ***dtstdc'*** DFP Z23													
10111			738 ***dtstdg'*** DFP Z23	739 ***drintn'*** DFP Z23												
11000			770 ***drsp*** DFP X	771 ***dqua'*** DFP Z												
11001			802 ***dcffix*** DFP X	803 ***drrnd'*** DFP Z												
11010			834 ***denbcd*** DFP X	835 ***dquai'*** DFP Z											846 ***fcfids*** FP X	
11011			866 ***diex*** DFP X	867 ***drintx'*** DFP Z23												
11100																
11101																
11110															974 ***fcfidus*** FP X	
11111				995 ***drintn'*** DFP Z23												

Table 16. (Right) Extended opcodes for primary opcode 59 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000			18 ***fdivs*** FP A		20 ***fsubs*** FP A	21 ***fadds*** FP A	22 ***fsqrts*** FP A		24 ***fres*** FP A	25 ***fmuls*** FP A	26 ***frsqrtes*** FP A		28 ***fmsubs*** FP A	29 ***fmadds*** FP A	30 ***fnmsubs*** FP A	31 ***fnmadds*** FP A
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111			***fdivs***		***fsubs***	***fadds***	***fsqrts***		***fres***	***fmuls***	***frsqrtes***		***fmsub***	***fmadds***	***fnmsubs***	***fnmadds***

Table 17.(Left) Extended opcodes for primary opcode 60 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000	0 **xsaddsp** VSX XX				4 **xsmaddasp** VSX XX				8 **xxsldwi** VSX XX							
00001	32 **xssubsp** VSX XX				36 **xsmaddmsp** VSX XX				40 **xxpermdi** VSX XX							
00010	64 **xsmulsp** VSX XX				68 **xsmsubasp** VSX XX				72 **xxmrglw** VSX XX							
00011	96 **xsdivsp** VSX XX				100 **xsmsubmsp** VSX XX											
00100	128 **xsadddp** VSX XX				132 **xsmaddadp** VSX XX				136 **xxsldwi** VSX XX				140 **xscmpudp** VSX XX			
00101	160 **xssubdp** VSX XX				164 **xsmaddmdp** VSX XX				168 **xxpermdi** VSX XX				172 **xscmpodp** VSX XX			
00110	192 **xsmuldp** VSX XX				196 **xsmsubadp** VSX XX				200 **xxmrglw** VSX XX							
00111	224 **xsdivdp** VSX XX				228 **xsmsubmdp** VSX XX											
01000	256 **xvdivsp** VSX XX				260 **xvmaddasp** VSX XX				264 **xxsldwi** VSX XX				268 **xvcmpeqsp** VSX XX			
01001	288 **xvdivsp** VSX XX				292 **xvmaddmsp** VSX XX				296 **xxpermdi** VSX XX				300 **xvcmpgtsp** VSX XX			
01010	320 **xvdivsp** VSX XX				324 **xvmsubasp** VSX XX				328 **xxspltw** VSX XX				332 **xvcmpgesp** VSX XX			
01011	352 **xvdivsp** VSX XX				356 **xvmsubmsp** VSX XX											
01100	384 **xvdivdp** VSX XX				388 **xvmaddadp** VSX XX				392 **xxsldwi** VSX XX				396 **xvcmpeqdp** VSX XX			
01101	416 **xvdivdp** VSX XX				420 **xvmaddmdp** VSX XX				424 **xxpermdi** VSX XX				428 **xvcmpgtdp** VSX XX			
01110	448 **xvdivdp** VSX XX				452 **xvmsubadp** VSX XX								460 **xvcmpgedp** VSX XX			
01111	480 **xvdivdp** VSX XX				484 **xvmsubmdp** VSX XX											
10000					516 **xsnmaddasp** VSX XX				520 **xxland** VSX XX							
10001					548 **xsnmaddmsp** VSX XX				552 **xxlandc** VSX XX							
10010					580 **xsnmsubasp** VSX XX				584 **xxlor** VSX XX							
10011					612 **xsnmsubmsp** VSX XX				616 **xxlxor** VSX XX							
10100	640 **xsmaxdp** VSX XX				644 **xsnmaddadp** VSX XX				648 **xxlnor** VSX XX							
10101	672 **xsnibdp** VSX XX				676 **xsnmaddmdp** VSX XX				680 **xxlorc** VSX XX							
10110	704 **xscpsgndp** VSX XX				708 **xsnmsubadp** VSX XX				712 **xxlnand** VSX XX							
10111					740 **xsnmsubmdp** VSX XX				744 **xxleqv** VSX XX							
11000	768 **xvmaxsp** VSX XX				772 **xvnmaddasp** VSX XX								780 **xvcmpeqsp.** VSX XX			
11001	800 **xvminsp** VSX XX				804 **xvnmaddmsp** VSX XX								812 **xvcmpgtsp.** VSX XX			
11010	832 **xvcpsgnsp** VSX XX				836 **xvnmsubasp** VSX XX								844 **xvcmpgesp.** VSX XX			
11011					868 **xvnmsubmsp** VSX XX											
11100	896 **xvmaxdp** VSX XX				900 **xvnmaddadp** VSX XX								908 **xvcmpeqdp.** VSX XX			
11101	928 **xvmindp** VSX XX				932 **xvnmaddmdp** VSX XX								940 **xvcmpgtdp.** VSX XX			
11110	960 **xvcpsgndp** VSX XX				964 **xvnmsubadp** VSX XX								972 **xvcmpgedp.** VSX XX			
11111					996 **xvnmsubmdp** VSX XX											

Table 17. (Right) Extended opcodes for primary opcode 60 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000					20 **xsrsqrtesp** VSX XX		22 **xssqrtsp** VSX XX		VSX				24 **xxsel**			XA
00001					52 **xsresp** VSX XX											
00010																
00011																
00100	144 **xscvdpuxws** VSX XX		146 **xsrdpi** VSX XX		148 **xsrsqrtedp** VSX XX		150 **xssqrtdp*** VSX XX									
00101	176 **xscvdpsxws** VSX XX		178 **xsrdpiz** VSX XX		180 **xsredp** VSX XX											
00110			210 **xsrdpip** VSX XX		212 **xstsqrtdp** VSX XX		214 **xsrdpic*** VSX XX									
00111			242 **xsrdpim** VSX XX		244 **xstdivdp** VSX XX											
01000	272 **xvcvspuxws** VSX XX		274 **xvrspi** VSX XX				278 **xvsqrtsp*** VSX XX									
01001	304 **xvcvspsxws** VSX XX		306 **xvrspiz** VSX XX													
01010	336 **xvcvuxwsp** VSX XX		338 **xvrspip** VSX XX				342 **xvrspic*** VSX XX									
01011	368 **xvcvsxwsp** VSX XX		370 **xvrspim** VSX XX		372 **xvtdivsp** VSX XX											
01100	400 **xvcvdpuxws** VSX XX		402 **xvrdpi** VSX XX		404 **xvrsqrtedp** VSX XX		406 **xvsqrtdp*** VSX XX									
01101	432 **xvcvdpsxws** VSX XX		434 **xvrdpiz** VSX XX		436 **xvredp** VSX XX											
01110	464 **xvcvuxwdp** VSX XX		466 **xvrdpip** VSX XX		468 **xvtsqrtdp** VSX XX		470 **xvrdpic*** VSX XX									
01111	496 **xvcvsxwdp** VSX XX		498 **xvrdpim** VSX XX		500 **xvtdivdp** VSX XX											
10000			530 **xscvdpsp** VSX XX				534 **xscvdpspn** VSX XX									
10001			562 **xsrsp** VSX XX													
10010	592 **xscvuxdsp** VSX XX															
10011	624 **xscvsxdsp** VSX XX															
10100	656 **xscvdpuxds** VSX XX		658 **xscvspdp** VSX XX				662 **xscvspdpn** VSX XX									
10101	688 **xscvdpsxds** VSX XX		690 **xsabsdp** VSX XX													
10110	720 **xscvuxddp** VSX XX		722 **xsnabsdp** VSX XX													
10111	752 **xscvsxddp** VSX XX		754 **xsnegdp** VSX XX													
11000	784 **xvcvspuxds** VSX XX		786 **xvcvdpsp** VSX XX													
11001	816 **xvcvspsxds** VSX XX		818 **xvabssp** VSX XX													
11010	848 **xvcvuxdsp** VSX XX		850 **xvnabssp** VSX XX													
11011	880 **xvcvsxdsp** VSX XX		882 **xvnegsp** VSX XX													
11100	912 **xvcvdpuxds** VSX XX		914 **xvcvspdp** VSX XX													
11101	944 **xvcvdpsxds** VSX XX		946 **xvabsdp** VSX XX													
11110	976 **xvcvuxddp** VSX XX		978 **xvnabsdp** VSX XX													
11111	1008 **xvcvsxddp** VSX XX		1010 **xvnegdp** VSX XX													

Table 18:(Left) Extended opcodes for primary opcode 63 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111
00000	0 ***fcmpu*** FP X		2 ***daddq*** DFP X	3 ***dquaq*** DFP Z					8 ***fcpsgn*** FP X				12 ***frsp*** FP X		14 ***fctiw*** FP X	15 ***fctiwz*** FP X
00001	32 ***fcmpo*** FP X		34 ***dmulq*** DFP X	35 ***drrndq*** DFP Z23			38 ***mtfsb1*** FP X		40 ***fneg*** FP X							
00010	64 ***mcrfs*** FP X		66 ***dscliq*** DFP Z22	67 ***dquaiq*** DFP Z			70 ***mtfsb0*** FP X		72 ***fmr*** FP X							
00011			98 ***dscriq*** DFP Z	99 ***drintxq*** DFP Z23												
00100	128 ***ftdiv*** FP X		130 ***dcmpoq*** DFP X				134 ***mtfsfi*** FP X		136 ***fnabs*** FP X						142 ***fctiwu*** FP X	143 ***fctiwuz*** FP X
00101	160 ***ftsqrt*** FP X		162 ***dtstexq*** DFP X													
00110			194 ***dtstdcq*** DFP Z22													
00111			226 ***dtstdgq*** DFP Z22	227 ***drintnq*** DFP Z23												
01000			258 ***dctqpq*** DFP X	259 ***dqua'*** DFP Z					264 ***fabs*** FP X							
01001			290 ***dctfixq*** DFP X	291 ***drrnd'*** DFP Z												
01010			322 ***ddedpdq*** DFP X	323 ***dquai'*** DFP Z												
01011			354 ***dxexq*** DFP X	355 ***drintx'*** DFP Z23												
01100									392 ***frin*** FP X							
01101									424 ***friz*** FP X							
01110									456 ***frip*** FP X							
01111				483 ***drintn'*** DFP Z23					488 ***frim*** FP X							
10000			514 ***dsubq*** DFP X	515 ***dqua'*** DFP Z												
10001			546 ***ddivq*** DFP X	547 ***drrnd'*** DFP Z												
10010			578 ***dscli'*** DFP Z22	579 ***dquai'*** DFP Z				583 ***mffs*** FP X								
10011			610 ***dscri'*** DFP Z22	611 ***drintx'*** DFP Z23												
10100			642 ***dcmpuq*** DFP X													
10101			674 ***dtstsfq*** DFP X													
10110			706 ***dtstdc'*** DFP Z23					711 ***mtfsf*** FP XFL								
10111			738 ***dtstdg'*** DFP Z23	739 ***drintn'*** DFP Z23												
11000			770 ***drdpq*** DFP X	771 ***dqua'*** DFP Z												
11001			802 ***dcffixq*** DFP X	803 ***drrnd'*** DFP Z											814 ***fctid*** FP X	815 ***fctidz*** FP X
11010			834 ***denbcdq*** DFP X	835 ***dquai'*** DFP Z			838 ***fmrgew*** VSX X								846 ***fcfid*** FP X	
11011			866 ***diexq*** DFP X	867 ***drintx'*** DFP Z23												
11100																
11101															942 ***fctidu*** FP X	943 ***fctiduz*** FP X
11110							995 ***fmrgew*** VSX X								974 ***fcfidu*** FP X	
11111				995 ***drintn'*** DFP Z23												

Table 18. (Right) Extended opcodes for primary opcode 63 (instruction bits 21:30)

	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000			18 ***fdiv*** FP A		20 ***fsub*** FP A	21 ***fadd*** FP A	22 ***fsqrt*** FP A	23 ***fsel*** FP A	24 ***fre*** FP A	25 ***fmul*** FP A	26 ***frsqrte*** FP A		28 ***fmsub*** FP A	29 ***fmadd*** FP A	30 ***fnmsub*** FP A	31 ***fnmadd*** FP A
00001																
00010																
00011																
00100																
00101																
00110																
00111																
01000																
01001																
01010																
01011																
01100																
01101																
01110																
01111																
10000																
10001																
10010																
10011																
10100																
10101																
10110																
10111																
11000																
11001																
11010																
11011																
11100																
11101																
11110																
11111			***fdiv***		***fsub***	***fadd***	***fsqrt***	***fsel***	***fre***	***fmul***	***frsqrte***		***fmsub***	***fmadd***	***fnmsub***	***fnmadd***

Appendix G. Power ISA Instruction Set Sorted by Category

This appendix lists all the instructions in the Power ISA, grouped by category, and in order by mnemonic within category.

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C0001F8			90	64	bpermd	Bit Permute Doubleword
X	31	7C000074	SR		89	64	cntlzd[.]	Count Leading Zeros Doubleword
XO	31	7C0003D2	SR		76	64	divd[.]	Divide Doubleword
XO	31	7C000352	SR		77	64	divde[.]	Divide Doubleword Extended
XO	31	7C000752	SR		77	64	divdeo[.]	Divide Doubleword Extended & record OV
XO	31	7C000312	SR		77	64	divdeu[.]	Divide Doubleword Extended Unsigned
XO	31	7C000712	SR		77	64	divdeuo[.]	Divide Doubleword Extended Unsigned & record OV
XO	31	7C0007D2	SR		76	64	divdo[.]	Divide Doubleword & record OV
XO	31	7C000392	SR		76	64	divdu[.]	Divide Doubleword Unsigned
XO	31	7C000792	SR		76	64	divduo[.]	Divide Doubleword Unsigned & record OV
X	31	7C0007B4	SR		89	64	extsw[.]	Extend Sign Word
DS	58	E8000000			53	64	ld	Load Doubleword
X	31	7C0000A8			782	64	ldarx	Load Doubleword And Reserve Indexed
X	31	7C000428			60	64	ldbrx	Load Doubleword Byte-Reverse Indexed
DS	58	E8000001			53	64	ldu	Load Doubleword with Update
X	31	7C00006A			53	64	ldux	Load Doubleword with Update Indexed
X	31	7C00002A			53	64	ldx	Load Doubleword Indexed
DS	58	E8000002			52	64	lwa	Load Word Algebraic
X	31	7C0002EA			52	64	lwaux	Load Word Algebraic with Update Indexed
X	31	7C0002AA			52	64	lwax	Load Word Algebraic Indexed
XO	31	7C000092	SR		75	64	mulhd[.]	Multiply High Doubleword
XO	31	7C000012	SR		64	64	mulhdu[.]	Multiply High Doubleword Unsigned
XO	31	7C0001D2	SR		64	64	mulld[.]	Multiply Low Doubleword
XO	31	7C0005D2	SR		64	64	mulldo[.]	Multiply Low Doubleword & record OV
X	31	7C0003F4			89	64	popcntd	Population Count Doubleword
X	31	7C000174			88	64	prtyd	Parity Doubleword
MDS	30	78000010	SR		95	64	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	78000012	SR		96	64	rldcr[.]	Rotate Left Doubleword then Clear Right
MD	30	78000008	SR		95	64	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	78000000	SR		94	64	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	78000004	SR		94	64	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	7800000C	SR		96	64	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
X	31	7C000036	SR		99	64	sld[.]	Shift Left Doubleword
X	31	7C000634	SR		100	64	srad[.]	Shift Right Algebraic Doubleword
XS	31	7C000674	SR		100	64	sradi[.]	Shift Right Algebraic Doubleword Immediate

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C000436	SR		99	64	srd[.]	Shift Right Doubleword
DS	62	F8000000			57	64	std	Store Doubleword
X	31	7C000528			60	64	stdbrx	Store Doubleword Byte-Reverse Indexed
X	31	7C0001AD			782	64	stdcx.	Store Doubleword Conditional Indexed & record CR0
DS	62	F8000001			57	64	stdu	Store Doubleword with Update
X	31	7C00016A			57	64	stdux	Store Doubleword with Update Indexed
X	31	7C00012A			57	64	stdx	Store Doubleword Indexed
X	31	7C000088			81	64	td	Trap Doubleword
D	2	08000000			81	64	tdi	Trap Doubleword Immediate
XO	31	7C000214	SR		67	B	add[.]	Add
XO	31	7C000014	SR		68	B	addc[.]	Add Carrying
XO	31	7C000414	SR		68	B	addco[.]	Add Carrying & record OV
XO	31	7C000114	SR		69	B	adde[.]	Add Extended
XO	31	7C000514	SR		69	B	addeo[.]	Add Extended & record OV & record OV
D	14	38000000			66	B	addi	Add Immediate
D	12	30000000	SR		67	B	addic	Add Immediate Carrying
D	13	34000000	SR		67	B	addic.	Add Immediate Carrying & record CR0
D	15	3C000000			66	B	addis	Add Immediate Shifted
XO	31	7C0001D4	SR		69	B	addme[.]	Add to Minus One Extended
XO	31	7C0005D4	SR		69	B	addmeo[.]	Add to Minus One Extended & record OV
XO	31	7C000614	SR		67	B	addo[.]	Add & record OV
XO	31	7C000194	SR		70	B	addze[.]	Add to Zero Extended
XO	31	7C000594	SR		70	B	addzeo[.]	Add to Zero Extended & record OV
X	31	7C000038	SR		84	B	and[.]	AND
X	31	7C000078	SR		85	B	andc[.]	AND with Complement
D	28	70000000	SR		82	B	andi.	AND Immediate & record CR0
D	29	74000000	SR		82	B	andis.	AND Immediate Shifted & record CR0
I	18	48000000			38	B	b[l][a]	Branch
B	16	40000000	CT		38	B	bc[l][a]	Branch Conditional
XL	19	4C000420	CT		39	B	bcctr[l]	Branch Conditional to Count Register
XL	19	4C000020	CT		39	B	bclr[l]	Branch Conditional to Link Register
X	19	4C000460			40	B	bctar[l]	Branch Conditional to Branch Target Address Register
X	31	7C000000			78	B	cmp	Compare
X	31	7C0003F8			86	B	cmpb	Compare Byte
D	11	2C000000			78	B	cmpi	Compare Immediate
X	31	7C000040			79	B	cmpl	Compare Logical
D	10	28000000			79	B	cmpli	Compare Logical Immediate
X	31	7C000034	SR		85	B	cntlzw[.]	Count Leading Zeros Word
XL	19	4C000202			41	B	crand	Condition Register AND
XL	19	4C000102			42	B	crandc	Condition Register AND with Complement
XL	19	4C000242			42	B	creqv	Condition Register Equivalent
XL	19	4C0001C2			41	B	crnand	Condition Register NAND
XL	19	4C000042			42	B	crnor	Condition Register NOR
XL	19	4C000382			41	B	cror	Condition Register OR
XL	19	4C000342			42	B	crorc	Condition Register OR with Complement
XL	19	4C000182			41	B	crxor	Condition Register XOR
X	31	7C0000AC			773	B	dcbf	Data Cache Block Flush
X	31	7C00006C			773	B	dcbst	Data Cache Block Store
X	31	7C00022C			770	B	dcbt	Data Cache Block Touch
X	31	7C0001EC			771	B	dcbtst	Data Cache Block Touch for Store
X	31	7C0007EC			773	B	dcbz	Data Cache Block Zero
XO	31	7C0003D6	SR		72	B	divw[.]	Divide Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	31	7C000356	SR		73	B	divwe[.]	Divide Word Extended
XO	31	7C000756	SR		73	B	divweo[.]	Divide Word Extended & record OV
XO	31	7C000316	SR		73	B	divweu[.]	Divide Word Extended Unsigned
XO	31	7C000716	SR		73	B	divweuo[.]	Divide Word Extended Unsigned & record OV
XO	31	7C0007D6	SR		72	B	divwo[.]	Divide Word & record OV
XO	31	7C000396	SR		72	B	divwu[.]	Divide Word Unsigned
XO	31	7C000796	SR		72	B	divwuo[.]	Divide Word Unsigned & record OV
X	31	7C000238	SR		85	B	eqv[.]	Equivalent
X	31	7C000774	SR		85	B	extsb[.]	Extend Sign Byte
X	31	7C000734	SR		85	B	extsh[.]	Extend Sign Halfword
X	31	7C0007AC			762	B	icbi	Instruction Cache Block Invalidate
A	31	7C00001E			81	B	isel	Integer Select
XL	19	4C00012C			776	B	isync	Instruction Synchronize
X	31	7C000068			777	B	lbarx	Load Byte And Reserve Indexed
D	34	88000000			48	B	lbz	Load Byte and Zero
D	35	8C000000			48	B	lbzu	Load Byte and Zero with Update
X	31	7C0000EE			48	B	lbzux	Load Byte and Zero with Update Indexed
X	31	7C0000AE			49	B	lbzx	Load Byte and Zero Indexed
D	42	A8000000			50	B	lha	Load Halfword Algebraic
X	31	7C0000E8			778	B	lharx	Load Halfword And Reserve Indexed Xform
D	43	AC000000			50	B	lhau	Load Halfword Algebraic with Update
X	31	7C0002EE			50	B	lhaux	Load Halfword Algebraic with Update Indexed
X	31	7C0002AE			50	B	lhax	Load Halfword Algebraic Indexed
X	31	7C00062C			59	B	lhbrx	Load Halfword Byte-Reverse Indexed
D	40	A0000000			49	B	lhz	Load Halfword and Zero
D	41	A4000000			49	B	lhzu	Load Halfword and Zero with Update
X	31	7C00026E			49	B	lhzux	Load Halfword and Zero with Update Indexed
X	31	7C00022E			49	B	lhzx	Load Halfword and Zero Indexed
D	46	B8000000			61	B	lmw	Load Multiple Word
X	31	7C000028			777	B	lwarx	Load Word and Reserve Indexed
X	31	7C00042C			59	B	lwbrx	Load Word Byte-Reverse Indexed
D	32	80000000			51	B	lwz	Load Word and Zero
D	33	84000000			51	B	lwzu	Load Word and Zero with Update
X	31	7C00006E			51	B	lwzux	Load Word and Zero with Update Indexed
X	31	7C00002E			51	B	lwzx	Load Word and Zero Indexed
XL	19	4C000000			42	B	mcrf	Move Condition Register Field
XFX	31	7C000026			108	B	mfcr	Move From Condition Register
XFX	31	7C100026			112	B	mfocrf	Move From One Condition Register Field
XFX	31	7C0002A6		O	106 813 885 1054	B	mfspr	Move From Special Purpose Register
XFX	31	7C000120			107	B	mtcrf	Move To Condition Register Fields
XFX	31	7C100120			112	B	mtocrf	Move To One Condition Register Field
XFX	31	7C0003A6		O	104 884 1053	B	mtspr	Move To Special Purpose Register
XO	31	7C000096	SR		71	B	mulhw[.]	Multiply High Word
XO	31	7C000016	SR		71	B	mulhwu[.]	Multiply High Word Unsigned
D	7	1C000000			71	B	mulli	Multiply Low Immediate
XO	31	7C0001D6	SR		71	B	mullw[.]	Multiply Low Word
XO	31	7C0005D6	SR		71	B	mullwo[.]	Multiply Low Word & record OV
X	31	7C0003B8	SR		84	B	nand[.]	NAND
XO	31	7C0000D0	SR		70	B	neg[.]	Negate
XO	31	7C0004D0	SR		70	B	nego[.]	Negate & record OV

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C0000F8	SR		85	B	nor[.]	NOR
X	31	7C000378	SR		84	B	or[.]	OR
X	31	7C000338	SR		85	B	orc[.]	OR with Complement
D	24	60000000			82	B	ori	OR Immediate
D	25	64000000			83	B	oris	OR Immediate Shifted
X	31	7C0000F4			87	B	popcntb	Population Count Byte-wise
X	31	7C0002F4			87	B	popcntw	Population Count Words
X	31	7C000134			88	B	prtyw	Parity Word
M	20	50000000	SR		93	B	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21	54000000	SR		91	B	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23	5C000000	SR		92	B	rlwnm[.]	Rotate Left Word then AND with Mask
SC	17	44000002			43 863 1040	B	sc	System Call
X	31	7C000030	SR		97	B	slw[.]	Shift Left Word
X	31	7C000630	SR		98	B	sraw[.]	Shift Right Algebraic Word
X	31	7C000670	SR		98	B	srawi[.]	Shift Right Algebraic Word Immediate
X	31	7C000430	SR		97	B	srw[.]	Shift Right Word
D	38	98000000			54	B	stb	Store Byte
X	31	7C00056D			779	B	stbcx.	Store Byte Conditional Indexed
D	39	9C000000			54	B	stbu	Store Byte with Update
X	31	7C0001EE			54	B	stbux	Store Byte with Update Indexed
X	31	7C0001AE			54	B	stbx	Store Byte Indexed
D	44	B0000000			55	B	sth	Store Halfword
X	31	7C00072C			59	B	sthbrx	Store Halfword Byte-Reverse Indexed
X	31	7C0005AD			780	B	sthcx.	Store Halfword Conditional Indexed Xform
D	45	B4000000			55	B	sthu	Store Halfword with Update
X	31	7C00036E			55	B	sthux	Store Halfword with Update Indexed
X	31	7C00032E			55	B	sthx	Store Halfword Indexed
D	47	BC000000			61	B	stmw	Store Multiple Word
D	36	90000000			56	B	stw	Store Word
X	31	7C00052C			59	B	stwbrx	Store Word Byte-Reverse Indexed
X	31	7C00012D			781	B	stwcx.	Store Word Conditional Indexed & record CR0
D	37	94000000			56	B	stwu	Store Word with Update
X	31	7C00016E			56	B	stwux	Store Word with Update Indexed
X	31	7C00012E			56	B	stwx	Store Word Indexed
XO	31	7C000050	SR		67	B	subf[.]	Subtract From
XO	31	7C000010	SR		68	B	subfc[.]	Subtract From Carrying
XO	31	7C000410	SR		68	B	subfco[.]	Subtract From Carrying & record OV
XO	31	7C000110	SR		69	B	subfe[.]	Subtract From Extended
XO	31	7C000510	SR		69	B	subfeo[.]	Subtract From Extended & record OV
D	8	20000000	SR		68	B	subfic	Subtract From Immediate Carrying
XO	31	7C0001D0	SR		69	B	subfme[.]	Subtract From Minus One Extended
XO	31	7C0005D0	SR		69	B	subfmeo[.]	Subtract From Minus One Extended & record OV
XO	31	7C000450	SR		67	B	subfo[.]	Subtract From & record OV
XO	31	7C000190	SR		70	B	subfze[.]	Subtract From Zero Extended
XO	31	7C000590	SR		70	B	subfzeo[.]	Subtract From Zero Extended & record OV
X	31	7C0004AC			786	B	sync	Synchronize
X	31	7C000008			80	B	tw	Trap Word
D	3	0C000000			80	B	twi	Trap Word Immediate
X	26	68000000				B	xnop	Executed No Operation
X	31	7C000278	SR		84	B	xor[.]	XOR

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
D	26	68000000			83	B	xori	XOR Immediate
D	27	6C000000			83	B	xoris	XOR Immediate Shifted
XO	31	7C000094		H	101	BCDA	addg6s	Add and Generate Sixes
X	31	7C000274		H	101	BCDA	cbcdtd	Convert Binary Coded Decimal To Declets
X	31	7C000234		H	101	BCDA	cdtbcd	Convert Declets To Binary Coded Decimal
X	59	EC000004			289	DFP	dadd[.]	Decimal Floating Add
X	63	FC000004			289	DFP	daddq[.]	Decimal Floating Add Quad
X	59	EC000644			311	DFP	dcffix[.]	Decimal Floating Convert From Fixed
X	63	FC000644			311	DFP	dcffixq[.]	Decimal Floating Convert From Fixed Quad
X	59	EC000104			295	DFP	dcmpo	Decimal Floating Compare Ordered
X	63	FC000104			295	DFP	dcmpoq	Decimal Floating Compare Ordered Quad
X	59	EC000504			294	DFP	dcmpu	Decimal Floating Compare Unordered
X	63	FC000504			295	DFP	dcmpuq	Decimal Floating Compare Unordered Quad
X	59	EC000204			309	DFP	dctdp[.]	Decimal Floating Convert To DFP Long
X	59	EC000244			311	DFP	dctfix[.]	Decimal Floating Convert To Fixed
X	63	FC000244			311	DFP	dctfixq[.]	Decimal Floating Convert To Fixed Quad
X	63	FC000204			309	DFP	dctqpq[.]	Decimal Floating Convert To DFP Extended
X	59	EC000284			313	DFP	ddedpd[.]	Decimal Floating Decode DPD To BCD
X	63	FC000284			313	DFP	ddedpdq[.]	Decimal Floating Decode DPD To BCD Quad
X	59	EC000444			292	DFP	ddiv[.]	Decimal Floating Divide
X	63	FC000444			292	DFP	ddivq[.]	Decimal Floating Divide Quad
X	59	EC000684			313	DFP	denbcd[.]	Decimal Floating Encode BCD To DPD
X	63	FC000684			313	DFP	denbcdq[.]	Decimal Floating Encode BCD To DPD Quad
X	59	EC0006C4			314	DFP	diex[.]	Decimal Floating Insert Exponent
X	63	FC0006C4			314	DFP	diexq[.]	Decimal Floating Insert Exponent Quad
X	59	EC000044			291	DFP	dmul[.]	Decimal Floating Multiply
X	63	FC000044			291	DFP	dmulq[.]	Decimal Floating Multiply Quad
Z23	59	EC000006			300	DFP	dqua[.]	Decimal Quantize
Z23	59	EC000086			299	DFP	dquai[.]	Decimal Quantize Immediate
Z23	63	FC000086			299	DFP	dquaiq[.]	Decimal Quantize Immediate Quad
Z23	63	FC000006			300	DFP	dquaq[.]	Decimal Quantize Quad
X	63	FC000604			310	DFP	drdpq[.]	Decimal Floating Round To DFP Long
Z23	59	EC0001C6			307	DFP	drintn[.]	Decimal Floating Round To FP Integer Without Inexact
Z23	63	FC0001C6			307	DFP	drintnq[.]	Decimal Floating Round To FP Integer Without Inexact Quad
Z23	59	EC0000C6			305	DFP	drintx[.]	Decimal Floating Round To FP Integer With Inexact
Z23	63	FC0000C6			305	DFP	drintxq[.]	Decimal Floating Round To FP Integer With Inexact Quad
Z23	59	EC000046			302	DFP	drrnd[.]	Decimal Floating Reround
Z23	63	FC000046			302	DFP	drrndq[.]	Decimal Floating Reround Quad
X	59	EC000604			310	DFP	drsp[.]	Decimal Floating Round To DFP Short
Z22	59	EC000084			316	DFP	dscli[.]	Decimal Floating Shift Coefficient Left Immediate
Z22	63	FC000084			316	DFP	dscliq[.]	Decimal Floating Shift Coefficient Left Immediate Quad
Z22	59	EC0000C4			316	DFP	dscri[.]	Decimal Floating Shift Coefficient Right Immediate
Z22	63	FC0000C4			316	DFP	dscriq[.]	Decimal Floating Shift Coefficient Right Immediate Quad
X	59	EC000404			289	DFP	dsub[.]	Decimal Floating Subtract
X	63	FC000404			289	DFP	dsubq[.]	Decimal Floating Subtract Quad
Z22	59	EC000184			296	DFP	dtstdc	Decimal Floating Test Data Class
Z22	63	FC000184			296	DFP	dtstdcq	Decimal Floating Test Data Class Quad

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
Z22	59	EC0001C4			296	DFP	dtstdg	Decimal Floating Test Data Group
Z22	63	FC0001C4			296	DFP	dtstdgq	Decimal Floating Test Data Group Quad
X	59	EC000144			297	DFP	dtstex	Decimal Floating Test Exponent
X	63	FC000144			297	DFP	dtstexq	Decimal Floating Test Exponent Quad
X	59	EC000544			298	DFP	dtstsf	Decimal Floating Test Significance
X	63	FC000544			298	DFP	dtstsfq	Decimal Floating Test Significance Quad
X	59	EC0002C4			314	DFP	dxex[.]	Decimal Floating Extract Exponent
X	63	FC0002C4			314	DFP	dxexq[.]	Decimal Floating Extract Exponent Quad
X	31	7C0003C6			824	DS	dsn	Decorated Storage Notify
X	31	7C000406			822	DS	lbdx	Load Byte with Decoration Indexed
X	31	7C0004C6			822	DS	lddx	Load Doubleword with Decoration Indexed
X	31	7C000646			822	DS	lfddx	Load Floating Doubleword with Decoration Indexed
X	31	7C000446			822	DS	lhdx	Load Halfword with Decoration Indexed
X	31	7C000486			822	DS	lwdx	Load Word with Decoration Indexed
X	31	7C000506			823	DS	stbdx	Store Byte with Decoration Indexed
X	31	7C0005C6			823	DS	stddx	Store Doubleword with Decoration Indexed
X	31	7C000746			823	DS	stfddx	Store Floating Doubleword with Decoration Indexed
X	31	7C000546			823	DS	sthdx	Store Halfword with Decoration Indexed
X	31	7C000586			823	DS	stwdx	Store Word with Decoration Indexed
X	31	7C0005EC			770	E	dcba	Data Cache Block Allocate
X	31	7C0003AC		P	1118	E	dcbi	Data Cache Block Invalidate
XFX	19	4C00018C			1228	E	dnh	Debugger Notify Halt
X	31	7C00002C			762	E	icbt	Instruction Cache Block Touch
X	31	7C0006AC			790	E	mbar	Memory Barrier
X	31	7C000400			113	E	mcrxr	Move to Condition Register from XER
XL	19	4C000066		P	1041	E	rfci	Return From Critical Interrupt
XL	19	4C000064		P	1041	E	rfi	Return From Interrupt
XL	19	4C00004C		P	1042	E	rfmci	Return From Machine Check Interrupt
X	31	7C000024		P	1134	E	tlbilx	TLB Invalidate Local Indexed
X	31	7C000624		P	1132	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	31	7C000764		P	1139	E	tlbre	TLB Read Entry
X	31	7C000724		P	1136	E	tlbsx	TLB Search Indexed
X	31	7C0007A4		P	1141	E	tlbwe	TLB Write Entry
X	31	7C000106		P	1056	E	wrtee	Write External Enable
X	31	7C000146		P	1057	E	wrteei	Write External Enable Immediate
X	31	7C00028C		P	1242	E.CD	dcread	Data Cache Read
X	31	7C0003CC		P	1242	E.CD	dcread	Data Cache Read
X	31	7C0007CC		P	1243	E.CD	icread	Instruction Cache Read
X	31	7C00038C		P	1239	E.CI	dci	Data Cache Invalidate
X	31	7C00078C		P	1239	E.CI	ici	Instruction Cache Invalidate
XFX	31	7C000286		P	1055	E.DC	mfdcr	Move From Device Control Register
X	31	7C000246			113	E.DC	mfdcrux	Move From Device Control Register User Mode Indexed
X	31	7C000206		P	1055	E.DC	mfdcrx	Move From Device Control Register Indexed
XFX	31	7C000386		P	1054	E.DC	mtdcr	Move To Device Control Register
X	31	7C000346			113	E.DC	mtdcrux	Move To Device Control Register User Mode Indexed
X	31	7C000306		P	1054	E.DC	mtdcrx	Move To Device Control Register Indexed
X	19	4C00004E		P	1042	E.ED	rfdi	Return From Debug Interrupt
XL	31	7C00021C			1043	E.HV	ehpriv	Embedded Hypervisor Privilege
XL	19	4C0000CC		P	1043	E.HV	rfgi	Return From Guest Interrupt
X	31	7C0000FE		P	1064	E.PD	dcbfep	Data Cache Block Flush by External PID

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C00007E		P	1063	E.PD	dcbstep	Data Cache Block Store by External PID
X	31	7C00027E		P	1063	E.PD	dcbtep	Data Cache Block Touch by External PID
X	31	7C0001FE		P	1066	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID
X	31	7C0007FE		P	1067	E.PD	dcbzep	Data Cache Block Zero by External PID
EVX	31	7C00063E		P	1069	E.PD	evlddepx	Vector Load Double Word into Double Word by External PID Indexed
EVX	31	7C00073E		P	1069	E.PD	evstddepx	Vector Store Double of Double by External PID Indexed
X	31	7C0007BE		P	1067	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	31	7C0000BE		P	1059	E.PD	lbepx	Load Byte and Zero by External PID Indexed
X	31	7C0004BE		P	1068	E.PD	lfdepx	Load Floating-Point Double by External PID Indexed
X	31	7C00023E		P	1059	E.PD	lhepx	Load Halfword and Zero by External PID Indexed
X	31	7C00024E		P	1070	E.PD	lvepx	Load Vector by External PID Indexed
X	31	7C00020E		P	1070	E.PD	lvepxl	Load Vector by External PID Indexed Last
X	31	7C00003E		P	1060	E.PD	lwepx	Load Word and Zero by External PID Indexed
X	31	7C0001BE		P	1061	E.PD	stbepx	Store Byte by External PID Indexed
X	31	7C0005BE		P	1068	E.PD	stfdepx	Store Floating-Point Double by External PID Indexed
X	31	7C00033E		P	1061	E.PD	sthepx	Store Halfword by External PID Indexed
X	31	7C00064E		P	1071	E.PD	stvepx	Store Vector by External PID Indexed
X	31	7C00060E		P	1071	E.PD	stvepxl	Store Vector by External PID Indexed Last
X	31	7C00013E		P	1062	E.PD	stwepx	Store Word by External PID Indexed
X	31	7C00003A		P	1060	E.PD;64	ldepx	Load Doubleword by External PID Indexed
X	31	7C00013A		P	1062	E.PD;64	stdepx	Store Doubleword by External PID Indexed
XFX	31	7C00029C		O	1257	E.PM	mfpmr	Move from Performance Monitor Register
XFX	31	7C00039C		O	1257	E.PM	mtpmr	Move To Performance Monitor Register
X	31	7C0006A5		P	1138	E.TWC	tlbsrx.	TLB Search and Reserve Indexed
X	31	7C00026C			826	EC	eciwx	External Control In Word Indexed
X	31	7C00036C			826	EC	ecowx	External Control Out Word Indexed
X	31	7C00030C		M	1123	ECL	dcblc	Data Cache Block Lock Clear
X	31	7C00034D		M	1121	ECL	dcblq.	Data Cache Block Lock Query
X	31	7C00014C		M	1122	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	31	7C00010C		M	1122	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	31	7C0001CC		M	1124	ECL	icblc	Instruction Cache Block Lock Clear
X	31	7C00018D		M	1121	ECL	icblq.	Instruction Cache Block Lock Query
X	31	7C0003CC		M	1123	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	63	FC000040			159	FP	fcmpo	Floating Compare Ordered
X	63	FC000000			159	FP	fcmpu	Floating Compare Unordered
X	63	FC000100			148	FP	ftdiv	Floating Test for software Divide
X	63	FC000140			148	FP	ftsqrt	Floating Test for software Square Root
D	50	C8000000			135	FP	lfd	Load Floating-Point Double
D	51	CC000000			135	FP	lfdu	Load Floating-Point Double with Update
X	31	7C0004EE			135	FP	lfdux	Load Floating-Point Double with Update Indexed
X	31	7C0004AE			135	FP	lfdx	Load Floating-Point Double Indexed
X	31	7C0006AE			136	FP	lfiwax	Load Floating-Point as Integer Word Algebraic Indexed
X	31	7C0006EE			136	FP	lfiwzx	Load Floating-Point as Integer Word and Zero Indexed
D	48	C0000000			138	FP	lfs	Load Floating-Point Single
D	49	C4000000			138	FP	lfsu	Load Floating-Point Single with Update

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C00046E			138	FP	lfsux	Load Floating-Point Single with Update Indexed
X	31	7C00042E			138	FP	lfsx	Load Floating-Point Single Indexed
X	63	FC000080			161	FP	mcrfs	Move To Condition Register from FPSCR
D	54	D8000000			139	FP	stfd	Store Floating-Point Double
D	55	DC000000			139	FP	stfdu	Store Floating-Point Double with Update
X	31	7C0005EE			139	FP	stfdux	Store Floating-Point Double with Update Indexed
X	31	7C0005AE			139	FP	stfdx	Store Floating-Point Double Indexed
X	31	7C0007AE			140	FP	stfiwx	Store Floating-Point as Integer Word Indexed
D	52	D0000000			138	FP	stfs	Store Floating-Point Single
D	53	D4000000			138	FP	stfsu	Store Floating-Point Single with Update
X	31	7C00056E			138	FP	stfsux	Store Floating-Point Single with Update Indexed
X	31	7C00052E			138	FP	stfsx	Store Floating-Point Single Indexed
DS	57	E4000000			141	FP.out	lfdp	Load Floating-Point Double Pair
X	31	7C00062E			141	FP.out	lfdpx	Load Floating-Point Double Pair Indexed
DS	61	F4000000			141	FP.out	stfdp	Store Floating-Point Double Pair
X	31	7C00072E			141	FP.out	stfdpx	Store Floating-Point Double Pair Indexed
X	63	FC000210			142	FP[R]	fabs[.]	Floating Absolute Value
A	63	FC00002A			144	FP[R]	fadd[.]	Floating Add
A	59	EC00002A			144	FP[R]	fadds[.]	Floating Add Single
X	63	FC00069C			155	FP[R]	fcfid[.]	Floating Convert From Integer Doubleword
X	59	EC00069C			156	FP[R]	fcfids[.]	Floating Convert From Integer Doubleword Single
X	63	FC00079C			156	FP[R]	fcfidu[.]	Floating Convert From Integer Doubleword Unsigned
X	59	EC00079C			157	FP[R]	fcfidus[.]	Floating Convert From Integer Doubleword Unsigned Single
X	63	FC000010			142	FP[R]	fcpsgn[.]	Floating Copy Sign
X	63	FC00065C			151	FP[R]	fctid[.]	Floating Convert To Integer Doubleword
X	63	FC00075C			152	FP[R]	fctidu[.]	Floating Convert To Integer Doubleword Unsigned
X	63	FC00075E			153	FP[R]	fctiduz[.]	Floating Convert To Integer Doubleword Unsigned with round toward Zero
X	63	FC00065E			152	FP[R]	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	FC00001C			153	FP[R]	fctiw[.]	Floating Convert To Integer Word
X	63	FC00011C			154	FP[R]	fctiwu[.]	Floating Convert To Integer Word Unsigned
X	63	FC00011E			155	FP[R]	fctiwuz[.]	Floating Convert To Integer Word Unsigned with round toward Zero
X	63	FC00001E			154	FP[R]	fctiwz[.]	Floating Convert To Integer Word with round to Zero
A	63	FC000024			145	FP[R]	fdiv[.]	Floating Divide
A	59	EC000024			145	FP[R]	fdivs[.]	Floating Divide Single
A	63	FC00003A			149	FP[R]	fmadd[.]	Floating Multiply-Add
A	59	EC00003A			149	FP[R]	fmadds[.]	Floating Multiply-Add Single
X	63	FC000090			142	FP[R]	fmr[.]	Floating Move Register
A	63	FC000038			149	FP[R]	fmsub[.]	Floating Multiply-Subtract
A	59	EC000038			149	FP[R]	fmsubs[.]	Floating Multiply-Subtract Single
A	63	FC000032			145	FP[R]	fmul[.]	Floating Multiply
A	59	EC000032			145	FP[R]	fmuls[.]	Floating Multiply Single
X	63	FC000110			142	FP[R]	fnabs[.]	Floating Negative Absolute Value
X	63	FC000050			142	FP[R]	fneg[.]	Floating Negate
A	63	FC00003E			150	FP[R]	fnmadd[.]	Floating Negative Multiply-Add
A	59	EC00003E			150	FP[R]	fnmadds[.]	Floating Negative Multiply-Add Single

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
A	63	FC00003C			150	FP[R]	fnmsub[.]	Floating Negative Multiply-Subtract
A	59	EC00003C			150	FP[R]	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	59	EC000030			146	FP[R]	fres[.]	Floating Reciprocal Estimate Single
X	63	FC000018			151	FP[R]	frsp[.]	Floating Round to Single-Precision
A	63	FC000034			147	FP[R]	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	63	FC00002E			160	FP[R]	fsel[.]	Floating Select
A	63	FC00002C			146	FP[R]	fsqrt[.]	Floating Square Root
A	59	EC00002C			146	FP[R]	fsqrts[.]	Floating Square Root Single
A	63	FC000028			144	FP[R]	fsub[.]	Floating Subtract
A	59	EC000028			144	FP[R]	fsubs[.]	Floating Subtract Single
X	63	FC00048E			161	FP[R]	mffs[.]	Move From FPSCR
X	63	FC00008C			163	FP[R]	mtfsb0[.]	Move To FPSCR Bit 0
X	63	FC00004C			163	FP[R]	mtfsb1[.]	Move To FPSCR Bit 1
XFL	63	FC00058E			162	FP[R]	mtfsf[.]	Move To FPSCR Fields
X	63	FC00010C			162	FP[R]	mtfsfi[.]	Move To FPSCR Field Immediate
A	63	FC000030			146	FP[R].in	fre[.]	Floating Reciprocal Estimate
X	63	FC0003D0			158	FP[R].in	frim[.]	Floating Round To Integer Minus
X	63	FC000310			158	FP[R].in	frin[.]	Floating Round To Integer Nearest
X	63	FC000390			158	FP[R].in	frip[.]	Floating Round To Integer Plus
X	63	FC000350			158	FP[R].in	friz[.]	Floating Round To Integer toward Zero
A	59	EC000034			147	FP[R].in	frsqrtes[.]	Floating Reciprocal Square Root Estimate Single
XO	4	10000158			675	LMA	macchw[.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	10000158			675	LMA	macchwo[.]	Multiply Accumulate Cross Halfword to Word Modulo Signed & record OV
XO	4	100001D8			675	LMA	macchws[.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	100001D8			675	LMA	macchwso[.]	Multiply Accumulate Cross Halfword to Word Saturate Signed & record OV
XO	4	10000198			676	LMA	macchwsu[.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
XO	4	10000198			676	LMA	macchwsuo[.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned & record OV
XO	4	10000118			676	LMA	macchwu[.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
XO	4	10000118			676	LMA	macchwuo[.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned & record OV
XO	4	10000058			677	LMA	machhw[.]	Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	10000058			677	LMA	machhwo[.]	Multiply Accumulate High Halfword to Word Modulo Signed & record OV
XO	4	100000D8			677	LMA	machhws[.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	100000D8			677	LMA	machhwso[.]	Multiply Accumulate High Halfword to Word Saturate Signed & record OV
XO	4	10000098			678	LMA	machhwsu[.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
XO	4	10000098			678	LMA	machhwsuo[.]	Multiply Accumulate High Halfword to Word Saturate Unsigned & record OV
XO	4	10000018			678	LMA	machhwu[.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
XO	4	10000018			678	LMA	machhwuo[.]	Multiply Accumulate High Halfword to Word Modulo Unsigned & record OV
XO	4	10000358			679	LMA	maclhw[.]	Multiply Accumulate Low Halfword to Word Modulo Signed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	4	10000358			679	LMA	maclhwo[.]	Multiply Accumulate Low Halfword to Word Modulo Signed & record OV
XO	4	100003D8			679	LMA	maclhws[.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	100003D8			679	LMA	maclhwso[.]	Multiply Accumulate Low Halfword to Word Saturate Signed & record OV
XO	4	10000398			680	LMA	maclhwsu[.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
XO	4	10000398			680	LMA	maclhwsuo[.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned & record OV
XO	4	10000318			680	LMA	maclhwu[.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
XO	4	10000318			680	LMA	maclhwuo[.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned & record OV
X	4	10000150			680	LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
X	4	10000110			680	LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
X	4	10000050			681	LMA	mulhhw[.]	Multiply High Halfword to Word Signed
X	4	10000010			681	LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
X	4	10000350			681	LMA	mullhw[.]	Multiply Low Halfword to Word Signed
X	4	10000310			681	LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
XO	4	1000015C			682	LMA	nmacchw[.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	1000015C			682	LMA	nmacchwo[.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed & record OV
XO	4	100001DC			682	LMA	nmacchws[.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	100001DC			682	LMA	nmacchwso[.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed & record OV
XO	4	1000005C			683	LMA	nmachhw[.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	1000005C			683	LMA	nmachhwo[.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed & record OV
XO	4	100000DC			683	LMA	nmachhws[.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	100000DC			683	LMA	nmachhwso[.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed & record OV
XO	4	1000035C			684	LMA	nmaclhw[.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	1000035C			684	LMA	nmaclhwo[.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed & record OV
XO	4	100003DC			684	LMA	nmaclhws[.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	100003DC			684	LMA	nmaclhwso[.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed & record OV
X	31	7C00009C			673	LMV	dlmzb[.]	Determine Leftmost Zero Byte
DQ	56	E0000000		P	58	LSQ	lq	Load Quadword
X	31	7C000228			784	LSQ	lqarx	Load Quadword And Reserve Indexed
DS	62	F8000002		P	58	LSQ	stq	Store Quadword
X	31	7C00016D			785	LSQ	stqcx.	Store Quadword Conditional Indexed and record CR0
X	31	7C0004AA			63	MA	lswi	Load String Word Immediate
X	31	7C00042A			63	MA	lswx	Load String Word Indexed
X	31	7C0005AA			64	MA	stswi	Store String Word Immediate
X	31	7C00052A			64	MA	stswx	Store String Word Indexed
X	31	7C00035C			44	S	clrbhrb	Clear BHRB
XL	19	4C000324		H	867	S	doze	Doze
X	31	7C0006AC			790	S	eieio	Enforce In-order Execution of I/O

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XL	19	4C000224		H	865	S	hrfid	Return From Interrupt Doubleword Hypervisor
X	31	7C0006AA		H	876	S	lbzcix	Load Byte and Zero Caching Inhibited Indexed
X	31	7C0006EA		H	876	S	ldcix	Load Doubleword Caching Inhibited Indexed
X	31	7C00066A		H	876	S	lhzcix	Load Halfword and Zero Caching Inhibited Indexed
X	31	7C00062A		H	876	S	lwzcix	Load Word and Zero Caching Inhibited Indexed
XFX	31	7C00025C			44	S	mfbhrbe	Move From Branch History Rolling Buffer
X	31	7C0004A6	32	P	927	S	mfsr	Move From Segment Register
X	31	7C000526	32	P	927	S	mfsrin	Move From Segment Register Indirect
X	31	7C000164		P	886	S	mtmsrd	Move To Machine State Register Doubleword
X	31	7C000126			108	S	mtsle	Move To Split Little Endian
X	31	7C0001A4	32	P	926	S	mtsr	Move To Segment Register
X	31	7C0001E4	32	P	926	S	mtsrin	Move To Segment Register Indirect
XL	19	4C000364		H	867	S	nap	Nap
XL	19	4C000124			820	S	rfebb	Return from Event Based Branch
XL	19	4C000024		P	864	S	rfid	Return from Interrupt Doubleword
XL	19	4C0003E4		H	868	S	rvwinkle	Rip Van Winkle
X	31	7C0007A7	SR	P	923	S	slbfee.	SLB Find Entry ESID
X	31	7C0003E4		P	920	S	slbia	SLB Invalidate All
X	31	7C000364		P	919	S	slbie	SLB Invalidate Entry
X	31	7C000726		P	923	S	slbmfee	SLB Move From Entry ESID
X	31	7C0006A6		P	922	S	slbmfev	SLB Move From Entry VSID
X	31	7C000324		P	921	S	slbmte	SLB Move To Entry
XL	19	4C0003A4		H	868	S	sleep	Sleep
X	31	7C0007AA		H	877	S	stbcix	Store Byte Caching Inhibited Indexed
X	31	7C0007EA		H	877	S	stdcix	Store Doubleword Caching Inhibited Indexed
X	31	7C00076A		H	877	S	sthcix	Store Halfword and Zero Caching Inhibited Indexed
X	31	7C00072A		H	877	S	stwcix	Store Word and Zero Caching Inhibited Indexed
X	31	7C0002E4		H	933	S	tlbia	TLB Invalidate All
X	31	7C000264	64	H	928	S	tlbie	TLB Invalidate Entry
X	31	7C000224	64	P	931	S	tlbiel	TLB Invalidate Entry Local
XFX	31	7C0002E6			813	S.out	mftb	Move From Time Base
X	31	7C00015C		P	1007	S.PC	msgclrp	Message Clear Privileged
X	31	7C00011C		P	1007	S.PC	msgsndp	Message Send Privileged
X	31	7C0001DC		H	1006 1233	S.PC E.PC	msgclr	Message Clear
X	31	7C00019C		H	1006 1233	S.PC E.PC	msgsnd	Message Send
X	31	7C0000A6		P	888 1055	S E	mfmsr	Move From Machine State Register
X	31	7C000124		P	884 1055	S E	mtmsr	Move To Machine State Register
X	31	7C00046C		H PH	934 1141	S E	tlbsync	TLB Synchronize
EVX	4	1000020F			594	SP	brinc	Bit Reversed Increment
EVX	4	10000208			594	SP	evabs	Vector Absolute Value
EVX	4	10000202			594	SP	evaddiw	Vector Add Immediate Word
EVX	4	100004C9			594	SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	4	100004C1			595	SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100004C8			595	SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	4	100004C0			595	SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	4	10000200			595	SP	evaddw	Vector Add Word
EVX	4	10000211			596	SP	evand	Vector AND
EVX	4	10000212			596	SP	evandc	Vector AND with Complement
EVX	4	10000234			596	SP	evcmpeq	Vector Compare Equal
EVX	4	10000231			596	SP	evcmpgts	Vector Compare Greater Than Signed
EVX	4	10000230			597	SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	4	10000233			597	SP	evcmplts	Vector Compare Less Than Signed
EVX	4	10000232			597	SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	4	1000020E			598	SP	evcntlsw	Vector Count Leading Signed Bits Word
EVX	4	1000020D			598	SP	evcntlzw	Vector Count Leading Zeros Word
EVX	4	100004C6			598	SP	evdivws	Vector Divide Word Signed
EVX	4	100004C7			599	SP	evdivwu	Vector Divide Word Unsigned
EVX	4	10000219			599	SP	eveqv	Vector Equivalent
EVX	4	1000020A			599	SP	evextsb	Vector Extend Sign Byte
EVX	4	1000020B			599	SP	evextsh	Vector Extend Sign Half Word
EVX	4	10000301			600	SP	evldd	Vector Load Double Word into Double Word
EVX	4	10000300			600	SP	evlddx	Vector Load Double Word into Double Word Indexed
EVX	4	10000305			600	SP	evldh	Vector Load Double into Four Half Words
EVX	4	10000304			600	SP	evldhx	Vector Load Double into Four Half Words Indexed
EVX	4	10000303			601	SP	evldw	Vector Load Double into Two Words
EVX	4	10000302			601	SP	evldwx	Vector Load Double into Two Words Indexed
EVX	4	10000309			601	SP	evlhhesplat	Vector Load Half Word into Half Words Even and Splat
EVX	4	10000308			601	SP	evlhhesplatx	Vector Load Half Word into Half Words Even and Splat Indexed
EVX	4	1000030F			602	SP	evlhhossplat	Vector Load Half Word into Half Word Odd Signed and Splat
EVX	4	1000030E			602	SP	evlhhossplatx	Vector Load Half Word into Half Word Odd Signed and Splat Indexed
EVX	4	1000030D			602	SP	evlhhousplat	Vector Load Half Word into Half Word Odd Unsigned and Splat
EVX	4	1000030C			602	SP	evlhhousplatx	Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed
EVX	4	10000311			603	SP	evlwhe	Vector Load Word into Two Half Words Even
EVX	4	10000310			603	SP	evlwhex	Vector Load Word into Two Half Words Even Indexed
EVX	4	10000317			603	SP	evlwhos	Vector Load Word into Two Half Words Odd Signed (with sign extension)
EVX	4	10000316			603	SP	evlwhosx	Vector Load Word into Two Half Words Odd Signed Indexed (with sign extension)
EVX	4	10000315			604	SP	evlwhou	Vector Load Word into Two Half Words Odd Unsigned (zero-extended)
EVX	4	10000314			604	SP	evlwhoux	Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)
EVX	4	1000031D			604	SP	evlwhsplat	Vector Load Word into Two Half Words and Splat
EVX	4	1000031C			604	SP	evlwhsplatx	Vector Load Word into Two Half Words and Splat Indexed
EVX	4	10000319			605	SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	4	10000318			605	SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	1000022C			605	SP	evmergehi	Vector Merge High
EVX	4	1000022E			606	SP	evmergehilo	Vector Merge High/Low
EVX	4	1000022D			605	SP	evmergelo	Vector Merge Low
EVX	4	1000022F			606	SP	evmergelohi	Vector Merge Low/High
EVX	4	1000052B			606	SP	evmhegsmfaa	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	100005AB			606	SP	evmhegsmfan	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	10000529			607	SP	evmhegsmiaa	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	100005A9			607	SP	evmhegsmian	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	10000528			607	SP	evmhegumiaa	Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	100005A8			607	SP	evmhegumian	Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1000040B			608	SP	evmhesmf	Vector Multiply Half Words, Even, Signed, Modulo, Fractional
EVX	4	1000042B			608	SP	evmhesmfa	Vector Multiply Half Words, Even, Signed, Modulo, Fractional to Accumulator
EVX	4	1000050B			608	SP	evmhesmfaaw	Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1000058B			608	SP	evmhesmfanw	Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	10000409			609	SP	evmhesmi	Vector Multiply Half Words, Even, Signed, Modulo, Integer
EVX	4	10000429			609	SP	evmhesmia	Vector Multiply Half Words, Even, Signed, Modulo, Integer to Accumulator
EVX	4	10000509			609	SP	evmhesmiaaw	Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	4	10000589			609	SP	evmhesmianw	Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000403			610	SP	evmhessf	Vector Multiply Half Words, Even, Signed, Saturate, Fractional
EVX	4	10000423			610	SP	evmhessfa	Vector Multiply Half Words, Even, Signed, Saturate, Fractional to Accumulator
EVX	4	10000503			611	SP	evmhessfaaw	Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	10000583			611	SP	evmhessfanw	Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	10000501			612	SP	evmhessiaaw	Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words
EVX	4	10000581			612	SP	evmhessianw	Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	10000408			613	SP	evmheumi	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer
EVX	4	10000428			613	SP	evmheumia	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer to Accumulator
EVX	4	10000508			613	SP	evmheumiaaw	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000588			613	SP	evmheumianw	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000500			614	SP	evmheusiaaw	Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	10000580			614	SP	evmheusianw	Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	1000052F			615	SP	evmhogsmfaa	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	100005AF			615	SP	evmhogsmfan	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1000052D			615	SP	evmhogsmiaa	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer, and Accumulate
EVX	4	100005AD			615	SP	evmhogsmian	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1000052C			616	SP	evmhogumiaa	Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	100005AC			616	SP	evmhogumian	Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1000040F			616	SP	evmhosmf	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional
EVX	4	1000042F			616	SP	evmhosmfa	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional to Accumulator
EVX	4	1000050F			617	SP	evmhosmfaaw	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1000058F			617	SP	evmhosmfanw	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1000040D			617	SP	evmhosmi	Vector Multiply Half Words, Odd, Signed, Modulo, Integer
EVX	4	1000042D			617	SP	evmhosmia	Vector Multiply Half Words, Odd, Signed, Modulo, Integer to Accumulator
EVX	4	1000050D			618	SP	evmhosmiaaw	Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1000058D			617	SP	evmhosmianw	Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000407			619	SP	evmhossf	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional
EVX	4	10000427			619	SP	evmhossfa	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional to Accumulator
EVX	4	10000507			620	SP	evmhossfaaw	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	10000587			620	SP	evmhossfanw	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	10000505			621	SP	evmhossiaaw	Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	4	10000585			621	SP	evmhossianw	Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	1000040C			621	SP	evmhoumi	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer
EVX	4	1000042C			621	SP	evmhoumia	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	4	1000050C			622	SP	evmhoumiaaw	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1000058C			618	SP	evmhoumianw	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000504			622	SP	evmhousiaaw	Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	10000584			622	SP	evmhousianw	Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	100004C4			623	SP	evmra	Initialize Accumulator
EVX	4	1000044F			623	SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	4	1000046F			623	SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	4	1000044D			623	SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	4	1000046D			623	SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	4	10000447			624	SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	4	10000467			624	SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	4	1000044C			624	SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
EVX	4	1000046C			624	SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	4	10000549			625	SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words
EVX	4	100005C9			625	SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
EVX	4	10000541			625	SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words
EVX	4	100005C1			625	SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
EVX	4	10000448			626	SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
EVX	4	10000468			626	SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	4	10000548			626	SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words
EVX	4	100005C8			626	SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words
EVX	4	10000540			627	SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words
EVX	4	100005C0			627	SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	4	1000045B			627	SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	4	1000047B			627	SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
EVX	4	1000055B			628	SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100005DB			628	SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
EVX	4	10000459			628	SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	4	10000479			628	SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	4	10000559			628	SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	4	100005D9			628	SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	4	10000453			629	SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	4	10000473			629	SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	4	10000553			629	SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	4	100005D3			630	SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	4	10000458			630	SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	4	10000478			630	SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	4	10000558			631	SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	4	100005D8			631	SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1000021E			631	SP	evnand	Vector NAND
EVX	4	10000209			631	SP	evneg	Vector Negate
EVX	4	10000218			631	SP	evnor	Vector NOR
EVX	4	10000217			632	SP	evor	Vector OR
EVX	4	1000021B			632	SP	evorc	Vector OR with Complement
EVX	4	10000228			632	SP	evrlw	Vector Rotate Left Word
EVX	4	1000022A			633	SP	evrlwi	Vector Rotate Left Word Immediate
EVX	4	1000020C			633	SP	evrndw	Vector Round Word
EVS	4	10000278			633	SP	evsel	Vector Select
EVX	4	10000224			634	SP	evslw	Vector Shift Left Word
EVX	4	10000226			634	SP	evslwi	Vector Shift Left Word Immediate
EVX	4	1000022B			634	SP	evsplatfi	Vector Splat Fractional Immediate
EVX	4	10000229			634	SP	evsplati	Vector Splat Immediate
EVX	4	10000223			634	SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	4	10000222			634	SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	4	10000221			635	SP	evsrws	Vector Shift Right Word Signed
EVX	4	10000220			635	SP	evsrwu	Vector Shift Right Word Unsigned
EVX	4	10000321			635	SP	evstdd	Vector Store Double of Double
EVX	4	10000320			635	SP	evstddx	Vector Store Double of Double Indexed
EVX	4	10000325			636	SP	evstdh	Vector Store Double of Four Half Words
EVX	4	10000324			636	SP	evstdhx	Vector Store Double of Four Half Words Indexed
EVX	4	10000323			636	SP	evstdw	Vector Store Double of Two Words
EVX	4	10000322			636	SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	4	10000331			637	SP	evstwhe	Vector Store Word of Two Half Words from Even
EVX	4	10000330			637	SP	evstwhex	Vector Store Word of Two Half Words from Even Indexed
EVX	4	10000335			637	SP	evstwho	Vector Store Word of Two Half Words from Odd
EVX	4	10000334			637	SP	evstwhox	Vector Store Word of Two Half Words from Odd Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000339			637	SP	evstwwe	Vector Store Word of Word from Even
EVX	4	10000338			637	SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	4	1000033D			638	SP	evstwwo	Vector Store Word of Word from Odd
EVX	4	1000033C			638	SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	4	100004CB			638	SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	4	100004C3			638	SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	4	100004CA			639	SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	4	100004C2			639	SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	4	10000204			639	SP	evsubfw	Vector Subtract from Word
EVX	4	10000206			639	SP	evsubifw	Vector Subtract Immediate from Word
EVX	4	10000216			639	SP	evxor	Vector XOR
EVX	4	100002E4			660	SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	4	100002E0			661	SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	4	100002EF			666	SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	4	100002F3			664	SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	4	100002F1			663	SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	4	100002E3			664	SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	4	100002F2			664	SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	4	100002F0			663	SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	4	100002E2			664	SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	4	100002EE			662	SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	4	100002EC			662	SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	4	100002ED			662	SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	4	100002F7			666	SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	4	100002F5			664	SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	4	100002EB			665	SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	4	100002FA			666	SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	4	100002F6			666	SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	4	100002F4			664	SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	4	100002EA			665	SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	4	100002F8			666	SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	4	100002E9			661	SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	4	100002E8			661	SP.FD	efdmul	Floating-Point Double-Precision Multiply

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100002E5			660	SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	4	100002E6			660	SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	4	100002E1			661	SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	4	100002FE			663	SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	4	100002FC			662	SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	4	100002FD			663	SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	4	100002CF			667	SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	4	100002C4			653	SP.FS	efsabs	Floating-Point Absolute Value
EVX	4	100002C0			654	SP.FS	efsadd	Floating-Point Add
EVX	4	100002D3			658	SP.FS	efscfsf	Convert Floating-Point from Signed Fraction
EVX	4	100002D1			658	SP.FS	efscfsi	Convert Floating-Point from Signed Integer
EVX	4	100002D2			658	SP.FS	efscfuf	Convert Floating-Point from Unsigned Fraction
EVX	4	100002D0			658	SP.FS	efscfui	Convert Floating-Point from Unsigned Integer
EVX	4	100002CE			656	SP.FS	efscmpeq	Floating-Point Compare Equal
EVX	4	100002CC			655	SP.FS	efscmpgt	Floating-Point Compare Greater Than
EVX	4	100002CD			655	SP.FS	efscmplt	Floating-Point Compare Less Than
EVX	4	100002D7			659	SP.FS	efsctsf	Convert Floating-Point to Signed Fraction
EVX	4	100002D5			658	SP.FS	efsctsi	Convert Floating-Point to Signed Integer
EVX	4	100002DA			659	SP.FS	efsctsiz	Convert Floating-Point to Signed Integer with Round toward Zero
EVX	4	100002D6			659	SP.FS	efsctuf	Convert Floating-Point to Unsigned Fraction
EVX	4	100002D4			658	SP.FS	efsctui	Convert Floating-Point to Unsigned Integer
EVX	4	100002D8			659	SP.FS	efsctuiz	Convert Floating-Point to Unsigned Integer with Round toward Zero
EVX	4	100002C9			654	SP.FS	efsdiv	Floating-Point Divide
EVX	4	100002C8			654	SP.FS	efsmul	Floating-Point Multiply
EVX	4	100002C5			653	SP.FS	efsnabs	Floating-Point Negative Absolute Value
EVX	4	100002C6			653	SP.FS	efsneg	Floating-Point Negate
EVX	4	100002C1			654	SP.FS	efssub	Floating-Point Subtract
EVX	4	100002DE			657	SP.FS	efststeq	Floating-Point Test Equal
EVX	4	100002DC			656	SP.FS	efststgt	Floating-Point Test Greater Than
EVX	4	100002DD			657	SP.FS	efststlt	Floating-Point Test Less Than
EVX	4	10000284			645	SP.FV	evfsabs	Vector Floating-Point Absolute Value
EVX	4	10000280			646	SP.FV	evfsadd	Vector Floating-Point Add
EVX	4	10000293			650	SP.FV	evfscfsf	Vector Convert Floating-Point from Signed Fraction
EVX	4	10000291			650	SP.FV	evfscfsi	Vector Convert Floating-Point from Signed Integer
EVX	4	10000292			650	SP.FV	evfscfuf	Vector Convert Floating-Point from Unsigned Fraction
EVX	4	10000290			650	SP.FV	evfscfui	Vector Convert Floating-Point from Unsigned Integer
EVX	4	1000028E			648	SP.FV	evfscmpeq	Vector Floating-Point Compare Equal
EVX	4	1000028C			647	SP.FV	evfscmpgt	Vector Floating-Point Compare Greater Than
EVX	4	1000028D			647	SP.FV	evfscmplt	Vector Floating-Point Compare Less Than
EVX	4	10000297			652	SP.FV	evfsctsf	Vector Convert Floating-Point to Signed Fraction
EVX	4	10000295			651	SP.FV	evfsctsi	Vector Convert Floating-Point to Signed Integer
EVX	4	1000029A			651	SP.FV	evfsctsiz	Vector Convert Floating-Point to Signed Integer with Round toward Zero

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000296			652	SP.FV	evfsctuf	Vector Convert Floating-Point to Unsigned Fraction
EVX	4	10000294			651	SP.FV	evfsctui	Vector Convert Floating-Point to Unsigned Integer
EVX	4	10000298			651	SP.FV	evfsctuiz	Vector Convert Floating-Point to Unsigned Integer with Round toward Zero
EVX	4	10000289			646	SP.FV	evfsdiv	Vector Floating-Point Divide
EVX	4	10000288			646	SP.FV	evfsmul	Vector Floating-Point Multiply
EVX	4	10000285			645	SP.FV	evfsnabs	Vector Floating-Point Negative Absolute Value
EVX	4	10000286			645	SP.FV	evfsneg	Vector Floating-Point Negate
EVX	4	10000281			646	SP.FV	evfssub	Vector Floating-Point Subtract
EVX	4	1000029E			649	SP.FV	evfststeq	Vector Floating-Point Test Equal
EVX	4	1000029C			648	SP.FV	evfststgt	Vector Floating-Point Test Greater Than
EVX	4	1000029D			649	SP.FV	evfststlt	Vector Floating-Point Test Less Than
X	31	7C00071D			808	TM	tabort.	Transaction Abort
X	31	7C00065D			809	TM	tabortdc.	Transaction Abort Doubleword Conditional
X	31	7C0006DD			810	TM	tabortdci.	Transaction Abort Doubleword Conditional Immediate
X	31	7C00061D			809	TM	tabortwc.	Transaction Abort Word Conditional
X	31	7C00069D			809	TM	tabortwci.	Transaction Abort Word Conditional Immediate
X	31	7C00051D			806	TM	tbegin.	Transaction Begin
X	31	7C00059C			811	TM	tcheck	Transaction Check
X	31	7C00055C			807	TM	tend.	Transaction End
X	31	7C0007DD			880	TM	trechkpt.	Transaction Recheckpoint
X	31	7C00075D			879	TM	treclaim.	Transaction Reclaim
X	31	7C0005DC			810	TM	tsr.	Transaction Suspend or Resume
VX	60	F0000400			267	V	bcdadd.	Decimal Add Modulo
VX	60	F0000440			267	V	bcdsub.	Decimal Subtract Modulo
X	31	7C00000E			184	V	lvebx	Load Vector Element Byte Indexed
X	31	7C00004E			181	V	lvehx	Load Vector Element Halfword Indexed
X	31	7C00008E			181	V	lvewx	Load Vector Element Word Indexed
X	31	7C00000C			186	V	lvsl	Load Vector for Shift Left
X	31	7C00004C			186	V	lvsr	Load Vector for Shift Right
X	31	7C0000CE			182	V	lvx	Load Vector Indexed
X	31	7C0002CE			182	V	lvxl	Load Vector Indexed Last
VX	4	10000604			268	V	mfvscr	Move From Vector Status and Control Register
VX	4	10000644			268	V	mtvscr	Move To Vector Status and Control Register
X	31	7C00010E			184	V	stvebx	Store Vector Element Byte Indexed
X	31	7C00014E			184	V	stvehx	Store Vector Element Halfword Indexed
X	31	7C00018E			185	V	stvewx	Store Vector Element Word Indexed
X	31	7C0001CE			182	V	stvx	Store Vector Indexed
X	31	7C0003CE			185	V	stvxl	Store Vector Indexed Last
VX	4	10000140			206	V	vaddcuq	Vector Add & write Carry Unsigned Quadword
VX	4	10000180			202	V	vaddcuw	Vector Add and Write Carry-Out Unsigned Word
VA	4	1000003D			206	V	vaddecuq	Vector Add Extended & write Carry Unsigned Quadword
VA	4	1000003C			206	V	vaddeuqm	Vector Add Extended Unsigned Quadword Modulo
VX	4	1000000A			244	V	vaddfp	Vector Add Single-Precision
VX	4	10000300			202	V	vaddsbs	Vector Add Signed Byte Saturate
VX	4	10000340			202	V	vaddshs	Vector Add Signed Halfword Saturate
VX	4	10000380			203	V	vaddsws	Vector Add Signed Word Saturate

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	10000000			203	V	vaddubm	Vector Add Unsigned Byte Modulo
VX	4	10000200			205	V	vaddubs	Vector Add Unsigned Byte Saturate
VX	4	100000C0			203	V	vaddudm	Vector Add Unsigned Doubleword Modulo
VX	4	10000040			203	V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	4	10000240			205	V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	4	10000100			206	V	vadduqm	Vector Add Unsigned Quadword Modulo
VX	4	10000080			204	V	vadduwm	Vector Add Unsigned Word Modulo
VX	4	10000280			205	V	vadduws	Vector Add Unsigned Word Saturate
VX	4	10000404			238	V	vand	Vector Logical AND
VX	4	10000444			238	V	vandc	Vector Logical AND with Complement
VX	4	10000502			226	V	vavgsb	Vector Average Signed Byte
VX	4	10000542			226	V	vavgsh	Vector Average Signed Halfword
VX	4	10000582			226	V	vavgsw	Vector Average Signed Word
VX	4	10000402			227	V	vavgub	Vector Average Unsigned Byte
VX	4	10000442			227	V	vavguh	Vector Average Unsigned Halfword
VX	4	10000482			227	V	vavguw	Vector Average Unsigned Word
VX	4	1000054C			265	V	vbpermq	Vector Bit Permute Quadword
VX	4	1000054C			265	V	vbpermq	Vector Bit Permute Quadword
VX	4	1000034A			248	V	vcfsx	Vector Convert From Signed Fixed-Point Word To Single-Precision
VX	4	1000030A			248	V	vcfux	Vector Convert From Unsigned Fixed-Point Word To Single-Precision
VX	4	10000702			263	V	vclzb	Vector Count Leading Zeros Byte
VX	4	100007C2			263	V	vclzd	Vector Count Leading Zeros Doubleword
VX	4	10000742			263	V	vclzh	Vector Count Leading Zeros Halfword
VX	4	10000782			263	V	vclzw	Vector Count Leading Zeros Word
VC	4	100003C6			251	V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VC	4	100000C6			252	V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VC	4	10000006			232	V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VC	4	100000C7			233	V	vcmpequd[.]	Vector Compare Equal To Unsigned Doubleword
VC	4	10000046			233	V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VC	4	10000086			233	V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VC	4	100001C6			252	V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VC	4	100002C6			253	V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VC	4	10000306			234	V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
VC	4	100003C7			234	V	vcmpgtsd[.]	Vector Compare Greater Than Signed Doubleword
VC	4	10000346			234	V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VC	4	10000386			235	V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VC	4	10000206			236	V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
VC	4	100002C7			236	V	vcmpgtud[.]	Vector Compare Greater Than Unsigned Doubleword
VC	4	10000246			236	V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VC	4	10000286			237	V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
VX	4	100003CA			247	V	vctsxs	Vector Convert From Single-Precision To Signed Fixed-Point Word Saturate
VX	4	1000038A			247	V	vctuxs	Vector Convert From Single-Precision To Unsigned Fixed-Point Word Saturate
VX	4	10000684			238	V	veqv	Vector Equivalence

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	1000018A			254	V	vexptefp	Vector 2 Raised to the Exponent Estimate Single-Precision
VX	4	1000050C			262	V	vgbbd	Vector Gather Bits by Byte by Doubleword
VX	4	100001CA			254	V	vlogefp	Vector Log Base 2 Estimate Single-Precision
VA	4	1000002E			245	V	vmaddfp	Vector Multiply-Add Single-Precision
VX	4	1000040A			246	V	vmaxfp	Vector Maximum Single-Precision
VX	4	10000102			228	V	vmaxsb	Vector Maximum Signed Byte
VX	4	100001C2			228	V	vmaxsd	Vector Maximum Signed Doubleword
VX	4	10000142			228	V	vmaxsh	Vector Maximum Signed Halfword
VX	4	10000182			228	V	vmaxsw	Vector Maximum Signed Word
VX	4	10000002			228	V	vmaxub	Vector Maximum Unsigned Byte
VX	4	100000C2			228	V	vmaxud	Vector Maximum Unsigned Doubleword
VX	4	10000042			228	V	vmaxuh	Vector Maximum Unsigned Halfword
VX	4	10000082			229	V	vmaxuw	Vector Maximum Unsigned Word
VA	4	10000020			218	V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	4	10000021			218	V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VX	4	1000044A			246	V	vminfp	Vector Minimum Single-Precision
VX	4	10000302			230	V	vminsb	Vector Minimum Signed Byte
X	4	100003C2			230	V	vminsd	Vector Minimum Signed Doubleword
VX	4	10000342			230	V	vminsh	Vector Minimum Signed Halfword
VX	4	10000382			231	V	vminsw	Vector Minimum Signed Word
VX	4	10000202			230	V	vminub	Vector Minimum Unsigned Byte
VX	4	100002C2			230	V	vminud	Vector Minimum Unsigned Doubleword
VX	4	10000242			230	V	vminuh	Vector Minimum Unsigned Halfword
VX	4	10000282			231	V	vminuw	Vector Minimum Unsigned Word
VA	4	10000022			219	V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VX	4	1000000C			194	V	vmrghb	Vector Merge High Byte
VX	4	1000004C			194	V	vmrghh	Vector Merge High Halfword
VX	4	1000008C			195	V	vmrghw	Vector Merge High Word
VX	4	1000010C			194	V	vmrglb	Vector Merge Low Byte
VX	4	1000014C			194	V	vmrglh	Vector Merge Low Halfword
VX	4	1000018C			195	V	vmrglw	Vector Merge Low Word
VA	4	10000025			220	V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	4	10000028			220	V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	4	10000029			221	V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	4	10000024			219	V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	4	10000026			221	V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	4	10000027			222	V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VX	4	10000308			214	V	vmulesb	Vector Multiply Even Signed Byte
VX	4	10000348			215	V	vmulesh	Vector Multiply Even Signed Halfword
VX	4	10000388			216	V	vmulesw	Vector Multiply Even Signed Word
VX	4	10000208			214	V	vmuleub	Vector Multiply Even Unsigned Byte
VX	4	10000248			215	V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	4	10000288			216	V	vmuleuw	Vector Multiply Even Unsigned Word
VX	4	10000108			214	V	vmulosb	Vector Multiply Odd Signed Byte
VX	4	10000148			215	V	vmulosh	Vector Multiply Odd Signed Halfword
VX	4	10000188			216	V	vmulosw	Vector Multiply Odd Signed Word
VX	4	10000008			214	V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	4	10000048			215	V	vmulouh	Vector Multiply Odd Unsigned Halfword

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	10000088			216	V	vmulouw	Vector Multiply Odd Unsigned Word
VX	4	10000089			217	V	vmuluwm	Vector Multiply Unsigned Word Modulo
VX	4	10000584			238	V	vnand	Vector NAND
VA	4	1000002F			245	V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	4	10000504			239	V	vnor	Vector Logical NOR
VX	4	10000484			239	V	vor	Vector Logical OR
VX	4	10000544			239	V	vorc	Vector OR with Complement
VA	4	1000002B			198	V	vperm	Vector Permute
VX	4	1000030E			187	V	vpkpx	Vector Pack Pixel
VX	4	100005CE			187	V	vpksdss	Vector Pack Signed Doubleword Signed Saturate
VX	4	1000054E			188	V	vpksdus	Vector Pack Signed Doubleword Unsigned Saturate
VX	4	1000018E			188	V	vpkshss	Vector Pack Signed Halfword Signed Saturate
VX	4	1000010E			189	V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	4	100001CE			189	V	vpkswss	Vector Pack Signed Word Signed Saturate
VX	4	1000014E			190	V	vpkswus	Vector Pack Signed Word Unsigned Saturate
VX	4	1000044E			190	V	vpkudum	Vector Pack Unsigned Doubleword Unsigned Modulo
VX	4	100004CE			190	V	vpkudus	Vector Pack Unsigned Doubleword Unsigned Saturate
VX	4	1000000E			190	V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VX	4	1000008E			191	V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
VX	4	1000004E			191	V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
VX	4	100000CE			191	V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
VX	4	10000408			259	V	vpmsumb	Vector Polynomial Multiply-Sum Byte
VX	4	100004C8			259	V	vpmsumd	Vector Polynomial Multiply-Sum Doubleword
VX	4	10000448			260	V	vpmsumh	Vector Polynomial Multiply-Sum Halfword
VX	4	10000488			260	V	vpmsumw	Vector Polynomial Multiply-Sum Word
VX	4	10000703			264	V	vpopcntb	Vector Population Count Byte
VX	4	100007C3			264	V	vpopcntd	Vector Population Count Doubleword
VX	4	10000743			264	V	vpopcnth	Vector Population Count Halfword
VX	4	10000783			264	V	vpopcntw	Vector Population Count Word
VX	4	1000010A			255	V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	4	100002CA			250	V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
VX	4	1000020A			249	V	vrfin	Vector Round to Single-Precision Integer Nearest
VX	4	1000028A			249	V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
VX	4	1000024A			249	V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	4	10000004			240	V	vrlb	Vector Rotate Left Byte
VX	4	100000C4			240	V	vrld	Vector Rotate Left Doubleword
VX	4	10000044			240	V	vrlh	Vector Rotate Left Halfword
VX	4	10000084			240	V	vrlw	Vector Rotate Left Word
VX	4	1000014A			255	V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VA	4	1000002A			199	V	vsel	Vector Select
VX	4	100001C4			200	V	vsl	Vector Shift Left
VX	4	10000104			241	V	vslb	Vector Shift Left Byte

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	100005C4			241	V	vsld	Vector Shift Left Doubleword
VA	4	1000002C			200	V	vsldoi	Vector Shift Left Double by Octet Immediate
VX	4	10000144			241	V	vslh	Vector Shift Left Halfword
VX	4	1000040C			200	V	vslo	Vector Shift Left by Octet
VX	4	10000184			241	V	vslw	Vector Shift Left Word
VX	4	1000020C			197	V	vspltb	Vector Splat Byte
VX	4	1000024C			197	V	vsplth	Vector Splat Halfword
VX	4	1000030C			198	V	vspltisb	Vector Splat Immediate Signed Byte
VX	4	1000034C			198	V	vspltish	Vector Splat Immediate Signed Halfword
VX	4	1000038C			198	V	vspltisw	Vector Splat Immediate Signed Word
VX	4	1000028C			197	V	vspltw	Vector Splat Word
VX	4	100002C4			201	V	vsr	Vector Shift Right
VX	4	10000304			243	V	vsrab	Vector Shift Right Algebraic Byte
VX	4	100003C4			243	V	vsrad	Vector Shift Right Algebraic Doubleword
VX	4	10000344			243	V	vsrah	Vector Shift Right Algebraic Halfword
VX	4	10000384			243	V	vsraw	Vector Shift Right Algebraic Word
VX	4	10000204			242	V	vsrb	Vector Shift Right Byte
VX	4	100006C4			242	V	vsrd	Vector Shift Right Doubleword
VX	4	10000244			242	V	vsrh	Vector Shift Right Halfword
VX	4	1000044C			201	V	vsro	Vector Shift Right by Octet
VX	4	10000284			242	V	vsrw	Vector Shift Right Word
VX	4	10000540			212	V	vsubcuq	Vector Subtract & write Carry Unsigned Quadword
VX	4	10000580			208	V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
VA	4	1000003F			212	V	vsubecuq	Vector Subtract Extended & write Carry Unsigned Quadword
VA	4	1000003E			212	V	vsubeuqm	Vector Subtract Extended Unsigned Quadword Modulo
VX	4	1000004A			244	V	vsubfp	Vector Subtract Single-Precision
VX	4	10000700			208	V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	4	10000740			208	V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	4	10000780			209	V	vsubsws	Vector Subtract Signed Word Saturate
VX	4	10000400			210	V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	4	10000600			211	V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	4	100004C0			210	V	vsubudm	Vector Subtract Unsigned Doubleword Modulo
VX	4	10000440			210	V	vsubuhm	Vector Subtract Unsigned Halfword Modulo
VX	4	10000640			210	V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	4	10000500			212	V	vsubuqm	Vector Subtract Unsigned Quadword Modulo
VX	4	10000480			210	V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	4	10000680			211	V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	4	10000688			223	V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	4	10000708			224	V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	4	10000648			224	V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	4	10000608			225	V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	4	10000788			223	V	vsumsws	Vector Sum across Signed Word Saturate
VX	4	1000034E			190	V	vupkhpx	Vector Unpack High Pixel
VX	4	1000020E			193	V	vupkhsb	Vector Unpack High Signed Byte
VX	4	1000024E			193	V	vupkhsh	Vector Unpack High Signed Halfword
VX	4	1000064E			193	V	vupkhsw	Vector Unpack High Signed Word
VX	4	100003CE			192	V	vupklpx	Vector Unpack Low Pixel
VX	4	1000028E			193	V	vupklsb	Vector Unpack Low Signed Byte

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	100002CE			193	V	vupklsh	Vector Unpack Low Signed Halfword
VX	4	100006CE			193	V	vupklsw	Vector Unpack Low Signed Word
VX	4	100004C4			239	V	vxor	Vector Logical XOR
VX	4	10000508			256	V.Crypto	vcipher	Vector AES Cipher
VX	4	10000509			256	V.Crypto	vcipherlast	Vector AES Cipher Last
VX	4	10000548			257	V.Crypto	vncipher	Vector AES Inverse Cipher
VX	4	10000549			257	V.Crypto	vncipherlast	Vector AES Inverse Cipher Last
VX	4	100005C8			257	V.Crypto	vsbox	Vector AES S-Box
VX	4	100006C2			258	V.Crypto	vshasigmad	Vector SHA-512 Sigma Doubleword
VX	4	10000682			258	V.Crypto	vshasigmaw	Vector SHA-256 Sigma Word
VA	4	1000002D			261	V.RAID	vpermxor	Vector Permute and Exclusive-OR
X	63	FC00078C			143	VSX	fmrgew	Floating Merge Even Word
X	63	FC00068C			143	VSX	fmrgow	Floating Merge Odd Word
XX1	31	7C000498			393	VSX	lxsdx	Load VSR Scalar Doubleword Indexed
XX1	31	7C000098			393	VSX	lxsiwax	Load VSX Scalar as Integer Word Algebraic Indexed
XX1	31	7C000018			394	VSX	lxsiwzx	Load VSX Scalar as Integer Word and Zero Indexed
XX1	31	7C000418			394	VSX	lxsspx	Load VSX Scalar Single-Precision Indexed
XX1	31	7C000698			395	VSX	lxvd2x	Load VSR Vector Doubleword*2 Indexed
XX1	31	7C000298			395	VSX	lxvdsx	Load VSR Vector Doubleword & Splat Indexed
XX1	31	7C000618			396	VSX	lxvw4x	Load VSR Vector Word*4 Indexed
XX1	31	7C000066			109	VSX	mfvsrd	Move From VSR Doubleword
XX1	31	7C0000E6			109	VSX	mfvsrwz	Move From VSR Word and Zero
XX2	31	7C000166			110	VSX	mtvsrd	Move To VSR Doubleword
XX2	31	7C0001A6			110	VSX	mtvsrwa	Move To VSR Word Algebraic
XX2	31	7C0001E6			111	VSX	mtvsrwz	Move To VSR Word and Zero
XX1	31	7C000598			396	VSX	stxsdx	Store VSR Scalar Doubleword Indexed
XX1	31	7C000118			394	VSX	stxsiwx	Store VSX Scalar as Integer Word Indexed
XX1	31	7C000518			394	VSX	stxsspx	Store VSR Scalar Word Indexed
XX1	31	7C000798			398	VSX	stxvd2x	Store VSR Vector Doubleword*2 Indexed
XX1	31	7C000718			398	VSX	stxvw4x	Store VSR Vector Word*4 Indexed
VX	4	1000078C			196	VSX	vmrgew	Vector Merge Even Word
VX	4	1000068C			196	VSX	vmrgow	Vector Merge Odd Word
XX2	60	F0000564			399	VSX	xsabsdp	VSX Scalar Absolute Value Double-Precision
XX3	60	F0000100			400	VSX	xsadddp	VSX Scalar Add Double-Precision
XX3	60	F0000000			405	VSX	xsaddsp	VSX Scalar Add Single-Precision
XX3	60	F0000158			407	VSX	xscmpodp	VSX Scalar Compare Ordered Double-Precision
XX3	60	F0000118			409	VSX	xscmpudp	VSX Scalar Compare Unordered Double-Precision
XX3	60	F0000580			411	VSX	xscpsgndp	VSX Scalar Copy Sign Double-Precision
XX2	60	F0000424			412	VSX	xscvdpsp	VSX Scalar Convert Double-Precision to Single-Precision
XX2	60	F000042C			413	VSX	xscvdpspn	VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling
XX2	60	F0000560			422	VSX	xscvdpsxds	VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000160			413	VSX	xscvdpsxws	VSX Scalar Convert Double-Precision to Signed Fixed-Point Word Saturate
XX2	60	F0000520			416	VSX	xscvdpuxds	VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000120			418	VSX	xscvdpuxws	VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F0000524			420	VSX	xscvspdp	VSX Scalar Convert Single-Precision to Double-Precision (p=1)

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F000052C			422	VSX	xscvspdpn	Scalar Convert Single-Precision to Double-Precision format Non-signalling
XX2	60	F00005E0			423	VSX	xscvsxddp	VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision
XX2	60	F00004E0			423	VSX	xscvsxdsp	VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision
XX2	60	F00005A0			424	VSX	xscvuxddp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Double-Precision
XX2	60	F00004A0			424	VSX	xscvuxdsp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision
XX3	60	F00001C0			425	VSX	xsdivdp	VSX Scalar Divide Double-Precision
XX3	60	F00000C0			427	VSX	xsdivsp	VSX Scalar Divide Single-Precision
XX3	60	F0000108			429	VSX	xsmaddadp	VSX Scalar Multiply-Add Type-A Double-Precision
XX3	60	F0000008			432	VSX	xsmaddasp	VSX Scalar Multiply-Add Type-A Single-Precision
XX3	60	F0000148			429	VSX	xsmaddmdp	VSX Scalar Multiply-Add Type-M Double-Precision
XX3	60	F0000048			432	VSX	xsmaddmsp	VSX Scalar Multiply-Add Type-M Single-Precision
XX3	60	F0000500			435	VSX	xsmaxdp	VSX Scalar Maximum Double-Precision
XX3	60	F0000540			437	VSX	xsmindp	VSX Scalar Minimum Double-Precision
XX3	60	F0000188			439	VSX	xsmsubadp	VSX Scalar Multiply-Subtract Type-A Double-Precision
XX3	60	F0000088			442	VSX	xsmsubasp	VSX Scalar Multiply-Subtract Type-A Single-Precision
XX3	60	F00001C8			439	VSX	xsmsubmdp	VSX Scalar Multiply-Subtract Type-M Double-Precision
XX3	60	F00000C8			442	VSX	xsmsubmsp	VSX Scalar Multiply-Subtract Type-M Single-Precision
XX3	60	F0000180			445	VSX	xsmuldp	VSX Scalar Multiply Double-Precision
XX3	60	F0000080			447	VSX	xsmulsp	VSX Scalar Multiply Single-Precision
XX2	60	F00005A4			449	VSX	xsnabsdp	VSX Scalar Negative Absolute Value Double-Precision
XX2	60	F00005E4			449	VSX	xsnegdp	VSX Scalar Negate Double-Precision
XX3	60	F0000508			450	VSX	xsnmaddadp	VSX Scalar Negative Multiply-Add Type-A Double-Precision
XX3	60	F0000408			455	VSX	xsnmaddasp	VSX Scalar Negative Multiply-Add Type-A Single-Precision
XX3	60	F0000548			450	VSX	xsnmaddmdp	VSX Scalar Negative Multiply-Add Type-M Double-Precision
XX3	60	F0000448			455	VSX	xsnmaddmsp	VSX Scalar Negative Multiply-Add Type-M Single-Precision
XX3	60	F0000588			458	VSX	xsnmsubadp	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision
XX3	60	F0000488			461	VSX	xsnmsubasp	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
XX3	60	F00005C8			458	VSX	xsnmsubmdp	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision
XX3	60	F00004C8			461	VSX	xsnmsubmsp	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision
XX2	60	F0000124			464	VSX	xsrdpi	VSX Scalar Round to Double-Precision Integer
XX2	60	F00001AC			465	VSX	xsrdpic	VSX Scalar Round to Double-Precision Integer using Current rounding mode
XX2	60	F00001E4			466	VSX	xsrdpim	VSX Scalar Round to Double-Precision Integer toward -Infinity

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F00001A4			466	VSX	xsrdpip	VSX Scalar Round to Double-Precision Integer toward +Infinity
XX2	60	F0000164			467	VSX	xsrdpiz	VSX Scalar Round to Double-Precision Integer toward Zero
XX3	60	F0000168			468	VSX	xsredp	VSX Scalar Reciprocal Estimate Double-Precision
XX3	60	F0000068			469	VSX	xsresp	VSX Scalar Reciprocal Estimate Single-Precision
XX2	60	F0000464			470	VSX	xsrsp	VSX Scalar Round to Single-Precision
XX2	60	F0000128			471	VSX	xsrsqrtedp	VSX Scalar Reciprocal Square Root Estimate Double-Precision
XX2	60	F0000028			472	VSX	xsrsqrtesp	VSX Scalar Reciprocal Square Root Estimate Single-Precision
XX2	60	F000012C			473	VSX	xssqrtdp	VSX Scalar Square Root Double-Precision
XX2	60	F000002C			474	VSX	xssqrtsp	VSX Scalar Square Root Single-Precision
XX3	60	F0000140			475	VSX	xssubdp	VSX Scalar Subtract Double-Precision
XX3	60	F0000040			477	VSX	xssubsp	VSX Scalar Subtract Single-Precision
XX3	60	F00001E8			479	VSX	xstdivdp	VSX Scalar Test for software Divide Double-Precision
XX2	60	F00001A8			480	VSX	xstsqrtdp	VSX Scalar Test for software Square Root Double-Precision
XX2	60	F0000764			480	VSX	xvabsdp	VSX Vector Absolute Value Double-Precision
XX2	60	F0000664			481	VSX	xvabssp	VSX Vector Absolute Value Single-Precision
XX3	60	F0000300			482	VSX	xvadddp	VSX Vector Add Double-Precision
XX3	60	F0000200			486	VSX	xvaddsp	VSX Vector Add Single-Precision
XX3	60	F0000318			488	VSX	xvcmpeqdp	VSX Vector Compare Equal To Double-Precision
XX3	60	F0000718			488	VSX	xvcmpeqdp.	VSX Vector Compare Equal To Double-Precision & record CR6
XX3	60	F0000218			489	VSX	xvcmpeqsp	VSX Vector Compare Equal To Single-Precision
XX3	60	F0000618			489	VSX	xvcmpeqsp.	VSX Vector Compare Equal To Single-Precision & record CR6
XX3	60	F0000398			490	VSX	xvcmpgedp	VSX Vector Compare Greater Than or Equal To Double-Precision
XX3	60	F0000798			490	VSX	xvcmpgedp.	VSX Vector Compare Greater Than or Equal To Double-Precision & record CR6
XX3	60	F0000298			491	VSX	xvcmpgesp	VSX Vector Compare Greater Than or Equal To Single-Precision
XX3	60	F0000698			491	VSX	xvcmpgesp.	VSX Vector Compare Greater Than or Equal To Single-Precision & record CR6
XX3	60	F0000358			492	VSX	xvcmpgtdp	VSX Vector Compare Greater Than Double-Precision
XX3	60	F0000758			492	VSX	xvcmpgtdp.	VSX Vector Compare Greater Than Double-Precision & record CR6
XX3	60	F0000258			493	VSX	xvcmpgtsp	VSX Vector Compare Greater Than Single-Precision
XX3	60	F0000658			493	VSX	xvcmpgtsp.	VSX Vector Compare Greater Than Single-Precision & record CR6
XX3	60	F0000780			494	VSX	xvcpsgndp	VSX Vector Copy Sign Double-Precision
XX3	60	F0000680			494	VSX	xvcpsgnsp	VSX Vector Copy Sign Single-Precision
XX2	60	F0000624			495	VSX	xvcvdpsp	VSX Vector Convert Double-Precision to Single-Precision
XX2	60	F0000760			496	VSX	xvcvdpsxds	VSX Vector Convert Double-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000360			498	VSX	xvcvdpsxws	VSX Vector Convert Double-Precision to Signed Fixed-Point Word Saturate

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F0000720			500	VSX	xvcvdpuxds	VSX Vector Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000320			502	VSX	xvcvdpuxws	VSX Vector Convert Double-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F0000724			504	VSX	xvcvspdp	VSX Vector Convert Single-Precision to Double-Precision
XX2	60	F0000660			505	VSX	xvcvspsxds	VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000260			507	VSX	xvcvspsxws	VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate
XX2	60	F0000620			509	VSX	xvcvspuxds	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000220			511	VSX	xvcvspuxws	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F00007E0			513	VSX	xvcvsxddp	VSX Vector Convert Signed Fixed-Point Doubleword to Double-Precision
XX2	60	F00006E0			513	VSX	xvcvsxdsp	VSX Vector Convert Signed Fixed-Point Doubleword to Single-Precision
XX2	60	F00003E0			514	VSX	xvcvsxwdp	VSX Vector Convert Signed Fixed-Point Word to Double-Precision
XX2	60	F00002E0			514	VSX	xvcvsxwsp	VSX Vector Convert Signed Fixed-Point Word to Single-Precision
XX2	60	F00007A0			515	VSX	xvcvuxddp	VSX Vector Convert Unsigned Fixed-Point Doubleword to Double-Precision
XX2	60	F00006A0			515	VSX	xvcvuxdsp	VSX Vector Convert Unsigned Fixed-Point Doubleword to Single-Precision
XX2	60	F00003A0			516	VSX	xvcvuxwdp	VSX Vector Convert Unsigned Fixed-Point Word to Double-Precision
XX2	60	F00002A0			516	VSX	xvcvuxwsp	VSX Vector Convert Unsigned Fixed-Point Word to Single-Precision
XX3	60	F00003C0			517	VSX	xvdivdp	VSX Vector Divide Double-Precision
XX3	60	F00002C0			519	VSX	xvdivsp	VSX Vector Divide Single-Precision
XX3	60	F0000308			521	VSX	xvmaddadp	VSX Vector Multiply-Add Type-A Double-Precision
XX3	60	F0000208			521	VSX	xvmaddasp	VSX Vector Multiply-Add Type-A Single-Precision
XX3	60	F0000348			524	VSX	xvmaddmdp	VSX Vector Multiply-Add Type-M Double-Precision
XX3	60	F0000248			524	VSX	xvmaddmsp	VSX Vector Multiply-Add Type-M Single-Precision
XX3	60	F0000700			527	VSX	xvmaxdp	VSX Vector Maximum Double-Precision
XX3	60	F0000600			529	VSX	xvmaxsp	VSX Vector Maximum Single-Precision
XX3	60	F0000740			531	VSX	xvmindp	VSX Vector Minimum Double-Precision
XX3	60	F0000640			533	VSX	xvminsp	VSX Vector Minimum Single-Precision
XX3	60	F0000388			535	VSX	xvmsubadp	VSX Vector Multiply-Subtract Type-A Double-Precision
XX3	60	F0000288			535	VSX	xvmsubasp	VSX Vector Multiply-Subtract Type-A Single-Precision
XX3	60	F00003C8			538	VSX	xvmsubmdp	VSX Vector Multiply-Subtract Type-M Double-Precision
XX3	60	F00002C8			538	VSX	xvmsubmsp	VSX Vector Multiply-Subtract Type-M Single-Precision
XX3	60	F0000380			541	VSX	xvmuldp	VSX Vector Multiply Double-Precision
XX3	60	F0000280			543	VSX	xvmulsp	VSX Vector Multiply Single-Precision
XX2	60	F00007A4			545	VSX	xvnabsdp	VSX Vector Negative Absolute Value Double-Precision
XX2	60	F00006A4			545	VSX	xvnabssp	VSX Vector Negative Absolute Value Single-Precision

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F00007E4			546	VSX	xvnegdp	VSX Vector Negate Double-Precision
XX2	60	F00006E4			546	VSX	xvnegsp	VSX Vector Negate Single-Precision
XX3	60	F0000708			547	VSX	xvnmaddadp	VSX Vector Negative Multiply-Add Type-A Double-Precision
XX3	60	F0000608			547	VSX	xvnmaddasp	VSX Vector Negative Multiply-Add Type-A Single-Precision
XX3	60	F0000748			552	VSX	xvnmaddmdp	VSX Vector Negative Multiply-Add Type-M Double-Precision
XX3	60	F0000648			552	VSX	xvnmaddmsp	VSX Vector Negative Multiply-Add Type-M Single-Precision
XX3	60	F0000788			555	VSX	xvnmsubadp	VSX Vector Negative Multiply-Subtract Type-A Double-Precision
XX3	60	F0000688			555	VSX	xvnmsubasp	VSX Vector Negative Multiply-Subtract Type-A Single-Precision
XX3	60	F00007C8			558	VSX	xvnmsubmdp	VSX Vector Negative Multiply-Subtract Type-M Double-Precision
XX3	60	F00006C8			558	VSX	xvnmsubmsp	VSX Vector Negative Multiply-Subtract Type-M Single-Precision
XX2	60	F0000324			561	VSX	xvrdpi	VSX Vector Round to Double-Precision Integer
XX2	60	F00003AC			561	VSX	xvrdpic	VSX Vector Round to Double-Precision Integer using Current rounding mode
XX2	60	F00003E4			562	VSX	xvrdpim	VSX Vector Round to Double-Precision Integer toward -Infinity
XX2	60	F00003A4			562	VSX	xvrdpip	VSX Vector Round to Double-Precision Integer toward +Infinity
XX2	60	F0000364			563	VSX	xvrdpiz	VSX Vector Round to Double-Precision Integer toward Zero
XX3	60	F0000368			564	VSX	xvredp	VSX Vector Reciprocal Estimate Double-Precision
XX3	60	F0000268			565	VSX	xvresp	VSX Vector Reciprocal Estimate Single-Precision
XX2	60	F0000224			566	VSX	xvrspi	VSX Vector Round to Single-Precision Integer
XX2	60	F00002AC			566	VSX	xvrspic	VSX Vector Round to Single-Precision Integer using Current rounding mode
XX2	60	F00002E4			567	VSX	xvrspim	VSX Vector Round to Single-Precision Integer toward -Infinity
XX2	60	F00002A4			567	VSX	xvrspip	VSX Vector Round to Single-Precision Integer toward +Infinity
XX2	60	F0000264			568	VSX	xvrspiz	VSX Vector Round to Single-Precision Integer toward Zero
XX2	60	F0000328			568	VSX	xvrsqrtedp	VSX Vector Reciprocal Square Root Estimate Double-Precision
XX2	60	F0000228			570	VSX	xvrsqrtesp	VSX Vector Reciprocal Square Root Estimate Single-Precision
XX2	60	F000032C			571	VSX	xvsqrtdp	VSX Vector Square Root Double-Precision
XX2	60	F000022C			572	VSX	xvsqrtsp	VSX Vector Square Root Single-Precision
XX3	60	F0000340			573	VSX	xvsubdp	VSX Vector Subtract Double-Precision
XX3	60	F0000240			575	VSX	xvsubsp	VSX Vector Subtract Single-Precision
XX3	60	F00003E8			577	VSX	xvtdivdp	VSX Vector Test for software Divide Double-Precision
XX3	60	F00002E8			578	VSX	xvtdivsp	VSX Vector Test for software Divide Single-Precision
XX2	60	F00003A8			579	VSX	xvtsqrtdp	VSX Vector Test for software Square Root Double-Precision
XX2	60	F00002A8			579	VSX	xvtsqrtsp	VSX Vector Test for software Square Root Single-Precision
XX3	60	F0000410			580	VSX	xxland	VSX Logical AND

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX3	60	F0000450			580	VSX	xxlandc	VSX Logical AND with Complement
XX3	60	F00005D0			581	VSX	xxleqv	VSX Logical Equivalence
XX3	60	F0000590			581	VSX	xxlnand	VSX Logical NAND
XX3	60	F0000510			582	VSX	xxlnor	VSX Logical NOR
XX3	60	F0000490			583	VSX	xxlor	VSX Logical OR
XX3	60	F0000550			582	VSX	xxlorc	VSX Logical OR with Complement
XX3	60	F00004D0			583	VSX	xxlxor	VSX Logical XOR
XX3	60	F0000090			584	VSX	xxmrghw	VSX Merge High Word
XX3	60	F0000190			584	VSX	xxmrglw	VSX Merge Low Word
XX3	60	F0000050			585	VSX	xxpermdi	VSX Permute Doubleword Immediate
XX4	60	F0000030			585	VSX	xxsel	VSX Select
XX3	60	F0000010			586	VSX	xxsldwi	VSX Shift Left Double by Word Immediate
XX3	60	F0000290			586	VSX	xxspltw	VSX Splat Word
X	31	7C00007C			791	WT	wait	Wait for Interrupt

[1] See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.

Appendix H. Power ISA Instruction Set Sorted by Opcode

This appendix lists all the instructions in the Power ISA, sorted by opcode.

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
D	2	08000000			81	64	tdi	Trap Doubleword Immediate
D	3	0C000000			80	B	twi	Trap Word Immediate
VX	4	10000000			203	V	vaddubm	Vector Add Unsigned Byte Modulo
VX	4	10000002			228	V	vmaxub	Vector Maximum Unsigned Byte
VX	4	10000004			240	V	vrlb	Vector Rotate Left Byte
VC	4	10000006			232	V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VX	4	10000008			214	V	vmuloub	Vector Multiply Odd Unsigned Byte
X	4	10000010			681	LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
XO	4	10000018			678	LMA	machhwu[.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
XO	4	10000018			678	LMA	machhwuo[.]	Multiply Accumulate High Halfword to Word Modulo Unsigned & record OV
VA	4	10000020			218	V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	4	10000021			218	V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VA	4	10000022			219	V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VA	4	10000024			219	V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	4	10000025			220	V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	4	10000026			221	V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	4	10000027			222	V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VA	4	10000028			220	V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo
VA	4	10000029			221	V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VX	4	10000040			203	V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	4	10000042			228	V	vmaxuh	Vector Maximum Unsigned Halfword
VX	4	10000044			240	V	vrlh	Vector Rotate Left Halfword
VC	4	10000046			233	V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VX	4	10000048			215	V	vmulouh	Vector Multiply Odd Unsigned Halfword
X	4	10000050			681	LMA	mulhhw[.]	Multiply High Halfword to Word Signed
XO	4	10000058			677	LMA	machhw[.]	Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	10000058			677	LMA	machhwo[.]	Multiply Accumulate High Halfword to Word Modulo Signed & record OV
VX	4	10000080			204	V	vadduwm	Vector Add Unsigned Word Modulo
VX	4	10000082			229	V	vmaxuw	Vector Maximum Unsigned Word
VX	4	10000084			240	V	vrlw	Vector Rotate Left Word
VC	4	10000086			233	V	vcmpequw[.]	Vector Compare Equal To Unsigned Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	10000088			216	V	vmulouw	Vector Multiply Odd Unsigned Word
VX	4	10000089			217	V	vmuluwm	Vector Multiply Unsigned Word Modulo
XO	4	10000098			678	LMA	machhwsu[.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
XO	4	10000098			678	LMA	machhwsuo[.]	Multiply Accumulate High Halfword to Word Saturate Unsigned & record OV
VX	4	10000100			206	V	vadduqm	Vector Add Unsigned Quadword Modulo
VX	4	10000102			228	V	vmaxsb	Vector Maximum Signed Byte
VX	4	10000104			241	V	vslb	Vector Shift Left Byte
VX	4	10000108			214	V	vmulosb	Vector Multiply Odd Signed Byte
X	4	10000110			680	LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
XO	4	10000118			676	LMA	macchwu[.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
XO	4	10000118			676	LMA	macchwuo[.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned & record OV
VX	4	10000140			206	V	vaddcuq	Vector Add & write Carry Unsigned Quadword
VX	4	10000142			228	V	vmaxsh	Vector Maximum Signed Halfword
VX	4	10000144			241	V	vslh	Vector Shift Left Halfword
VX	4	10000148			215	V	vmulosh	Vector Multiply Odd Signed Halfword
X	4	10000150			680	LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
XO	4	10000158			675	LMA	macchw[.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	10000158			675	LMA	macchwo[.]	Multiply Accumulate Cross Halfword to Word Modulo Signed & record OV
VX	4	10000180			202	V	vaddcuw	Vector Add and Write Carry-Out Unsigned Word
VX	4	10000182			228	V	vmaxsw	Vector Maximum Signed Word
VX	4	10000184			241	V	vslw	Vector Shift Left Word
VX	4	10000188			216	V	vmulosw	Vector Multiply Odd Signed Word
XO	4	10000198			676	LMA	macchwsu[.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
XO	4	10000198			676	LMA	macchwsuo[.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned & record OV
EVX	4	10000200			595	SP	evaddw	Vector Add Word
VX	4	10000200			205	V	vaddubs	Vector Add Unsigned Byte Saturate
EVX	4	10000202			594	SP	evaddiw	Vector Add Immediate Word
VX	4	10000202			230	V	vminub	Vector Minimum Unsigned Byte
EVX	4	10000204			639	SP	evsubfw	Vector Subtract from Word
VX	4	10000204			242	V	vsrb	Vector Shift Right Byte
EVX	4	10000206			639	SP	evsubifw	Vector Subtract Immediate from Word
VC	4	10000206			236	V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
EVX	4	10000208			594	SP	evabs	Vector Absolute Value
VX	4	10000208			214	V	vmuleub	Vector Multiply Even Unsigned Byte
EVX	4	10000209			631	SP	evneg	Vector Negate
EVX	4	10000211			596	SP	evand	Vector AND
EVX	4	10000212			596	SP	evandc	Vector AND with Complement
EVX	4	10000216			639	SP	evxor	Vector XOR
EVX	4	10000217			632	SP	evor	Vector OR
EVX	4	10000218			631	SP	evnor	Vector NOR
EVX	4	10000219			599	SP	eveqv	Vector Equivalent
EVX	4	10000220			635	SP	evsrwu	Vector Shift Right Word Unsigned
EVX	4	10000221			635	SP	evsrws	Vector Shift Right Word Signed
EVX	4	10000222			634	SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	4	10000223			634	SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	4	10000224			634	SP	evslw	Vector Shift Left Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000226			634	SP	evslwi	Vector Shift Left Word Immediate
EVX	4	10000228			632	SP	evrlw	Vector Rotate Left Word
EVX	4	10000229			634	SP	evsplati	Vector Splat Immediate
EVX	4	10000230			597	SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	4	10000231			596	SP	evcmpgts	Vector Compare Greater Than Signed
EVX	4	10000232			597	SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	4	10000233			597	SP	evcmplts	Vector Compare Less Than Signed
EVX	4	10000234			596	SP	evcmpeq	Vector Compare Equal
VX	4	10000240			205	V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	4	10000242			230	V	vminuh	Vector Minimum Unsigned Halfword
VX	4	10000244			242	V	vsrh	Vector Shift Right Halfword
VC	4	10000246			236	V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VX	4	10000248			215	V	vmuleuh	Vector Multiply Even Unsigned Halfword
EVS	4	10000278			633	SP	evsel	Vector Select
EVX	4	10000280			646	SP.FV	evfsadd	Vector Floating-Point Add
VX	4	10000280			205	V	vadduws	Vector Add Unsigned Word Saturate
EVX	4	10000281			646	SP.FV	evfssub	Vector Floating-Point Subtract
VX	4	10000282			231	V	vminuw	Vector Minimum Unsigned Word
EVX	4	10000284			645	SP.FV	evfsabs	Vector Floating-Point Absolute Value
VX	4	10000284			242	V	vsrw	Vector Shift Right Word
EVX	4	10000285			645	SP.FV	evfsnabs	Vector Floating-Point Negative Absolute Value
EVX	4	10000286			645	SP.FV	evfsneg	Vector Floating-Point Negate
VC	4	10000286			237	V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
EVX	4	10000288			646	SP.FV	evfsmul	Vector Floating-Point Multiply
VX	4	10000288			216	V	vmuleuw	Vector Multiply Even Unsigned Word
EVX	4	10000289			646	SP.FV	evfsdiv	Vector Floating-Point Divide
EVX	4	10000290			650	SP.FV	evfscfui	Vector Convert Floating-Point from Unsigned Integer
EVX	4	10000291			650	SP.FV	evfscfsi	Vector Convert Floating-Point from Signed Integer
EVX	4	10000292			650	SP.FV	evfscfuf	Vector Convert Floating-Point from Unsigned Fraction
EVX	4	10000293			650	SP.FV	evfscfsf	Vector Convert Floating-Point from Signed Fraction
EVX	4	10000294			651	SP.FV	evfsctui	Vector Convert Floating-Point to Unsigned Integer
EVX	4	10000295			651	SP.FV	evfsctsi	Vector Convert Floating-Point to Signed Integer
EVX	4	10000296			652	SP.FV	evfsctuf	Vector Convert Floating-Point to Unsigned Fraction
EVX	4	10000297			652	SP.FV	evfsctsf	Vector Convert Floating-Point to Signed Fraction
EVX	4	10000298			651	SP.FV	evfsctuiz	Vector Convert Floating-Point to Unsigned Integer with Round toward Zero
EVX	4	10000300			600	SP	evlddx	Vector Load Double Word into Double Word Indexed
VX	4	10000300			202	V	vaddsbs	Vector Add Signed Byte Saturate
EVX	4	10000301			600	SP	evldd	Vector Load Double Word into Double Word
EVX	4	10000302			601	SP	evldwx	Vector Load Double into Two Words Indexed
VX	4	10000302			230	V	vminsb	Vector Minimum Signed Byte
EVX	4	10000303			601	SP	evldw	Vector Load Double into Two Words
EVX	4	10000304			600	SP	evldhx	Vector Load Double into Four Half Words Indexed
VX	4	10000304			243	V	vsrab	Vector Shift Right Algebraic Byte

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000305			600	SP	evldh	Vector Load Double into Four Half Words
VC	4	10000306			234	V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte
EVX	4	10000308			601	SP	evlhhesplatx	Vector Load Half Word into Half Words Even and Splat Indexed
VX	4	10000308			214	V	vmulesb	Vector Multiply Even Signed Byte
EVX	4	10000309			601	SP	evlhhesplat	Vector Load Half Word into Half Words Even and Splat
EVX	4	10000310			603	SP	evlwhex	Vector Load Word into Two Half Words Even Indexed
X	4	10000310			681	LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
EVX	4	10000311			603	SP	evlwhe	Vector Load Word into Two Half Words Even
EVX	4	10000314			604	SP	evlwhoux	Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)
EVX	4	10000315			604	SP	evlwhou	Vector Load Word into Two Half Words Odd Unsigned (zero-extended)
EVX	4	10000316			603	SP	evlwhosx	Vector Load Word into Two Half Words Odd Signed Indexed (with sign extension)
EVX	4	10000317			603	SP	evlwhos	Vector Load Word into Two Half Words Odd Signed (with sign extension)
EVX	4	10000318			605	SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
XO	4	10000318			680	LMA	maclhwu[.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
XO	4	10000318			680	LMA	maclhwuo[.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned & record OV
EVX	4	10000319			605	SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	4	10000320			635	SP	evstddx	Vector Store Double of Double Indexed
EVX	4	10000321			635	SP	evstdd	Vector Store Double of Double
EVX	4	10000322			636	SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	4	10000323			636	SP	evstdw	Vector Store Double of Two Words
EVX	4	10000324			636	SP	evstdhx	Vector Store Double of Four Half Words Indexed
EVX	4	10000325			636	SP	evstdh	Vector Store Double of Four Half Words
EVX	4	10000330			637	SP	evstwhex	Vector Store Word of Two Half Words from Even Indexed
EVX	4	10000331			637	SP	evstwhe	Vector Store Word of Two Half Words from Even
EVX	4	10000334			637	SP	evstwhox	Vector Store Word of Two Half Words from Odd Indexed
EVX	4	10000335			637	SP	evstwho	Vector Store Word of Two Half Words from Odd
EVX	4	10000338			637	SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	4	10000339			637	SP	evstwwe	Vector Store Word of Word from Even
VX	4	10000340			202	V	vaddshs	Vector Add Signed Halfword Saturate
VX	4	10000342			230	V	vminsh	Vector Minimum Signed Halfword
VX	4	10000344			243	V	vsrah	Vector Shift Right Algebraic Halfword
VC	4	10000346			234	V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VX	4	10000348			215	V	vmulesh	Vector Multiply Even Signed Halfword
X	4	10000350			681	LMA	mullhw[.]	Multiply Low Halfword to Word Signed
XO	4	10000358			679	LMA	maclhw[.]	Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	10000358			679	LMA	maclhwo[.]	Multiply Accumulate Low Halfword to Word Modulo Signed & record OV
VX	4	10000380			203	V	vaddsws	Vector Add Signed Word Saturate
VX	4	10000382			231	V	vminsw	Vector Minimum Signed Word
VX	4	10000384			243	V	vsraw	Vector Shift Right Algebraic Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VC	4	10000386			235	V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VX	4	10000388			216	V	vmulesw	Vector Multiply Even Signed Word
XO	4	10000398			680	LMA	maclhwsu[.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
XO	4	10000398			680	LMA	maclhwsuo[.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned & record OV
VX	4	10000400			210	V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	4	10000402			227	V	vavgub	Vector Average Unsigned Byte
EVX	4	10000403			610	SP	evmhessf	Vector Multiply Half Words, Even, Signed, Saturate, Fractional
VX	4	10000404			238	V	vand	Vector Logical AND
EVX	4	10000407			619	SP	evmhossf	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional
EVX	4	10000408			613	SP	evmheumi	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer
VX	4	10000408			259	V	vpmsumb	Vector Polynomial Multiply-Sum Byte
EVX	4	10000409			609	SP	evmhesmi	Vector Multiply Half Words, Even, Signed, Modulo, Integer
EVX	4	10000423			610	SP	evmhessfa	Vector Multiply Half Words, Even, Signed, Saturate, Fractional to Accumulator
EVX	4	10000427			619	SP	evmhossfa	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional to Accumulator
EVX	4	10000428			613	SP	evmheumia	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer to Accumulator
EVX	4	10000429			609	SP	evmhesmia	Vector Multiply Half Words, Even, Signed, Modulo, Integer to Accumulator
VX	4	10000440			210	V	vsubuhm	Vector Subtract Unsigned Halfword Modulo
VX	4	10000442			227	V	vavguh	Vector Average Unsigned Halfword
VX	4	10000444			238	V	vandc	Vector Logical AND with Complement
EVX	4	10000447			624	SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	4	10000448			626	SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
VX	4	10000448			260	V	vpmsumh	Vector Polynomial Multiply-Sum Halfword
EVX	4	10000453			629	SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	4	10000458			630	SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	4	10000459			628	SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	4	10000467			624	SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	4	10000468			626	SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	4	10000473			629	SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	4	10000478			630	SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	4	10000479			628	SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
VX	4	10000480			210	V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	4	10000482			227	V	vavguw	Vector Average Unsigned Word
VX	4	10000484			239	V	vor	Vector Logical OR
VX	4	10000488			260	V	vpmsumw	Vector Polynomial Multiply-Sum Word
EVX	4	10000500			614	SP	evmheusiaaw	Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words
VX	4	10000500			212	V	vsubuqm	Vector Subtract Unsigned Quadword Modulo

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000501			612	SP	evmhessiaaw	Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words
VX	4	10000502			226	V	vavgsb	Vector Average Signed Byte
EVX	4	10000503			611	SP	evmhessfaaw	Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	10000504			622	SP	evmhousiaaw	Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words
VX	4	10000504			239	V	vnor	Vector Logical NOR
EVX	4	10000505			621	SP	evmhossiaaw	Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	4	10000507			620	SP	evmhossfaaw	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	10000508			613	SP	evmheumiaaw	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words
VX	4	10000508			256	V.Crypto	vcipher	Vector AES Cipher
EVX	4	10000509			609	SP	evmhesmiaaw	Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words
VX	4	10000509			256	V.Crypto	vcipherlast	Vector AES Cipher Last
EVX	4	10000528			607	SP	evmhegumiaa	Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	10000529			607	SP	evmhegsmiaa	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	10000540			627	SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words
VX	4	10000540			212	V	vsubcuq	Vector Subtract & write Carry Unsigned Quadword
EVX	4	10000541			625	SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words
VX	4	10000542			226	V	vavgsh	Vector Average Signed Halfword
VX	4	10000544			239	V	vorc	Vector OR with Complement
EVX	4	10000548			626	SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words
VX	4	10000548			257	V.Crypto	vncipher	Vector AES Inverse Cipher
EVX	4	10000549			625	SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words
VX	4	10000549			257	V.Crypto	vncipherlast	Vector AES Inverse Cipher Last
EVX	4	10000553			629	SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	4	10000558			631	SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	4	10000559			628	SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	4	10000580			614	SP	evmheusianw	Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
VX	4	10000580			208	V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
EVX	4	10000581			612	SP	evmhessianw	Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words
VX	4	10000582			226	V	vavgsw	Vector Average Signed Word
EVX	4	10000583			611	SP	evmhessfanw	Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000584			622	SP	evmhousianw	Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
VX	4	10000584			238	V	vnand	Vector NAND
EVX	4	10000585			621	SP	evmhossianw	Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	10000587			620	SP	evmhossfanw	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	10000588			613	SP	evmheumianw	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000589			609	SP	evmhesmianw	Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words
VX	4	10000600			211	V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	4	10000604			268	V	mfvscr	Move From Vector Status and Control Register
VX	4	10000608			225	V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	4	10000640			210	V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	4	10000644			268	V	mtvscr	Move To Vector Status and Control Register
VX	4	10000648			224	V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	4	10000680			211	V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	4	10000682			258	V.Crypto	vshasigmaw	Vector SHA-256 Sigma Word
VX	4	10000684			238	V	veqv	Vector Equivalence
VX	4	10000688			223	V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	4	10000700			208	V	vsubsbs	Vector Subtract Signed Byte Saturate
VX	4	10000702			263	V	vclzb	Vector Count Leading Zeros Byte
VX	4	10000703			264	V	vpopcntb	Vector Population Count Byte
VX	4	10000708			224	V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	4	10000740			208	V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	4	10000742			263	V	vclzh	Vector Count Leading Zeros Halfword
VX	4	10000743			264	V	vpopcnth	Vector Population Count Halfword
VX	4	10000780			209	V	vsubsws	Vector Subtract Signed Word Saturate
VX	4	10000782			263	V	vclzw	Vector Count Leading Zeros Word
VX	4	10000783			264	V	vpopcntw	Vector Population Count Word
VX	4	10000788			223	V	vsumsws	Vector Sum across Signed Word Saturate
VX	4	1000000A			244	V	vaddfp	Vector Add Single-Precision
VX	4	1000000C			194	V	vmrghb	Vector Merge High Byte
VX	4	1000000E			190	V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VA	4	1000002A			199	V	vsel	Vector Select
VA	4	1000002B			198	V	vperm	Vector Permute
VA	4	1000002C			200	V	vsldoi	Vector Shift Left Double by Octet Immediate
VA	4	1000002D			261	V.RAID	vpermxor	Vector Permute and Exclusive-OR
VA	4	1000002E			245	V	vmaddfp	Vector Multiply-Add Single-Precision
VA	4	1000002F			245	V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VA	4	1000003C			206	V	vaddeuqm	Vector Add Extended Unsigned Quadword Modulo
VA	4	1000003D			206	V	vaddecuq	Vector Add Extended & write Carry Unsigned Quadword

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VA	4	1000003E			212	V	vsubeuqm	Vector Subtract Extended Unsigned Quadword Modulo
VA	4	1000003F			212	V	vsubecuq	Vector Subtract Extended & write Carry Unsigned Quadword
VX	4	1000004A			244	V	vsubfp	Vector Subtract Single-Precision
VX	4	1000004C			194	V	vmrghh	Vector Merge High Halfword
VX	4	1000004E			191	V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
XO	4	1000005C			683	LMA	nmachhw[.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	1000005C			683	LMA	nmachhwo[.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed & record OV
VX	4	1000008C			195	V	vmrghw	Vector Merge High Word
VX	4	1000008E			191	V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
VX	4	100000C0			203	V	vaddudm	Vector Add Unsigned Doubleword Modulo
VX	4	100000C2			228	V	vmaxud	Vector Maximum Unsigned Doubleword
VX	4	100000C4			240	V	vrld	Vector Rotate Left Doubleword
VC	4	100000C6			252	V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VC	4	100000C7			233	V	vcmpequd[.]	Vector Compare Equal To Unsigned Doubleword
VX	4	100000CE			191	V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
XO	4	100000D8			677	LMA	machhws[.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	100000D8			677	LMA	machhwso[.]	Multiply Accumulate High Halfword to Word Saturate Signed & record OV
XO	4	100000DC			683	LMA	nmachhws[.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	100000DC			683	LMA	nmachhwso[.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed & record OV
VX	4	1000010A			255	V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	4	1000010C			194	V	vmrglb	Vector Merge Low Byte
VX	4	1000010E			189	V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	4	1000014A			255	V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VX	4	1000014C			194	V	vmrglh	Vector Merge Low Halfword
VX	4	1000014E			190	V	vpkswus	Vector Pack Signed Word Unsigned Saturate
XO	4	1000015C			682	LMA	nmacchw[.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	1000015C			682	LMA	nmacchwo[.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed & record OV
VX	4	1000018A			254	V	vexptefp	Vector 2 Raised to the Exponent Estimate Single-Precision
VX	4	1000018C			195	V	vmrglw	Vector Merge Low Word
VX	4	1000018E			188	V	vpkshss	Vector Pack Signed Halfword Signed Saturate
VX	4	100001C2			228	V	vmaxsd	Vector Maximum Signed Doubleword
VX	4	100001C4			200	V	vsl	Vector Shift Left
VC	4	100001C6			252	V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VX	4	100001CA			254	V	vlogefp	Vector Log Base 2 Estimate Single-Precision
VX	4	100001CE			189	V	vpkswss	Vector Pack Signed Word Signed Saturate
XO	4	100001D8			675	LMA	macchws[.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	100001D8			675	LMA	macchwso[.]	Multiply Accumulate Cross Halfword to Word Saturate Signed & record OV

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	4	100001DC			682	LMA	nmacchws[.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	100001DC			682	LMA	nmacchwso[.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed & record OV
EVX	4	1000020A			599	SP	evextsb	Vector Extend Sign Byte
VX	4	1000020A			249	V	vrfin	Vector Round to Single-Precision Integer Nearest
EVX	4	1000020B			599	SP	evextsh	Vector Extend Sign Half Word
EVX	4	1000020C			633	SP	evrndw	Vector Round Word
VX	4	1000020C			197	V	vspltb	Vector Splat Byte
EVX	4	1000020D			598	SP	evcntlzw	Vector Count Leading Zeros Word
EVX	4	1000020E			598	SP	evcntlsw	Vector Count Leading Signed Bits Word
VX	4	1000020E			193	V	vupkhsb	Vector Unpack High Signed Byte
EVX	4	1000020F			594	SP	brinc	Bit Reversed Increment
EVX	4	1000021B			632	SP	evorc	Vector OR with Complement
EVX	4	1000021E			631	SP	evnand	Vector NAND
EVX	4	1000022A			633	SP	evrlwi	Vector Rotate Left Word Immediate
EVX	4	1000022B			634	SP	evsplatfi	Vector Splat Fractional Immediate
EVX	4	1000022C			605	SP	evmergehi	Vector Merge High
EVX	4	1000022D			605	SP	evmergelo	Vector Merge Low
EVX	4	1000022E			606	SP	evmergehilo	Vector Merge High/Low
EVX	4	1000022F			606	SP	evmergelohi	Vector Merge Low/High
VX	4	1000024A			249	V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	4	1000024C			197	V	vsplth	Vector Splat Halfword
VX	4	1000024E			193	V	vupkhsh	Vector Unpack High Signed Halfword
VX	4	1000028A			249	V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
EVX	4	1000028C			647	SP.FV	evfscmpgt	Vector Floating-Point Compare Greater Than
VX	4	1000028C			197	V	vspltw	Vector Splat Word
EVX	4	1000028D			647	SP.FV	evfscmplt	Vector Floating-Point Compare Less Than
EVX	4	1000028E			648	SP.FV	evfscmpeq	Vector Floating-Point Compare Equal
VX	4	1000028E			193	V	vupklsb	Vector Unpack Low Signed Byte
EVX	4	1000029A			651	SP.FV	evfsctsiz	Vector Convert Floating-Point to Signed Integer with Round toward Zero
EVX	4	1000029C			648	SP.FV	evfststgt	Vector Floating-Point Test Greater Than
EVX	4	1000029D			649	SP.FV	evfststlt	Vector Floating-Point Test Less Than
EVX	4	1000029E			649	SP.FV	evfststeq	Vector Floating-Point Test Equal
EVX	4	100002C0			654	SP.FS	efsadd	Floating-Point Add
EVX	4	100002C1			654	SP.FS	efssub	Floating-Point Subtract
VX	4	100002C2			230	V	vminud	Vector Minimum Unsigned Doubleword
EVX	4	100002C4			653	SP.FS	efsabs	Floating-Point Absolute Value
VX	4	100002C4			201	V	vsr	Vector Shift Right
EVX	4	100002C5			653	SP.FS	efsnabs	Floating-Point Negative Absolute Value
EVX	4	100002C6			653	SP.FS	efsneg	Floating-Point Negate
VC	4	100002C6			253	V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VC	4	100002C7			236	V	vcmpgtud[.]	Vector Compare Greater Than Unsigned Doubleword
EVX	4	100002C8			654	SP.FS	efsmul	Floating-Point Multiply
EVX	4	100002C9			654	SP.FS	efsdiv	Floating-Point Divide
VX	4	100002CA			250	V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
EVX	4	100002CC			655	SP.FS	efscmpgt	Floating-Point Compare Greater Than
EVX	4	100002CD			655	SP.FS	efscmplt	Floating-Point Compare Less Than

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100002CE			656	SP.FS	efscmpeq	Floating-Point Compare Equal
VX	4	100002CE			193	V	vupklsh	Vector Unpack Low Signed Halfword
EVX	4	100002CF			667	SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	4	100002D0			658	SP.FS	efscfui	Convert Floating-Point from Unsigned Integer
EVX	4	100002D1			658	SP.FS	efscfsi	Convert Floating-Point from Signed Integer
EVX	4	100002D2			658	SP.FS	efscfuf	Convert Floating-Point from Unsigned Fraction
EVX	4	100002D3			658	SP.FS	efscfsf	Convert Floating-Point from Signed Fraction
EVX	4	100002D4			658	SP.FS	efsctui	Convert Floating-Point to Unsigned Integer
EVX	4	100002D5			658	SP.FS	efsctsi	Convert Floating-Point to Signed Integer
EVX	4	100002D6			659	SP.FS	efsctuf	Convert Floating-Point to Unsigned Fraction
EVX	4	100002D7			659	SP.FS	efsctsf	Convert Floating-Point to Signed Fraction
EVX	4	100002D8			659	SP.FS	efsctuiz	Convert Floating-Point to Unsigned Integer with Round toward Zero
EVX	4	100002DA			659	SP.FS	efsctsiz	Convert Floating-Point to Signed Integer with Round toward Zero
EVX	4	100002DC			656	SP.FS	efststgt	Floating-Point Test Greater Than
EVX	4	100002DD			657	SP.FS	efststlt	Floating-Point Test Less Than
EVX	4	100002DE			657	SP.FS	efststeq	Floating-Point Test Equal
EVX	4	100002E0			661	SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	4	100002E1			661	SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	4	100002E2			664	SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	4	100002E3			664	SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	4	100002E4			660	SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	4	100002E5			660	SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	4	100002E6			660	SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	4	100002E8			661	SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	4	100002E9			661	SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	4	100002EA			665	SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	4	100002EB			665	SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	4	100002EC			662	SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	4	100002ED			662	SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	4	100002EE			662	SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	4	100002EF			666	SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	4	100002F0			663	SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	4	100002F1			663	SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	4	100002F2			664	SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	4	100002F3			664	SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	4	100002F4			664	SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100002F5			664	SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	4	100002F6			666	SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	4	100002F7			666	SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	4	100002F8			666	SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	4	100002FA			666	SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	4	100002FC			662	SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	4	100002FD			663	SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	4	100002FE			663	SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
VX	4	1000030A			248	V	vcfux	Vector Convert From Unsigned Fixed-Point Word To Single-Precision
EVX	4	1000030C			602	SP	evlhhousplatx	Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed
VX	4	1000030C			198	V	vspltisb	Vector Splat Immediate Signed Byte
EVX	4	1000030D			602	SP	evlhhousplat	Vector Load Half Word into Half Word Odd Unsigned and Splat
EVX	4	1000030E			602	SP	evlhhossplatx	Vector Load Half Word into Half Word Odd Signed and Splat Indexed
VX	4	1000030E			187	V	vpkpx	Vector Pack Pixel
EVX	4	1000030F			602	SP	evlhhossplat	Vector Load Half Word into Half Word Odd Signed and Splat
EVX	4	1000031C			604	SP	evlwhsplatx	Vector Load Word into Two Half Words and Splat Indexed
EVX	4	1000031D			604	SP	evlwhsplat	Vector Load Word into Two Half Words and Splat
EVX	4	1000033C			638	SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	4	1000033D			638	SP	evstwwo	Vector Store Word of Word from Odd
VX	4	1000034A			248	V	vcfsx	Vector Convert From Signed Fixed-Point Word To Single-Precision
VX	4	1000034C			198	V	vspltish	Vector Splat Immediate Signed Halfword
VX	4	1000034E			190	V	vupkhpx	Vector Unpack High Pixel
XO	4	1000035C			684	LMA	nmaclhw[.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	1000035C			684	LMA	nmaclhwo[.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed & record OV
VX	4	1000038A			247	V	vctuxs	Vector Convert From Single-Precision To Unsigned Fixed-Point Word Saturate
VX	4	1000038C			198	V	vspltisw	Vector Splat Immediate Signed Word
X	4	100003C2			230	V	vminsd	Vector Minimum Signed Doubleword
VX	4	100003C4			243	V	vsrad	Vector Shift Right Algebraic Doubleword
VC	4	100003C6			251	V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VC	4	100003C7			234	V	vcmpgtsd[.]	Vector Compare Greater Than Signed Doubleword
VX	4	100003CA			247	V	vctsxs	Vector Convert From Single-Precision To Signed Fixed-Point Word Saturate
VX	4	100003CE			192	V	vupklpx	Vector Unpack Low Pixel
XO	4	100003D8			679	LMA	maclhws[.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	100003D8			679	LMA	maclhwso[.]	Multiply Accumulate Low Halfword to Word Saturate Signed & record OV
XO	4	100003DC			684	LMA	nmaclhws[.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	4	100003DC			684	LMA	nmaclhwso[.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed & record OV
VX	4	1000040A			246	V	vmaxfp	Vector Maximum Single-Precision
EVX	4	1000040B			608	SP	evmhesmf	Vector Multiply Half Words, Even, Signed, Modulo, Fractional
EVX	4	1000040C			621	SP	evmhoumi	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer
VX	4	1000040C			200	V	vslo	Vector Shift Left by Octet
EVX	4	1000040D			617	SP	evmhosmi	Vector Multiply Half Words, Odd, Signed, Modulo, Integer
EVX	4	1000040F			616	SP	evmhosmf	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional
EVX	4	1000042B			608	SP	evmhesmfa	Vector Multiply Half Words, Even, Signed, Modulo, Fractional to Accumulator
EVX	4	1000042C			621	SP	evmhoumia	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	4	1000042D			617	SP	evmhosmia	Vector Multiply Half Words, Odd, Signed, Modulo, Integer to Accumulator
EVX	4	1000042F			616	SP	evmhosmfa	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional to Accumulator
VX	4	1000044A			246	V	vminfp	Vector Minimum Single-Precision
EVX	4	1000044C			624	SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
VX	4	1000044C			201	V	vsro	Vector Shift Right by Octet
EVX	4	1000044D			623	SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
VX	4	1000044E			190	V	vpkudum	Vector Pack Unsigned Doubleword Unsigned Modulo
EVX	4	1000044F			623	SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	4	1000045B			627	SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	4	1000046C			624	SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	4	1000046D			623	SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	4	1000046F			623	SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	4	1000047B			627	SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
EVX	4	100004C0			595	SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
VX	4	100004C0			210	V	vsubudm	Vector Subtract Unsigned Doubleword Modulo
EVX	4	100004C1			595	SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	4	100004C2			639	SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	4	100004C3			638	SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	4	100004C4			623	SP	evmra	Initialize Accumulator
VX	4	100004C4			239	V	vxor	Vector Logical XOR
EVX	4	100004C6			598	SP	evdivws	Vector Divide Word Signed
EVX	4	100004C7			599	SP	evdivwu	Vector Divide Word Unsigned
EVX	4	100004C8			595	SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
VX	4	100004C8			259	V	vpmsumd	Vector Polynomial Multiply-Sum Doubleword
EVX	4	100004C9			594	SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100004CA			639	SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	4	100004CB			638	SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
VX	4	100004CE			190	V	vpkudus	Vector Pack Unsigned Doubleword Unsigned Saturate
EVX	4	1000050B			608	SP	evmhesmfaaw	Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1000050C			622	SP	evmhoumiaaw	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words
VX	4	1000050C			262	V	vgbbd	Vector Gather Bits by Byte by Doubleword
EVX	4	1000050D			618	SP	evmhosmiaaw	Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1000050F			617	SP	evmhosmfaaw	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1000052B			606	SP	evmhegsmfaa	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	1000052C			616	SP	evmhogumiaa	Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	1000052D			615	SP	evmhogsmiaa	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer, and Accumulate
EVX	4	1000052F			615	SP	evmhogsmfaa	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
VX	4	1000054C			265	V	vbpermq	Vector Bit Permute Quadword
VX	4	1000054C			265	V	vbpermq	Vector Bit Permute Quadword
VX	4	1000054E			188	V	vpksdus	Vector Pack Signed Doubleword Unsigned Saturate
EVX	4	1000055B			628	SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	4	1000058B			608	SP	evmhesmfanw	Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1000058C			618	SP	evmhoumianw	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	1000058D			617	SP	evmhosmianw	Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	1000058F			617	SP	evmhosmfanw	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	100005A8			607	SP	evmhegumian	Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	100005A9			607	SP	evmhegsmian	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	100005AB			606	SP	evmhegsmfan	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	100005AC			616	SP	evmhogumian	Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	100005AD			615	SP	evmhogsmian	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100005AF			615	SP	evmhogsmfan	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	100005C0			627	SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	4	100005C1			625	SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
VX	4	100005C4			241	V	vsld	Vector Shift Left Doubleword
EVX	4	100005C8			626	SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words
VX	4	100005C8			257	V.Crypto	vsbox	Vector AES S-Box
EVX	4	100005C9			625	SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
VX	4	100005CE			187	V	vpksdss	Vector Pack Signed Doubleword Signed Saturate
EVX	4	100005D3			630	SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	4	100005D8			631	SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	100005D9			628	SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	4	100005DB			628	SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
VX	4	1000064E			193	V	vupkhsw	Vector Unpack High Signed Word
VX	4	1000068C			196	VSX	vmrgow	Vector Merge Odd Word
VX	4	100006C2			258	V.Crypto	vshasigmad	Vector SHA-512 Sigma Doubleword
VX	4	100006C4			242	V	vsrd	Vector Shift Right Doubleword
VX	4	100006CE			193	V	vupklsw	Vector Unpack Low Signed Word
VX	4	1000078C			196	VSX	vmrgew	Vector Merge Even Word
VX	4	100007C2			263	V	vclzd	Vector Count Leading Zeros Doubleword
VX	4	100007C3			264	V	vpopcntd	Vector Population Count Doubleword
D	7	1C000000			71	B	mulli	Multiply Low Immediate
D	8	20000000	SR		68	B	subfic	Subtract From Immediate Carrying
D	10	28000000			79	B	cmpli	Compare Logical Immediate
D	11	2C000000			78	B	cmpi	Compare Immediate
D	12	30000000	SR		67	B	addic	Add Immediate Carrying
D	13	34000000	SR		67	B	addic.	Add Immediate Carrying & record CR0
D	14	38000000			66	B	addi	Add Immediate
D	15	3C000000			66	B	addis	Add Immediate Shifted
B	16	40000000	CT		38	B	bc[l][a]	Branch Conditional
SC	17	44000002			43 863 1040	B	sc	System Call
I	18	48000000			38	B	b[l][a]	Branch
XL	19	4C000000			42	B	mcrf	Move Condition Register Field
XL	19	4C000020	CT		39	B	bclr[l]	Branch Conditional to Link Register
XL	19	4C000024		P	864	S	rfid	Return from Interrupt Doubleword
XL	19	4C000042			42	B	crnor	Condition Register NOR
XL	19	4C00004C		P	1042	E	rfmci	Return From Machine Check Interrupt
X	19	4C00004E		P	1042	E.ED	rfdi	Return From Debug Interrupt
XL	19	4C000064		P	1041	E	rfi	Return From Interrupt
XL	19	4C000066		P	1041	E	rfci	Return From Critical Interrupt
XL	19	4C0000CC		P	1043	E.HV	rfgi	Return From Guest Interrupt
XL	19	4C000102			42	B	crandc	Condition Register AND with Complement
XL	19	4C000124			820	S	rfebb	Return from Event Based Branch
XL	19	4C00012C			776	B	isync	Instruction Synchronize

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XL	19	4C000182			41	B	crxor	Condition Register XOR
XFX	19	4C00018C			1228	E	dnh	Debugger Notify Halt
XL	19	4C0001C2			41	B	crnand	Condition Register NAND
XL	19	4C000202			41	B	crand	Condition Register AND
XL	19	4C000224		H	865	S	hrfid	Return From Interrupt Doubleword Hypervisor
XL	19	4C000242			42	B	creqv	Condition Register Equivalent
XL	19	4C000324		H	867	S	doze	Doze
XL	19	4C000342			42	B	crorc	Condition Register OR with Complement
XL	19	4C000364		H	867	S	nap	Nap
XL	19	4C000382			41	B	cror	Condition Register OR
XL	19	4C0003A4		H	868	S	sleep	Sleep
XL	19	4C0003E4		H	868	S	rvwinkle	Rip Van Winkle
XL	19	4C000420	CT		39	B	bcctr[l]	Branch Conditional to Count Register
X	19	4C000460			40	B	bctar[l]	Branch Conditional to Branch Target Address Register
M	20	50000000	SR		93	B	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21	54000000	SR		91	B	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23	5C000000	SR		92	B	rlwnm[.]	Rotate Left Word then AND with Mask
D	24	60000000			82	B	ori	OR Immediate
D	25	64000000			83	B	oris	OR Immediate Shifted
X	26	68000000				B	xnop	Executed No Operation
D	26	68000000			83	B	xori	XOR Immediate
D	27	6C000000			83	B	xoris	XOR Immediate Shifted
D	28	70000000	SR		82	B	andi.	AND Immediate & record CR0
D	29	74000000	SR		82	B	andis.	AND Immediate Shifted & record CR0
MD	30	78000000	SR		94	64	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	78000004	SR		94	64	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	78000008	SR		95	64	rldic[.]	Rotate Left Doubleword Immediate then Clear
MDS	30	78000010	SR		95	64	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	78000012	SR		96	64	rldcr[.]	Rotate Left Doubleword then Clear Right
MD	30	7800000C	SR		96	64	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
X	31	7C000000			78	B	cmp	Compare
X	31	7C000008			80	B	tw	Trap Word
X	31	7C00000C			186	V	lvsl	Load Vector for Shift Left
X	31	7C00000E			184	V	lvebx	Load Vector Element Byte Indexed
XO	31	7C000010	SR		68	B	subfc[.]	Subtract From Carrying
XO	31	7C000012	SR		64	64	mulhdu[.]	Multiply High Doubleword Unsigned
XO	31	7C000014	SR		68	B	addc[.]	Add Carrying
XO	31	7C000016	SR		71	B	mulhwu[.]	Multiply High Word Unsigned
XX1	31	7C000018			394	VSX	lxsiwzx	Load VSX Scalar as Integer Word and Zero Indexed
A	31	7C00001E			81	B	isel	Integer Select
X	31	7C000024		P	1134	E	tlbilx	TLB Invalidate Local Indexed
XFX	31	7C000026			108	B	mfcr	Move From Condition Register
X	31	7C000028			777	B	lwarx	Load Word and Reserve Indexed
X	31	7C00002A			53	64	ldx	Load Doubleword Indexed
X	31	7C00002C			762	E	icbt	Instruction Cache Block Touch
X	31	7C00002E			51	B	lwzx	Load Word and Zero Indexed
X	31	7C000030	SR		97	B	slw[.]	Shift Left Word
X	31	7C000034	SR		85	B	cntlzw[.]	Count Leading Zeros Word
X	31	7C000036	SR		99	64	sld[.]	Shift Left Doubleword

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C000038	SR		84	B	and[.]	AND
X	31	7C00003A		P	1060	E.PD;64	ldepx	Load Doubleword by External PID Indexed
X	31	7C00003E		P	1060	E.PD	lwepx	Load Word and Zero by External PID Indexed
X	31	7C000040			79	B	cmpl	Compare Logical
X	31	7C00004C			186	V	lvsr	Load Vector for Shift Right
X	31	7C00004E			181	V	lvehx	Load Vector Element Halfword Indexed
XO	31	7C000050	SR		67	B	subf[.]	Subtract From
XX1	31	7C000066			109	VSX	mfvsrd	Move From VSR Doubleword
X	31	7C000068			777	B	lbarx	Load Byte And Reserve Indexed
X	31	7C00006A			53	64	ldux	Load Doubleword with Update Indexed
X	31	7C00006C			773	B	dcbst	Data Cache Block Store
X	31	7C00006E			51	B	lwzux	Load Word and Zero with Update Indexed
X	31	7C000074	SR		89	64	cntlzd[.]	Count Leading Zeros Doubleword
X	31	7C000078	SR		85	B	andc[.]	AND with Complement
X	31	7C00007C			791	WT	wait	Wait for Interrupt
X	31	7C00007E		P	1063	E.PD	dcbstep	Data Cache Block Store by External PID
X	31	7C000088			81	64	td	Trap Doubleword
X	31	7C00008E			181	V	lvewx	Load Vector Element Word Indexed
XO	31	7C000092	SR		75	64	mulhd[.]	Multiply High Doubleword
XO	31	7C000094		H	101	BCDA	addg6s	Add and Generate Sixes
XO	31	7C000096	SR		71	B	mulhw[.]	Multiply High Word
XX1	31	7C000098			393	VSX	lxsiwax	Load VSX Scalar as Integer Word Algebraic Indexed
X	31	7C00009C			673	LMV	dlmzb[.]	Determine Leftmost Zero Byte
X	31	7C0000A6		P	888 1055	S E	mfmsr	Move From Machine State Register
X	31	7C0000A8			782	64	ldarx	Load Doubleword And Reserve Indexed
X	31	7C0000AC			773	B	dcbf	Data Cache Block Flush
X	31	7C0000AE			49	B	lbzx	Load Byte and Zero Indexed
X	31	7C0000BE		P	1059	E.PD	lbepx	Load Byte and Zero by External PID Indexed
X	31	7C0000CE			182	V	lvx	Load Vector Indexed
XO	31	7C0000D0	SR		70	B	neg[.]	Negate
XX1	31	7C0000E6			109	VSX	mfvsrwz	Move From VSR Word and Zero
X	31	7C0000E8			778	B	lharx	Load Halfword And Reserve Indexed Xform
X	31	7C0000EE			48	B	lbzux	Load Byte and Zero with Update Indexed
X	31	7C0000F4			87	B	popcntb	Population Count Byte-wise
X	31	7C0000F8	SR		85	B	nor[.]	NOR
X	31	7C0000FE		P	1064	E.PD	dcbfep	Data Cache Block Flush by External PID
X	31	7C000106		P	1056	E	wrtee	Write External Enable
X	31	7C00010C		M	1122	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	31	7C00010E			184	V	stvebx	Store Vector Element Byte Indexed
XO	31	7C000110	SR		69	B	subfe[.]	Subtract From Extended
XO	31	7C000114	SR		69	B	adde[.]	Add Extended
XX1	31	7C000118			394	VSX	stxsiwx	Store VSX Scalar as Integer Word Indexed
X	31	7C00011C		P	1007	S.PC	msgsndp	Message Send Privileged
XFX	31	7C000120			107	B	mtcrf	Move To Condition Register Fields
X	31	7C000124		P	884 1055	S E	mtmsr	Move To Machine State Register
X	31	7C000126			108	S	mtsle	Move To Split Little Endian
X	31	7C00012A			57	64	stdx	Store Doubleword Indexed
X	31	7C00012D			781	B	stwcx.	Store Word Conditional Indexed & record CR0
X	31	7C00012E			56	B	stwx	Store Word Indexed
X	31	7C000134			88	B	prtyw	Parity Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C00013A		P	1062	E.PD;64	stdepx	Store Doubleword by External PID Indexed
X	31	7C00013E		P	1062	E.PD	stwepx	Store Word by External PID Indexed
X	31	7C000146		P	1057	E	wrteei	Write External Enable Immediate
X	31	7C00014C		M	1122	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	31	7C00014E			184	V	stvehx	Store Vector Element Halfword Indexed
X	31	7C00015C		P	1007	S.PC	msgclrp	Message Clear Privileged
X	31	7C000164		P	886	S	mtmsrd	Move To Machine State Register Doubleword
XX2	31	7C000166			110	VSX	mtvsrd	Move To VSR Doubleword
X	31	7C00016A			57	64	stdux	Store Doubleword with Update Indexed
X	31	7C00016D			785	LSQ	stqcx.	Store Quadword Conditional Indexed and record CR0
X	31	7C00016E			56	B	stwux	Store Word with Update Indexed
X	31	7C000174			88	64	prtyd	Parity Doubleword
X	31	7C00018D			1121	ECL	icblq.	Instruction Cache Block Lock Query
X	31	7C00018E			185	V	stvewx	Store Vector Element Word Indexed
XO	31	7C000190	SR		70	B	subfze[.]	Subtract From Zero Extended
XO	31	7C000194	SR		70	B	addze[.]	Add to Zero Extended
X	31	7C00019C		H	1006 1233	S.PC E.PC	msgsnd	Message Send
X	31	7C0001A4	32	P	926	S	mtsr	Move To Segment Register
XX2	31	7C0001A6			110	VSX	mtvsrwa	Move To VSR Word Algebraic
X	31	7C0001AD			782	64	stdcx.	Store Doubleword Conditional Indexed & record CR0
X	31	7C0001AE			54	B	stbx	Store Byte Indexed
X	31	7C0001BE		P	1061	E.PD	stbepx	Store Byte by External PID Indexed
X	31	7C0001CC		M	1124	ECL	icblc	Instruction Cache Block Lock Clear
X	31	7C0001CE			182	V	stvx	Store Vector Indexed
XO	31	7C0001D0	SR		69	B	subfme[.]	Subtract From Minus One Extended
XO	31	7C0001D2	SR		64	64	mulld[.]	Multiply Low Doubleword
XO	31	7C0001D4	SR		69	B	addme[.]	Add to Minus One Extended
XO	31	7C0001D6	SR		71	B	mullw[.]	Multiply Low Word
X	31	7C0001DC		H	1006 1233	S.PC E.PC	msgclr	Message Clear
X	31	7C0001E4	32	P	926	S	mtsrin	Move To Segment Register Indirect
XX2	31	7C0001E6			111	VSX	mtvsrwz	Move To VSR Word and Zero
X	31	7C0001EC			771	B	dcbtst	Data Cache Block Touch for Store
X	31	7C0001EE			54	B	stbux	Store Byte with Update Indexed
X	31	7C0001F8			90	64	bpermd	Bit Permute Doubleword
X	31	7C0001FE		P	1066	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID
X	31	7C000206		P	1055	E.DC	mfdcrx	Move From Device Control Register Indexed
X	31	7C00020E		P	1070	E.PD	lvepxl	Load Vector by External PID Indexed Last
XO	31	7C000214	SR		67	B	add[.]	Add
XL	31	7C00021C			1043	E.HV	ehpriv	Embedded Hypervisor Privilege
X	31	7C000224	64	P	931	S	tlbiel	TLB Invalidate Entry Local
X	31	7C000228			784	LSQ	lqarx	Load Quadword And Reserve Indexed
X	31	7C00022C			770	B	dcbt	Data Cache Block Touch
X	31	7C00022E			49	B	lhzx	Load Halfword and Zero Indexed
X	31	7C000234		H	101	BCDA	cdtbcd	Convert Declets To Binary Coded Decimal
X	31	7C000238	SR		85	B	eqv[.]	Equivalent
X	31	7C00023E		P	1059	E.PD	lhepx	Load Halfword and Zero by External PID Indexed
X	31	7C000246			113	E.DC	mfdcrux	Move From Device Control Register User Mode Indexed
X	31	7C00024E		P	1070	E.PD	lvepx	Load Vector by External PID Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XFX	31	7C00025C			44	S	mfbhrbe	Move From Branch History Rolling Buffer
X	31	7C000264	64	H	928	S	tlbie	TLB Invalidate Entry
X	31	7C00026C			826	EC	eciwx	External Control In Word Indexed
X	31	7C00026E			49	B	lhzux	Load Halfword and Zero with Update Indexed
X	31	7C000274		H	101	BCDA	cbcdtd	Convert Binary Coded Decimal To Declets
X	31	7C000278	SR		84	B	xor[.]	XOR
X	31	7C00027E		P	1063	E.PD	dcbtep	Data Cache Block Touch by External PID
XFX	31	7C000286		P	1055	E.DC	mfdcr	Move From Device Control Register
X	31	7C00028C		P	1242	E.CD	dcread	Data Cache Read
XX1	31	7C000298			395	VSX	lxvdsx	Load VSR Vector Doubleword & Splat Indexed
XFX	31	7C00029C		O	1257	E.PM	mfpmr	Move from Performance Monitor Register
XFX	31	7C0002A6		O	106 813 885 1054	B	mfspr	Move From Special Purpose Register
X	31	7C0002AA			52	64	lwax	Load Word Algebraic Indexed
X	31	7C0002AE			50	B	lhax	Load Halfword Algebraic Indexed
X	31	7C0002CE			182	V	lvxl	Load Vector Indexed Last
X	31	7C0002E4		H	933	S	tlbia	TLB Invalidate All
XFX	31	7C0002E6			813	S.out	mftb	Move From Time Base
X	31	7C0002EA			52	64	lwaux	Load Word Algebraic with Update Indexed
X	31	7C0002EE			50	B	lhaux	Load Halfword Algebraic with Update Indexed
X	31	7C0002F4			87	B	popcntw	Population Count Words
X	31	7C000306		P	1054	E.DC	mtdcrx	Move To Device Control Register Indexed
X	31	7C00030C		M	1123	ECL	dcblc	Data Cache Block Lock Clear
XO	31	7C000312	SR		77	64	divdeu[.]	Divide Doubleword Extended Unsigned
XO	31	7C000316	SR		73	B	divweu[.]	Divide Word Extended Unsigned
X	31	7C000324		P	921	S	slbmte	SLB Move To Entry
X	31	7C00032E			55	B	sthx	Store Halfword Indexed
X	31	7C000338	SR		85	B	orc[.]	OR with Complement
X	31	7C00033E		P	1061	E.PD	sthepx	Store Halfword by External PID Indexed
X	31	7C000346			113	E.DC	mtdcrux	Move To Device Control Register User Mode Indexed
X	31	7C00034D			1121	ECL	dcblq.	Data Cache Block Lock Query
XO	31	7C000352	SR		77	64	divde[.]	Divide Doubleword Extended
XO	31	7C000356	SR		73	B	divwe[.]	Divide Word Extended
X	31	7C00035C			44	S	clrbhrb	Clear BHRB
X	31	7C000364		P	919	S	slbie	SLB Invalidate Entry
X	31	7C00036C			826	EC	ecowx	External Control Out Word Indexed
X	31	7C00036E			55	B	sthux	Store Halfword with Update Indexed
X	31	7C000378	SR		84	B	or[.]	OR
XFX	31	7C000386		P	1054	E.DC	mtdcr	Move To Device Control Register
X	31	7C00038C		P	1239	E.CI	dci	Data Cache Invalidate
XO	31	7C000392	SR		76	64	divdu[.]	Divide Doubleword Unsigned
XO	31	7C000396	SR		72	B	divwu[.]	Divide Word Unsigned
XFX	31	7C00039C		O	1257	E.PM	mtpmr	Move To Performance Monitor Register
XFX	31	7C0003A6		O	104 884 1053	B	mtspr	Move To Special Purpose Register
X	31	7C0003AC		P	1118	E	dcbi	Data Cache Block Invalidate
X	31	7C0003B8	SR		84	B	nand[.]	NAND
X	31	7C0003C6			824	DS	dsn	Decorated Storage Notify
X	31	7C0003CC		P	1242	E.CD	dcread	Data Cache Read
X	31	7C0003CC		M	1123	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	31	7C0003CE			185	V	stvxl	Store Vector Indexed Last

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	31	7C0003D2	SR		76	64	divd[.]	Divide Doubleword
XO	31	7C0003D6	SR		72	B	divw[.]	Divide Word
X	31	7C0003E4		P	920	S	slbia	SLB Invalidate All
X	31	7C0003F4			89	64	popcntd	Population Count Doubleword
X	31	7C0003F8			86	B	cmpb	Compare Byte
X	31	7C000400			113	E	mcrxr	Move to Condition Register from XER
X	31	7C000406			822	DS	lbdx	Load Byte with Decoration Indexed
XO	31	7C000410	SR		68	B	subfco[.]	Subtract From Carrying & record OV
XO	31	7C000414	SR		68	B	addco[.]	Add Carrying & record OV
XX1	31	7C000418			394	VSX	lxsspx	Load VSX Scalar Single-Precision Indexed
X	31	7C000428			60	64	ldbrx	Load Doubleword Byte-Reverse Indexed
X	31	7C00042A			63	MA	lswx	Load String Word Indexed
X	31	7C00042C			59	B	lwbrx	Load Word Byte-Reverse Indexed
X	31	7C00042E			138	FP	lfsx	Load Floating-Point Single Indexed
X	31	7C000430	SR		97	B	srw[.]	Shift Right Word
X	31	7C000436	SR		99	64	srd[.]	Shift Right Doubleword
X	31	7C000446			822	DS	lhdx	Load Halfword with Decoration Indexed
XO	31	7C000450	SR		67	B	subfo[.]	Subtract From & record OV
X	31	7C00046C		H PH	934 1141	S E	tlbsync	TLB Synchronize
X	31	7C00046E			138	FP	lfsux	Load Floating-Point Single with Update Indexed
X	31	7C000486			822	DS	lwdx	Load Word with Decoration Indexed
XX1	31	7C000498			393	VSX	lxsdx	Load VSR Scalar Doubleword Indexed
X	31	7C0004A6	32	P	927	S	mfsr	Move From Segment Register
X	31	7C0004AA			63	MA	lswi	Load String Word Immediate
X	31	7C0004AC			786	B	sync	Synchronize
X	31	7C0004AE			135	FP	lfdx	Load Floating-Point Double Indexed
X	31	7C0004BE		P	1068	E.PD	lfdepx	Load Floating-Point Double by External PID Indexed
X	31	7C0004C6			822	DS	lddx	Load Doubleword with Decoration Indexed
XO	31	7C0004D0	SR		70	B	nego[.]	Negate & record OV
X	31	7C0004EE			135	FP	lfdux	Load Floating-Point Double with Update Indexed
X	31	7C000506			823	DS	stbdx	Store Byte with Decoration Indexed
XO	31	7C000510	SR		69	B	subfeo[.]	Subtract From Extended & record OV
XO	31	7C000514	SR		69	B	addeo[.]	Add Extended & record OV & record OV
XX1	31	7C000518			394	VSX	stxsspx	Store VSR Scalar Word Indexed
X	31	7C00051D			806	TM	tbegin.	Transaction Begin
X	31	7C000526	32	P	927	S	mfsrin	Move From Segment Register Indirect
X	31	7C000528			60	64	stdbrx	Store Doubleword Byte-Reverse Indexed
X	31	7C00052A			64	MA	stswx	Store String Word Indexed
X	31	7C00052C			59	B	stwbrx	Store Word Byte-Reverse Indexed
X	31	7C00052E			138	FP	stfsx	Store Floating-Point Single Indexed
X	31	7C000546			823	DS	sthdx	Store Halfword with Decoration Indexed
X	31	7C00055C			807	TM	tend.	Transaction End
X	31	7C00056D			779	B	stbcx.	Store Byte Conditional Indexed
X	31	7C00056E			138	FP	stfsux	Store Floating-Point Single with Update Indexed
X	31	7C000586			823	DS	stwdx	Store Word with Decoration Indexed
XO	31	7C000590	SR		70	B	subfzeo[.]	Subtract From Zero Extended & record OV
XO	31	7C000594	SR		70	B	addzeo[.]	Add to Zero Extended & record OV
XX1	31	7C000598			396	VSX	stxsdx	Store VSR Scalar Doubleword Indexed
X	31	7C00059C			811	TM	tcheck	Transaction Check
X	31	7C0005AA			64	MA	stswi	Store String Word Immediate

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C0005AD			780	B	sthcx.	Store Halfword Conditional Indexed Xform
X	31	7C0005AE			139	FP	stfdx	Store Floating-Point Double Indexed
X	31	7C0005BE		P	1068	E.PD	stfdepx	Store Floating-Point Double by External PID Indexed
X	31	7C0005C6			823	DS	stddx	Store Doubleword with Decoration Indexed
XO	31	7C0005D0	SR		69	B	subfmeo[.]	Subtract From Minus One Extended & record OV
XO	31	7C0005D2	SR		64	64	mulldo[.]	Multiply Low Doubleword & record OV
XO	31	7C0005D4	SR		69	B	addmeo[.]	Add to Minus One Extended & record OV
XO	31	7C0005D6	SR		71	B	mullwo[.]	Multiply Low Word & record OV
X	31	7C0005DC			810	TM	tsr.	Transaction Suspend or Resume
X	31	7C0005EC			770	E	dcba	Data Cache Block Allocate
X	31	7C0005EE			139	FP	stfdux	Store Floating-Point Double with Update Indexed
X	31	7C00060E		P	1071	E.PD	stvepxl	Store Vector by External PID Indexed Last
XO	31	7C000614	SR		67	B	addo[.]	Add & record OV
XX1	31	7C000618			396	VSX	lxvw4x	Load VSR Vector Word*4 Indexed
X	31	7C00061D			809	TM	tabortwc.	Transaction Abort Word Conditional
X	31	7C000624		P	1132	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	31	7C00062A		H	876	S	lwzcix	Load Word and Zero Caching Inhibited Indexed
X	31	7C00062C			59	B	lhbrx	Load Halfword Byte-Reverse Indexed
X	31	7C00062E			141	FP.out	lfdpx	Load Floating-Point Double Pair Indexed
X	31	7C000630	SR		98	B	sraw[.]	Shift Right Algebraic Word
X	31	7C000634	SR		100	64	srad[.]	Shift Right Algebraic Doubleword
EVX	31	7C00063E		P	1069	E.PD	evlddepx	Vector Load Double Word into Double Word by External PID Indexed
X	31	7C000646			822	DS	lfddx	Load Floating Doubleword with Decoration Indexed
X	31	7C00064E		P	1071	E.PD	stvepx	Store Vector by External PID Indexed
X	31	7C00065D			809	TM	tabortdc.	Transaction Abort Doubleword Conditional
X	31	7C00066A		H	876	S	lhzcix	Load Halfword and Zero Caching Inhibited Indexed
X	31	7C000670	SR		98	B	srawi[.]	Shift Right Algebraic Word Immediate
XS	31	7C000674	SR		100	64	sradi[.]	Shift Right Algebraic Doubleword Immediate
XX1	31	7C000698			395	VSX	lxvd2x	Load VSR Vector Doubleword*2 Indexed
X	31	7C00069D			809	TM	tabortwci.	Transaction Abort Word Conditional Immediate
X	31	7C0006A5		P	1138	E.TWC	tlbsrx.	TLB Search and Reserve Indexed
X	31	7C0006A6		P	922	S	slbmfev	SLB Move From Entry VSID
X	31	7C0006AA		H	876	S	lbzcix	Load Byte and Zero Caching Inhibited Indexed
X	31	7C0006AC			790	S	eieio	Enforce In-order Execution of I/O
X	31	7C0006AC			790	E	mbar	Memory Barrier
X	31	7C0006AE			136	FP	lfiwax	Load Floating-Point as Integer Word Algebraic Indexed
X	31	7C0006DD			810	TM	tabortdci.	Transaction Abort Doubleword Conditional Immediate
X	31	7C0006EA		H	876	S	ldcix	Load Doubleword Caching Inhibited Indexed
X	31	7C0006EE			136	FP	lfiwzx	Load Floating-Point as Integer Word and Zero Indexed
XO	31	7C000712	SR		77	64	divdeuo[.]	Divide Doubleword Extended Unsigned & record OV
XO	31	7C000716	SR		73	B	divweuo[.]	Divide Word Extended Unsigned & record OV
XX1	31	7C000718			398	VSX	stxvw4x	Store VSR Vector Word*4 Indexed
X	31	7C00071D			808	TM	tabort.	Transaction Abort

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C000724		P	1136	E	tlbsx	TLB Search Indexed
X	31	7C000726		P	923	S	slbmfee	SLB Move From Entry ESID
X	31	7C00072A		H	877	S	stwcix	Store Word and Zero Caching Inhibited Indexed
X	31	7C00072C			59	B	sthbrx	Store Halfword Byte-Reverse Indexed
X	31	7C00072E			141	FP.out	stfdpx	Store Floating-Point Double Pair Indexed
X	31	7C000734	SR		85	B	extsh[.]	Extend Sign Halfword
EVX	31	7C00073E		P	1069	E.PD	evstddepx	Vector Store Double of Double by External PID Indexed
X	31	7C000746			823	DS	stfddx	Store Floating Doubleword with Decoration Indexed
XO	31	7C000752	SR		77	64	divdeo[.]	Divide Doubleword Extended & record OV
XO	31	7C000756	SR		73	B	divweo[.]	Divide Word Extended & record OV
X	31	7C00075D			879	TM	treclaim.	Transaction Reclaim
X	31	7C000764		P	1139	E	tlbre	TLB Read Entry
X	31	7C00076A		H	877	S	sthcix	Store Halfword and Zero Caching Inhibited Indexed
X	31	7C000774	SR		85	B	extsb[.]	Extend Sign Byte
X	31	7C00078C		P	1239	E.CI	ici	Instruction Cache Invalidate
XO	31	7C000792	SR		76	64	divduo[.]	Divide Doubleword Unsigned & record OV
XO	31	7C000796	SR		72	B	divwuo[.]	Divide Word Unsigned & record OV
XX1	31	7C000798			398	VSX	stxvd2x	Store VSR Vector Doubleword*2 Indexed
X	31	7C0007A4		P	1141	E	tlbwe	TLB Write Entry
X	31	7C0007A7	SR	P	923	S	slbfee.	SLB Find Entry ESID
X	31	7C0007AA		H	877	S	stbcix	Store Byte Caching Inhibited Indexed
X	31	7C0007AC			762	B	icbi	Instruction Cache Block Invalidate
X	31	7C0007AE			140	FP	stfiwx	Store Floating-Point as Integer Word Indexed
X	31	7C0007B4	SR		89	64	extsw[.]	Extend Sign Word
X	31	7C0007BE		P	1067	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	31	7C0007CC		P	1243	E.CD	icread	Instruction Cache Read
XO	31	7C0007D2	SR		76	64	divdo[.]	Divide Doubleword & record OV
XO	31	7C0007D6	SR		72	B	divwo[.]	Divide Word & record OV
X	31	7C0007DD			880	TM	trechkpt.	Transaction Recheckpoint
X	31	7C0007EA		H	877	S	stdcix	Store Doubleword Caching Inhibited Indexed
X	31	7C0007EC			773	B	dcbz	Data Cache Block Zero
X	31	7C0007FE		P	1067	E.PD	dcbzep	Data Cache Block Zero by External PID
XFX	31	7C100026			112	B	mfocrf	Move From One Condition Register Field
XFX	31	7C100120			112	B	mtocrf	Move To One Condition Register Field
D	32	80000000			51	B	lwz	Load Word and Zero
D	33	84000000			51	B	lwzu	Load Word and Zero with Update
D	34	88000000			48	B	lbz	Load Byte and Zero
D	35	8C000000			48	B	lbzu	Load Byte and Zero with Update
D	36	90000000			56	B	stw	Store Word
D	37	94000000			56	B	stwu	Store Word with Update
D	38	98000000			54	B	stb	Store Byte
D	39	9C000000			54	B	stbu	Store Byte with Update
D	40	A0000000			49	B	lhz	Load Halfword and Zero
D	41	A4000000			49	B	lhzu	Load Halfword and Zero with Update
D	42	A8000000			50	B	lha	Load Halfword Algebraic
D	43	AC000000			50	B	lhau	Load Halfword Algebraic with Update
D	44	B0000000			55	B	sth	Store Halfword
D	45	B4000000			55	B	sthu	Store Halfword with Update
D	46	B8000000			61	B	lmw	Load Multiple Word
D	47	BC000000			61	B	stmw	Store Multiple Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
D	48	C0000000			138	FP	lfs	Load Floating-Point Single
D	49	C4000000			138	FP	lfsu	Load Floating-Point Single with Update
D	50	C8000000			135	FP	lfd	Load Floating-Point Double
D	51	CC000000			135	FP	lfdu	Load Floating-Point Double with Update
D	52	D0000000			138	FP	stfs	Store Floating-Point Single
D	53	D4000000			138	FP	stfsu	Store Floating-Point Single with Update
D	54	D8000000			139	FP	stfd	Store Floating-Point Double
D	55	DC000000			139	FP	stfdu	Store Floating-Point Double with Update
DQ	56	E0000000		P	58	LSQ	lq	Load Quadword
DS	57	E4000000			141	FP.out	lfdp	Load Floating-Point Double Pair
DS	58	E8000000			53	64	ld	Load Doubleword
DS	58	E8000001			53	64	ldu	Load Doubleword with Update
DS	58	E8000002			52	64	lwa	Load Word Algebraic
X	59	EC000004			289	DFP	dadd[.]	Decimal Floating Add
Z23	59	EC000006			300	DFP	dqua[.]	Decimal Quantize
A	59	EC000024			145	FP[R]	fdivs[.]	Floating Divide Single
A	59	EC000028			144	FP[R]	fsubs[.]	Floating Subtract Single
A	59	EC00002A			144	FP[R]	fadds[.]	Floating Add Single
A	59	EC00002C			146	FP[R]	fsqrts[.]	Floating Square Root Single
A	59	EC000030			146	FP[R]	fres[.]	Floating Reciprocal Estimate Single
A	59	EC000032			145	FP[R]	fmuls[.]	Floating Multiply Single
A	59	EC000034			147	FP[R].in	frsqrtes[.]	Floating Reciprocal Square Root Estimate Single
A	59	EC000038			149	FP[R]	fmsubs[.]	Floating Multiply-Subtract Single
A	59	EC00003A			149	FP[R]	fmadds[.]	Floating Multiply-Add Single
A	59	EC00003C			150	FP[R]	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	59	EC00003E			150	FP[R]	fnmadds[.]	Floating Negative Multiply-Add Single
X	59	EC000044			291	DFP	dmul[.]	Decimal Floating Multiply
Z23	59	EC000046			302	DFP	drrnd[.]	Decimal Floating Reround
Z22	59	EC000084			316	DFP	dscli[.]	Decimal Floating Shift Coefficient Left Immediate
Z23	59	EC000086			299	DFP	dquai[.]	Decimal Quantize Immediate
Z22	59	EC0000C4			316	DFP	dscri[.]	Decimal Floating Shift Coefficient Right Immediate
Z23	59	EC0000C6			305	DFP	drintx[.]	Decimal Floating Round To FP Integer With Inexact
X	59	EC000104			295	DFP	dcmpo	Decimal Floating Compare Ordered
X	59	EC000144			297	DFP	dtstex	Decimal Floating Test Exponent
Z22	59	EC000184			296	DFP	dtstdc	Decimal Floating Test Data Class
Z22	59	EC0001C4			296	DFP	dtstdg	Decimal Floating Test Data Group
Z23	59	EC0001C6			307	DFP	drintn[.]	Decimal Floating Round To FP Integer Without Inexact
X	59	EC000204			309	DFP	dctdp[.]	Decimal Floating Convert To DFP Long
X	59	EC000244			311	DFP	dctfix[.]	Decimal Floating Convert To Fixed
X	59	EC000284			313	DFP	ddedpd[.]	Decimal Floating Decode DPD To BCD
X	59	EC0002C4			314	DFP	dxex[.]	Decimal Floating Extract Exponent
X	59	EC000404			289	DFP	dsub[.]	Decimal Floating Subtract
X	59	EC000444			292	DFP	ddiv[.]	Decimal Floating Divide
X	59	EC000504			294	DFP	dcmpu	Decimal Floating Compare Unordered
X	59	EC000544			298	DFP	dtstsf	Decimal Floating Test Significance
X	59	EC000604			310	DFP	drsp[.]	Decimal Floating Round To DFP Short
X	59	EC000644			311	DFP	dcffix[.]	Decimal Floating Convert From Fixed
X	59	EC000684			313	DFP	denbcd[.]	Decimal Floating Encode BCD To DPD
X	59	EC00069C			156	FP[R]	fcfids[.]	Floating Convert From Integer Doubleword Single

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	59	EC0006C4			314	DFP	diex[.]	Decimal Floating Insert Exponent
X	59	EC00079C			157	FP[R]	fcfidus[.]	Floating Convert From Integer Doubleword Unsigned Single
XX3	60	F0000000			405	VSX	xsaddsp	VSX Scalar Add Single-Precision
XX3	60	F0000008			432	VSX	xsmaddasp	VSX Scalar Multiply-Add Type-A Single-Precision
XX3	60	F0000010			586	VSX	xxsldwi	VSX Shift Left Double by Word Immediate
XX2	60	F0000028			472	VSX	xsrsqrtesp	VSX Scalar Reciprocal Square Root Estimate Single-Precision
XX2	60	F000002C			474	VSX	xssqrtsp	VSX Scalar Square Root Single-Precision
XX4	60	F0000030			585	VSX	xxsel	VSX Select
XX3	60	F0000040			477	VSX	xssubsp	VSX Scalar Subtract Single-Precision
XX3	60	F0000048			432	VSX	xsmaddmsp	VSX Scalar Multiply-Add Type-M Single-Precision
XX3	60	F0000050			585	VSX	xxpermdi	VSX Permute Doubleword Immediate
XX3	60	F0000068			469	VSX	xsresp	VSX Scalar Reciprocal Estimate Single-Precision
XX3	60	F0000080			447	VSX	xsmulsp	VSX Scalar Multiply Single-Precision
XX3	60	F0000088			442	VSX	xsmsubasp	VSX Scalar Multiply-Subtract Type-A Single-Precision
XX3	60	F0000090			584	VSX	xxmrghw	VSX Merge High Word
XX3	60	F00000C0			427	VSX	xsdivsp	VSX Scalar Divide Single-Precision
XX3	60	F00000C8			442	VSX	xsmsubmsp	VSX Scalar Multiply-Subtract Type-M Single-Precision
XX3	60	F0000100			400	VSX	xsadddp	VSX Scalar Add Double-Precision
XX3	60	F0000108			429	VSX	xsmaddadp	VSX Scalar Multiply-Add Type-A Double-Precision
XX3	60	F0000118			409	VSX	xscmpudp	VSX Scalar Compare Unordered Double-Precision
XX2	60	F0000120			418	VSX	xscvdpuxws	VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F0000124			464	VSX	xsrdpi	VSX Scalar Round to Double-Precision Integer
XX2	60	F0000128			471	VSX	xsrsqrtedp	VSX Scalar Reciprocal Square Root Estimate Double-Precision
XX2	60	F000012C			473	VSX	xssqrtdp	VSX Scalar Square Root Double-Precision
XX3	60	F0000140			475	VSX	xssubdp	VSX Scalar Subtract Double-Precision
XX3	60	F0000148			429	VSX	xsmaddmdp	VSX Scalar Multiply-Add Type-M Double-Precision
XX3	60	F0000158			407	VSX	xscmpodp	VSX Scalar Compare Ordered Double-Precision
XX2	60	F0000160			413	VSX	xscvdpsxws	VSX Scalar Convert Double-Precision to Signed Fixed-Point Word Saturate
XX2	60	F0000164			467	VSX	xsrdpiz	VSX Scalar Round to Double-Precision Integer toward Zero
XX3	60	F0000168			468	VSX	xsredp	VSX Scalar Reciprocal Estimate Double-Precision
XX3	60	F0000180			445	VSX	xsmuldp	VSX Scalar Multiply Double-Precision
XX3	60	F0000188			439	VSX	xsmsubadp	VSX Scalar Multiply-Subtract Type-A Double-Precision
XX3	60	F0000190			584	VSX	xxmrglw	VSX Merge Low Word
XX2	60	F00001A4			466	VSX	xsrdpip	VSX Scalar Round to Double-Precision Integer toward +Infinity
XX2	60	F00001A8			480	VSX	xstsqrtdp	VSX Scalar Test for software Square Root Double-Precision
XX2	60	F00001AC			465	VSX	xsrdpic	VSX Scalar Round to Double-Precision Integer using Current rounding mode

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX3	60	F00001C0			425	VSX	xsdivdp	VSX Scalar Divide Double-Precision
XX3	60	F00001C8			439	VSX	xsmsubmdp	VSX Scalar Multiply-Subtract Type-M Double-Precision
XX2	60	F00001E4			466	VSX	xsrdpim	VSX Scalar Round to Double-Precision Integer toward -Infinity
XX3	60	F00001E8			479	VSX	xstdivdp	VSX Scalar Test for software Divide Double-Precision
XX3	60	F0000200			486	VSX	xvaddsp	VSX Vector Add Single-Precision
XX3	60	F0000208			521	VSX	xvmaddasp	VSX Vector Multiply-Add Type-A Single-Precision
XX3	60	F0000218			489	VSX	xvcmpeqsp	VSX Vector Compare Equal To Single-Precision
XX2	60	F0000220			511	VSX	xvcvspuxws	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F0000224			566	VSX	xvrspi	VSX Vector Round to Single-Precision Integer
XX2	60	F0000228			570	VSX	xvrsqrtesp	VSX Vector Reciprocal Square Root Estimate Single-Precision
XX2	60	F000022C			572	VSX	xvsqrtsp	VSX Vector Square Root Single-Precision
XX3	60	F0000240			575	VSX	xvsubsp	VSX Vector Subtract Single-Precision
XX3	60	F0000248			524	VSX	xvmaddmsp	VSX Vector Multiply-Add Type-M Single-Precision
XX3	60	F0000258			493	VSX	xvcmpgtsp	VSX Vector Compare Greater Than Single-Precision
XX2	60	F0000260			507	VSX	xvcvspsxws	VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate
XX2	60	F0000264			568	VSX	xvrspiz	VSX Vector Round to Single-Precision Integer toward Zero
XX3	60	F0000268			565	VSX	xvresp	VSX Vector Reciprocal Estimate Single-Precision
XX3	60	F0000280			543	VSX	xvmulsp	VSX Vector Multiply Single-Precision
XX3	60	F0000288			535	VSX	xvmsubasp	VSX Vector Multiply-Subtract Type-A Single-Precision
XX3	60	F0000290			586	VSX	xxspltw	VSX Splat Word
XX3	60	F0000298			491	VSX	xvcmpgesp	VSX Vector Compare Greater Than or Equal To Single-Precision
XX2	60	F00002A0			516	VSX	xvcvuxwsp	VSX Vector Convert Unsigned Fixed-Point Word to Single-Precision
XX2	60	F00002A4			567	VSX	xvrspip	VSX Vector Round to Single-Precision Integer toward +Infinity
XX2	60	F00002A8			579	VSX	xvtsqrtsp	VSX Vector Test for software Square Root Single-Precision
XX2	60	F00002AC			566	VSX	xvrspic	VSX Vector Round to Single-Precision Integer using Current rounding mode
XX3	60	F00002C0			519	VSX	xvdivsp	VSX Vector Divide Single-Precision
XX3	60	F00002C8			538	VSX	xvmsubmsp	VSX Vector Multiply-Subtract Type-M Single-Precision
XX2	60	F00002E0			514	VSX	xvcvsxwsp	VSX Vector Convert Signed Fixed-Point Word to Single-Precision
XX2	60	F00002E4			567	VSX	xvrspim	VSX Vector Round to Single-Precision Integer toward -Infinity
XX3	60	F00002E8			578	VSX	xvtdivsp	VSX Vector Test for software Divide Single-Precision
XX3	60	F0000300			482	VSX	xvadddp	VSX Vector Add Double-Precision
XX3	60	F0000308			521	VSX	xvmaddadp	VSX Vector Multiply-Add Type-A Double-Precision
XX3	60	F0000318			488	VSX	xvcmpeqdp	VSX Vector Compare Equal To Double-Precision

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F0000320			502	VSX	xvcvdpuxws	VSX Vector Convert Double-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F0000324			561	VSX	xvrdpi	VSX Vector Round to Double-Precision Integer
XX2	60	F0000328			568	VSX	xvrsqrtedp	VSX Vector Reciprocal Square Root Estimate Double-Precision
XX2	60	F000032C			571	VSX	xvsqrtdp	VSX Vector Square Root Double-Precision
XX3	60	F0000340			573	VSX	xvsubdp	VSX Vector Subtract Double-Precision
XX3	60	F0000348			524	VSX	xvmaddmdp	VSX Vector Multiply-Add Type-M Double-Precision
XX3	60	F0000358			492	VSX	xvcmpgtdp	VSX Vector Compare Greater Than Double-Precision
XX2	60	F0000360			498	VSX	xvcvdpsxws	VSX Vector Convert Double-Precision to Signed Fixed-Point Word Saturate
XX2	60	F0000364			563	VSX	xvrdpiz	VSX Vector Round to Double-Precision Integer toward Zero
XX3	60	F0000368			564	VSX	xvredp	VSX Vector Reciprocal Estimate Double-Precision
XX3	60	F0000380			541	VSX	xvmuldp	VSX Vector Multiply Double-Precision
XX3	60	F0000388			535	VSX	xvmsubadp	VSX Vector Multiply-Subtract Type-A Double-Precision
XX3	60	F0000398			490	VSX	xvcmpgedp	VSX Vector Compare Greater Than or Equal To Double-Precision
XX2	60	F00003A0			516	VSX	xvcvuxwdp	VSX Vector Convert Unsigned Fixed-Point Word to Double-Precision
XX2	60	F00003A4			562	VSX	xvrdpip	VSX Vector Round to Double-Precision Integer toward +Infinity
XX2	60	F00003A8			579	VSX	xvtsqrtdp	VSX Vector Test for software Square Root Double-Precision
XX2	60	F00003AC			561	VSX	xvrdpic	VSX Vector Round to Double-Precision Integer using Current rounding mode
XX3	60	F00003C0			517	VSX	xvdivdp	VSX Vector Divide Double-Precision
XX3	60	F00003C8			538	VSX	xvmsubmdp	VSX Vector Multiply-Subtract Type-M Double-Precision
XX2	60	F00003E0			514	VSX	xvcvsxwdp	VSX Vector Convert Signed Fixed-Point Word to Double-Precision
XX2	60	F00003E4			562	VSX	xvrdpim	VSX Vector Round to Double-Precision Integer toward -Infinity
XX3	60	F00003E8			577	VSX	xvtdivdp	VSX Vector Test for software Divide Double-Precision
VX	60	F0000400			267	V	bcdadd.	Decimal Add Modulo
XX3	60	F0000408			455	VSX	xsnmaddasp	VSX Scalar Negative Multiply-Add Type-A Single-Precision
XX3	60	F0000410			580	VSX	xxland	VSX Logical AND
XX2	60	F0000424			412	VSX	xscvdpsp	VSX Scalar Convert Double-Precision to Single-Precision
XX2	60	F000042C			413	VSX	xscvdpspn	VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling
VX	60	F0000440			267	V	bcdsub.	Decimal Subtract Modulo
XX3	60	F0000448			455	VSX	xsnmaddmsp	VSX Scalar Negative Multiply-Add Type-M Single-Precision
XX3	60	F0000450			580	VSX	xxlandc	VSX Logical AND with Complement
XX2	60	F0000464			470	VSX	xsrsp	VSX Scalar Round to Single-Precision
XX3	60	F0000488			461	VSX	xsnmsubasp	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
XX3	60	F0000490			583	VSX	xxlor	VSX Logical OR
XX2	60	F00004A0			424	VSX	xscvuxdsp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX3	60	F00004C8			461	VSX	xsnmsubmsp	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision
XX3	60	F00004D0			583	VSX	xxlxor	VSX Logical XOR
XX2	60	F00004E0			423	VSX	xscvsxdsp	VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision
XX3	60	F0000500			435	VSX	xsmaxdp	VSX Scalar Maximum Double-Precision
XX3	60	F0000508			450	VSX	xsnmaddadp	VSX Scalar Negative Multiply-Add Type-A Double-Precision
XX3	60	F0000510			582	VSX	xxlnor	VSX Logical NOR
XX2	60	F0000520			416	VSX	xscvdpuxds	VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000524			420	VSX	xscvspdp	VSX Scalar Convert Single-Precision to Double-Precision (p=1)
XX2	60	F000052C			422	VSX	xscvspdpn	Scalar Convert Single-Precision to Double-Precision format Non-signalling
XX3	60	F0000540			437	VSX	xsmindp	VSX Scalar Minimum Double-Precision
XX3	60	F0000548			450	VSX	xsnmaddmdp	VSX Scalar Negative Multiply-Add Type-M Double-Precision
XX3	60	F0000550			582	VSX	xxlorc	VSX Logical OR with Complement
XX2	60	F0000560			422	VSX	xscvdpsxds	VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000564			399	VSX	xsabsdp	VSX Scalar Absolute Value Double-Precision
XX3	60	F0000580			411	VSX	xscpsgndp	VSX Scalar Copy Sign Double-Precision
XX3	60	F0000588			458	VSX	xsnmsubadp	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision
XX3	60	F0000590			581	VSX	xxlnand	VSX Logical NAND
XX2	60	F00005A0			424	VSX	xscvuxddp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Double-Precision
XX2	60	F00005A4			449	VSX	xsnabsdp	VSX Scalar Negative Absolute Value Double-Precision
XX3	60	F00005C8			458	VSX	xsnmsubmdp	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision
XX3	60	F00005D0			581	VSX	xxleqv	VSX Logical Equivalence
XX2	60	F00005E0			423	VSX	xscvsxddp	VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision
XX2	60	F00005E4			449	VSX	xsnegdp	VSX Scalar Negate Double-Precision
XX3	60	F0000600			529	VSX	xvmaxsp	VSX Vector Maximum Single-Precision
XX3	60	F0000608			547	VSX	xvnmaddasp	VSX Vector Negative Multiply-Add Type-A Single-Precision
XX3	60	F0000618			489	VSX	xvcmpeqsp.	VSX Vector Compare Equal To Single-Precision & record CR6
XX2	60	F0000620			509	VSX	xvcvspuxds	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000624			495	VSX	xvcvdpsp	VSX Vector Convert Double-Precision to Single-Precision
XX3	60	F0000640			533	VSX	xvminsp	VSX Vector Minimum Single-Precision
XX3	60	F0000648			552	VSX	xvnmaddmsp	VSX Vector Negative Multiply-Add Type-M Single-Precision
XX3	60	F0000658			493	VSX	xvcmpgtsp.	VSX Vector Compare Greater Than Single-Precision & record CR6
XX2	60	F0000660			505	VSX	xvcvspsxds	VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000664			481	VSX	xvabssp	VSX Vector Absolute Value Single-Precision
XX3	60	F0000680			494	VSX	xvcpsgnsp	VSX Vector Copy Sign Single-Precision
XX3	60	F0000688			555	VSX	xvnmsubasp	VSX Vector Negative Multiply-Subtract Type-A Single-Precision
XX3	60	F0000698			491	VSX	xvcmpgesp.	VSX Vector Compare Greater Than or Equal To Single-Precision & record CR6

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F00006A0			515	VSX	xvcvuxdsp	VSX Vector Convert Unsigned Fixed-Point Doubleword to Single-Precision
XX2	60	F00006A4			545	VSX	xvnabssp	VSX Vector Negative Absolute Value Single-Precision
XX3	60	F00006C8			558	VSX	xvnmsubmsp	VSX Vector Negative Multiply-Subtract Type-M Single-Precision
XX2	60	F00006E0			513	VSX	xvcvsxdsp	VSX Vector Convert Signed Fixed-Point Doubleword to Single-Precision
XX2	60	F00006E4			546	VSX	xvnegsp	VSX Vector Negate Single-Precision
XX3	60	F0000700			527	VSX	xvmaxdp	VSX Vector Maximum Double-Precision
XX3	60	F0000708			547	VSX	xvnmaddadp	VSX Vector Negative Multiply-Add Type-A Double-Precision
XX3	60	F0000718			488	VSX	xvcmpeqdp.	VSX Vector Compare Equal To Double-Precision & record CR6
XX2	60	F0000720			500	VSX	xvcvdpuxds	VSX Vector Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000724			504	VSX	xvcvspdp	VSX Vector Convert Single-Precision to Double-Precision
XX3	60	F0000740			531	VSX	xvmindp	VSX Vector Minimum Double-Precision
XX3	60	F0000748			552	VSX	xvnmaddmdp	VSX Vector Negative Multiply-Add Type-M Double-Precision
XX3	60	F0000758			492	VSX	xvcmpgtdp.	VSX Vector Compare Greater Than Double-Precision & record CR6
XX2	60	F0000760			496	VSX	xvcvdpsxds	VSX Vector Convert Double-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000764			480	VSX	xvabsdp	VSX Vector Absolute Value Double-Precision
XX3	60	F0000780			494	VSX	xvcpsgndp	VSX Vector Copy Sign Double-Precision
XX3	60	F0000788			555	VSX	xvnmsubadp	VSX Vector Negative Multiply-Subtract Type-A Double-Precision
XX3	60	F0000798			490	VSX	xvcmpgedp.	VSX Vector Compare Greater Than or Equal To Double-Precision & record CR6
XX2	60	F00007A0			515	VSX	xvcvuxddp	VSX Vector Convert Unsigned Fixed-Point Doubleword to Double-Precision
XX2	60	F00007A4			545	VSX	xvnabsdp	VSX Vector Negative Absolute Value Double-Precision
XX3	60	F00007C8			558	VSX	xvnmsubmdp	VSX Vector Negative Multiply-Subtract Type-M Double-Precision
XX2	60	F00007E0			513	VSX	xvcvsxddp	VSX Vector Convert Signed Fixed-Point Doubleword to Double-Precision
XX2	60	F00007E4			546	VSX	xvnegdp	VSX Vector Negate Double-Precision
DS	61	F4000000			141	FP.out	stfdp	Store Floating-Point Double Pair
DS	62	F8000000			57	64	std	Store Doubleword
DS	62	F8000001			57	64	stdu	Store Doubleword with Update
DS	62	F8000002		P	58	LSQ	stq	Store Quadword
X	63	FC000000			159	FP	fcmpu	Floating Compare Unordered
X	63	FC000004			289	DFP	daddq[.]	Decimal Floating Add Quad
Z23	63	FC000006			300	DFP	dquaq[.]	Decimal Quantize Quad
X	63	FC000010			142	FP[R]	fcpsgn[.]	Floating Copy Sign
X	63	FC000018			151	FP[R]	frsp[.]	Floating Round to Single-Precision
X	63	FC00001C			153	FP[R]	fctiw[.]	Floating Convert To Integer Word
X	63	FC00001E			154	FP[R]	fctiwz[.]	Floating Convert To Integer Word with round to Zero
A	63	FC000024			145	FP[R]	fdiv[.]	Floating Divide
A	63	FC000028			144	FP[R]	fsub[.]	Floating Subtract
A	63	FC00002A			144	FP[R]	fadd[.]	Floating Add
A	63	FC00002C			146	FP[R]	fsqrt[.]	Floating Square Root
A	63	FC00002E			160	FP[R]	fsel[.]	Floating Select

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
A	63	FC000030			146	FP[R].in	fre[.]	Floating Reciprocal Estimate
A	63	FC000032			145	FP[R]	fmul[.]	Floating Multiply
A	63	FC000034			147	FP[R]	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	63	FC000038			149	FP[R]	fmsub[.]	Floating Multiply-Subtract
A	63	FC00003A			149	FP[R]	fmadd[.]	Floating Multiply-Add
A	63	FC00003C			150	FP[R]	fnmsub[.]	Floating Negative Multiply-Subtract
A	63	FC00003E			150	FP[R]	fnmadd[.]	Floating Negative Multiply-Add
X	63	FC000040			159	FP	fcmpo	Floating Compare Ordered
X	63	FC000044			291	DFP	dmulq[.]	Decimal Floating Multiply Quad
Z23	63	FC000046			302	DFP	drrndq[.]	Decimal Floating Reround Quad
X	63	FC00004C			163	FP[R]	mtfsb1[.]	Move To FPSCR Bit 1
X	63	FC000050			142	FP[R]	fneg[.]	Floating Negate
X	63	FC000080			161	FP	mcrfs	Move To Condition Register from FPSCR
Z22	63	FC000084			316	DFP	dscliq[.]	Decimal Floating Shift Coefficient Left Immediate Quad
Z23	63	FC000086			299	DFP	dquaiq[.]	Decimal Quantize Immediate Quad
X	63	FC00008C			163	FP[R]	mtfsb0[.]	Move To FPSCR Bit 0
X	63	FC000090			142	FP[R]	fmr[.]	Floating Move Register
Z22	63	FC0000C4			316	DFP	dscriq[.]	Decimal Floating Shift Coefficient Right Immediate Quad
Z23	63	FC0000C6			305	DFP	drintxq[.]	Decimal Floating Round To FP Integer With Inexact Quad
X	63	FC000100			148	FP	ftdiv	Floating Test for software Divide
X	63	FC000104			295	DFP	dcmpoq	Decimal Floating Compare Ordered Quad
X	63	FC00010C			162	FP[R]	mtfsfi[.]	Move To FPSCR Field Immediate
X	63	FC000110			142	FP[R]	fnabs[.]	Floating Negative Absolute Value
X	63	FC00011C			154	FP[R]	fctiwu[.]	Floating Convert To Integer Word Unsigned
X	63	FC00011E			155	FP[R]	fctiwuz[.]	Floating Convert To Integer Word Unsigned with round toward Zero
X	63	FC000140			148	FP	ftsqrt	Floating Test for software Square Root
X	63	FC000144			297	DFP	dtstexq	Decimal Floating Test Exponent Quad
Z22	63	FC000184			296	DFP	dtstdcq	Decimal Floating Test Data Class Quad
Z22	63	FC0001C4			296	DFP	dtstdgq	Decimal Floating Test Data Group Quad
Z23	63	FC0001C6			307	DFP	drintnq[.]	Decimal Floating Round To FP Integer Without Inexact Quad
X	63	FC000204			309	DFP	dctqpq[.]	Decimal Floating Convert To DFP Extended
X	63	FC000210			142	FP[R]	fabs[.]	Floating Absolute Value
X	63	FC000244			311	DFP	dctfixq[.]	Decimal Floating Convert To Fixed Quad
X	63	FC000284			313	DFP	ddedpdq[.]	Decimal Floating Decode DPD To BCD Quad
X	63	FC0002C4			314	DFP	dxexq[.]	Decimal Floating Extract Exponent Quad
X	63	FC000310			158	FP[R].in	frin[.]	Floating Round To Integer Nearest
X	63	FC000350			158	FP[R].in	friz[.]	Floating Round To Integer toward Zero
X	63	FC000390			158	FP[R].in	frip[.]	Floating Round To Integer Plus
X	63	FC0003D0			158	FP[R].in	frim[.]	Floating Round To Integer Minus
X	63	FC000404			289	DFP	dsubq[.]	Decimal Floating Subtract Quad
X	63	FC000444			292	DFP	ddivq[.]	Decimal Floating Divide Quad
X	63	FC00048E			161	FP[R]	mffs[.]	Move From FPSCR
X	63	FC000504			295	DFP	dcmpuq	Decimal Floating Compare Unordered Quad
X	63	FC000544			298	DFP	dtstsfq	Decimal Floating Test Significance Quad
XFL	63	FC00058E			162	FP[R]	mtfsf[.]	Move To FPSCR Fields
X	63	FC000604			310	DFP	drdpq[.]	Decimal Floating Round To DFP Long
X	63	FC000644			311	DFP	dcffixq[.]	Decimal Floating Convert From Fixed Quad
X	63	FC00065C			151	FP[R]	fctid[.]	Floating Convert To Integer Doubleword
X	63	FC00065E			152	FP[R]	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	63	FC000684			313	DFP	denbcdq[.]	Decimal Floating Encode BCD To DPD Quad
X	63	FC00068C			143	VSX	fmrgow	Floating Merge Odd Word
X	63	FC00069C			155	FP[R]	fcfid[.]	Floating Convert From Integer Doubleword
X	63	FC0006C4			314	DFP	diexq[.]	Decimal Floating Insert Exponent Quad
X	63	FC00075C			152	FP[R]	fctidu[.]	Floating Convert To Integer Doubleword Unsigned
X	63	FC00075E			153	FP[R]	fctiduz[.]	Floating Convert To Integer Doubleword Unsigned with round toward Zero
X	63	FC00078C			143	VSX	fmrgew	Floating Merge Even Word
X	63	FC00079C			156	FP[R]	fcfidu[.]	Floating Convert From Integer Doubleword Unsigned

[1] See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.

Appendix I. Power ISA Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the Power ISA, sorted by mnemonic.

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	31	7C000214	SR		67	B	add[.]	Add
XO	31	7C000014	SR		68	B	addc[.]	Add Carrying
XO	31	7C000414	SR		68	B	addco[.]	Add Carrying & record OV
XO	31	7C000114	SR		69	B	adde[.]	Add Extended
XO	31	7C000514	SR		69	B	addeo[.]	Add Extended & record OV & record OV
XO	31	7C000094		H	101	BCDA	addg6s	Add and Generate Sixes
D	14	38000000			66	B	addi	Add Immediate
D	12	30000000	SR		67	B	addic	Add Immediate Carrying
D	13	34000000	SR		67	B	addic.	Add Immediate Carrying & record CR0
D	15	3C000000			66	B	addis	Add Immediate Shifted
XO	31	7C0001D4	SR		69	B	addme[.]	Add to Minus One Extended
XO	31	7C0005D4	SR		69	B	addmeo[.]	Add to Minus One Extended & record OV
XO	31	7C000614	SR		67	B	addo[.]	Add & record OV
XO	31	7C000194	SR		70	B	addze[.]	Add to Zero Extended
XO	31	7C000594	SR		70	B	addzeo[.]	Add to Zero Extended & record OV
X	31	7C000038	SR		84	B	and[.]	AND
X	31	7C000078	SR		85	B	andc[.]	AND with Complement
D	28	70000000	SR		82	B	andi.	AND Immediate & record CR0
D	29	74000000	SR		82	B	andis.	AND Immediate Shifted & record CR0
I	18	48000000			38	B	b[l][a]	Branch
B	16	40000000	CT		38	B	bc[l][a]	Branch Conditional
XL	19	4C000420	CT		39	B	bcctr[l]	Branch Conditional to Count Register
VX	60	F0000400			267	V	bcdadd.	Decimal Add Modulo
VX	60	F0000440			267	V	bcdsub.	Decimal Subtract Modulo
XL	19	4C000020	CT		39	B	bclr[l]	Branch Conditional to Link Register
X	19	4C000460			40	B	bctar[l]	Branch Conditional to Branch Target Address Register
X	31	7C0001F8			90	64	bpermd	Bit Permute Doubleword
EVX	4	1000020F			594	SP	brinc	Bit Reversed Increment
X	31	7C000274		H	101	BCDA	cbcdtd	Convert Binary Coded Decimal To Declets
X	31	7C000234		H	101	BCDA	cdtbcd	Convert Declets To Binary Coded Decimal
X	31	7C00035C			44	S	clrbhrb	Clear BHRB
X	31	7C000000			78	B	cmp	Compare
X	31	7C0003F8			86	B	cmpb	Compare Byte
D	11	2C000000			78	B	cmpi	Compare Immediate
X	31	7C000040			79	B	cmpl	Compare Logical
D	10	28000000			79	B	cmpli	Compare Logical Immediate
X	31	7C000074	SR		89	64	cntlzd[.]	Count Leading Zeros Doubleword
X	31	7C000034	SR		85	B	cntlzw[.]	Count Leading Zeros Word
XL	19	4C000202			41	B	crand	Condition Register AND

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XL	19	4C000102			42	B	crandc	Condition Register AND with Complement
XL	19	4C000242			42	B	creqv	Condition Register Equivalent
XL	19	4C0001C2			41	B	crnand	Condition Register NAND
XL	19	4C000042			42	B	crnor	Condition Register NOR
XL	19	4C000382			41	B	cror	Condition Register OR
XL	19	4C000342			42	B	crorc	Condition Register OR with Complement
XL	19	4C000182			41	B	crxor	Condition Register XOR
X	59	EC000004			289	DFP	dadd[.]	Decimal Floating Add
X	63	FC000004			289	DFP	daddq[.]	Decimal Floating Add Quad
X	31	7C0005EC			770	E	dcba	Data Cache Block Allocate
X	31	7C0000AC			773	B	dcbf	Data Cache Block Flush
X	31	7C0000FE		P	1064	E.PD	dcbfep	Data Cache Block Flush by External PID
X	31	7C0003AC		P	1118	E	dcbi	Data Cache Block Invalidate
X	31	7C00030C		M	1123	ECL	dcblc	Data Cache Block Lock Clear
X	31	7C00034D		M	1121	ECL	dcblq.	Data Cache Block Lock Query
X	31	7C00006C			773	B	dcbst	Data Cache Block Store
X	31	7C00007E		P	1063	E.PD	dcbstep	Data Cache Block Store by External PID
X	31	7C00022C			770	B	dcbt	Data Cache Block Touch
X	31	7C00027E		P	1063	E.PD	dcbtep	Data Cache Block Touch by External PID
X	31	7C00014C		M	1122	ECL	dcbtls	Data Cache Block Touch and Lock Set
X	31	7C0001EC			771	B	dcbtst	Data Cache Block Touch for Store
X	31	7C0001FE		P	1066	E.PD	dcbtstep	Data Cache Block Touch for Store by External PID
X	31	7C00010C		M	1122	ECL	dcbtstls	Data Cache Block Touch for Store and Lock Set
X	31	7C0007EC			773	B	dcbz	Data Cache Block Zero
X	31	7C0007FE		P	1067	E.PD	dcbzep	Data Cache Block Zero by External PID
X	59	EC000644			311	DFP	dcffix[.]	Decimal Floating Convert From Fixed
X	63	FC000644			311	DFP	dcffixq[.]	Decimal Floating Convert From Fixed Quad
X	31	7C00038C		P	1239	E.CI	dci	Data Cache Invalidate
X	59	EC000104			295	DFP	dcmpo	Decimal Floating Compare Ordered
X	63	FC000104			295	DFP	dcmpoq	Decimal Floating Compare Ordered Quad
X	59	EC000504			294	DFP	dcmpu	Decimal Floating Compare Unordered
X	63	FC000504			295	DFP	dcmpuq	Decimal Floating Compare Unordered Quad
X	31	7C00028C		P	1242	E.CD	dcread	Data Cache Read
X	31	7C0003CC		P	1242	E.CD	dcread	Data Cache Read
X	59	EC000204			309	DFP	dctdp[.]	Decimal Floating Convert To DFP Long
X	59	EC000244			311	DFP	dctfix[.]	Decimal Floating Convert To Fixed
X	63	FC000244			311	DFP	dctfixq[.]	Decimal Floating Convert To Fixed Quad
X	63	FC000204			309	DFP	dctqpq[.]	Decimal Floating Convert To DFP Extended
X	59	EC000284			313	DFP	ddedpd[.]	Decimal Floating Decode DPD To BCD
X	63	FC000284			313	DFP	ddedpdq[.]	Decimal Floating Decode DPD To BCD Quad
X	59	EC000444			292	DFP	ddiv[.]	Decimal Floating Divide
X	63	FC000444			292	DFP	ddivq[.]	Decimal Floating Divide Quad
X	59	EC000684			313	DFP	denbcd[.]	Decimal Floating Encode BCD To DPD
X	63	FC000684			313	DFP	denbcdq[.]	Decimal Floating Encode BCD To DPD Quad
X	59	EC0006C4			314	DFP	diex[.]	Decimal Floating Insert Exponent
X	63	FC0006C4			314	DFP	diexq[.]	Decimal Floating Insert Exponent Quad
XO	31	7C0003D2	SR		76	64	divd[.]	Divide Doubleword
XO	31	7C000352	SR		77	64	divde[.]	Divide Doubleword Extended
XO	31	7C000752	SR		77	64	divdeo[.]	Divide Doubleword Extended & record OV
XO	31	7C000312	SR		77	64	divdeu[.]	Divide Doubleword Extended Unsigned
XO	31	7C000712	SR		77	64	divdeuo[.]	Divide Doubleword Extended Unsigned & record OV

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	31	7C0007D2	SR		76	64	divdo[.]	Divide Doubleword & record OV
XO	31	7C000392	SR		76	64	divdu[.]	Divide Doubleword Unsigned
XO	31	7C000792	SR		76	64	divduo[.]	Divide Doubleword Unsigned & record OV
XO	31	7C0003D6	SR		72	B	divw[.]	Divide Word
XO	31	7C000356	SR		73	B	divwe[.]	Divide Word Extended
XO	31	7C000756	SR		73	B	divweo[.]	Divide Word Extended & record OV
XO	31	7C000316	SR		73	B	divweu[.]	Divide Word Extended Unsigned
XO	31	7C000716	SR		73	B	divweuo[.]	Divide Word Extended Unsigned & record OV
XO	31	7C0007D6	SR		72	B	divwo[.]	Divide Word & record OV
XO	31	7C000396	SR		72	B	divwu[.]	Divide Word Unsigned
XO	31	7C000796	SR		72	B	divwuo[.]	Divide Word Unsigned & record OV
X	31	7C00009C			673	LMV	dlmzb[.]	Determine Leftmost Zero Byte
X	59	EC000044			291	DFP	dmul[.]	Decimal Floating Multiply
X	63	FC000044			291	DFP	dmulq[.]	Decimal Floating Multiply Quad
XFX	19	4C00018C			1228	E	dnh	Debugger Notify Halt
XL	19	4C000324		H	867	S	doze	Doze
Z23	59	EC000006			300	DFP	dqua[.]	Decimal Quantize
Z23	59	EC000086			299	DFP	dquai[.]	Decimal Quantize Immediate
Z23	63	FC000086			299	DFP	dquaiq[.]	Decimal Quantize Immediate Quad
Z23	63	FC000006			300	DFP	dquaq[.]	Decimal Quantize Quad
X	63	FC000604			310	DFP	drdpq[.]	Decimal Floating Round To DFP Long
Z23	59	EC0001C6			307	DFP	drintn[.]	Decimal Floating Round To FP Integer Without Inexact
Z23	63	FC0001C6			307	DFP	drintnq[.]	Decimal Floating Round To FP Integer Without Inexact Quad
Z23	59	EC0000C6			305	DFP	drintx[.]	Decimal Floating Round To FP Integer With Inexact
Z23	63	FC0000C6			305	DFP	drintxq[.]	Decimal Floating Round To FP Integer With Inexact Quad
Z23	59	EC000046			302	DFP	drrnd[.]	Decimal Floating Reround
Z23	63	FC000046			302	DFP	drrndq[.]	Decimal Floating Reround Quad
X	59	EC000604			310	DFP	drsp[.]	Decimal Floating Round To DFP Short
Z22	59	EC000084			316	DFP	dscli[.]	Decimal Floating Shift Coefficient Left Immediate
Z22	63	FC000084			316	DFP	dscliq[.]	Decimal Floating Shift Coefficient Left Immediate Quad
Z22	59	EC0000C4			316	DFP	dscri[.]	Decimal Floating Shift Coefficient Right Immediate
Z22	63	FC0000C4			316	DFP	dscriq[.]	Decimal Floating Shift Coefficient Right Immediate Quad
X	31	7C0003C6			824	DS	dsn	Decorated Storage Notify
X	59	EC000404			289	DFP	dsub[.]	Decimal Floating Subtract
X	63	FC000404			289	DFP	dsubq[.]	Decimal Floating Subtract Quad
Z22	59	EC000184			296	DFP	dtstdc	Decimal Floating Test Data Class
Z22	63	FC000184			296	DFP	dtstdcq	Decimal Floating Test Data Class Quad
Z22	59	EC0001C4			296	DFP	dtstdg	Decimal Floating Test Data Group
Z22	63	FC0001C4			296	DFP	dtstdgq	Decimal Floating Test Data Group Quad
X	59	EC000144			297	DFP	dtstex	Decimal Floating Test Exponent
X	63	FC000144			297	DFP	dtstexq	Decimal Floating Test Exponent Quad
X	59	EC000544			298	DFP	dtstsf	Decimal Floating Test Significance
X	63	FC000544			298	DFP	dtstsfq	Decimal Floating Test Significance Quad
X	59	EC0002C4			314	DFP	dxex[.]	Decimal Floating Extract Exponent
X	63	FC0002C4			314	DFP	dxexq[.]	Decimal Floating Extract Exponent Quad
X	31	7C00026C			826	EC	eciwx	External Control In Word Indexed
X	31	7C00036C			826	EC	ecowx	External Control Out Word Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100002E4			660	SP.FD	efdabs	Floating-Point Double-Precision Absolute Value
EVX	4	100002E0			661	SP.FD	efdadd	Floating-Point Double-Precision Add
EVX	4	100002EF			666	SP.FD	efdcfs	Floating-Point Double-Precision Convert from Single-Precision
EVX	4	100002F3			664	SP.FD	efdcfsf	Convert Floating-Point Double-Precision from Signed Fraction
EVX	4	100002F1			663	SP.FD	efdcfsi	Convert Floating-Point Double-Precision from Signed Integer
EVX	4	100002E3			664	SP.FD	efdcfsid	Convert Floating-Point Double-Precision from Signed Integer Doubleword
EVX	4	100002F2			664	SP.FD	efdcfuf	Convert Floating-Point Double-Precision from Unsigned Fraction
EVX	4	100002F0			663	SP.FD	efdcfui	Convert Floating-Point Double-Precision from Unsigned Integer
EVX	4	100002E2			664	SP.FD	efdcfuid	Convert Floating-Point Double-Precision from Unsigned Integer Doubleword
EVX	4	100002EE			662	SP.FD	efdcmpeq	Floating-Point Double-Precision Compare Equal
EVX	4	100002EC			662	SP.FD	efdcmpgt	Floating-Point Double-Precision Compare Greater Than
EVX	4	100002ED			662	SP.FD	efdcmplt	Floating-Point Double-Precision Compare Less Than
EVX	4	100002F7			666	SP.FD	efdctsf	Convert Floating-Point Double-Precision to Signed Fraction
EVX	4	100002F5			664	SP.FD	efdctsi	Convert Floating-Point Double-Precision to Signed Integer
EVX	4	100002EB			665	SP.FD	efdctsidz	Convert Floating-Point Double-Precision to Signed Integer Doubleword with Round toward Zero
EVX	4	100002FA			666	SP.FD	efdctsiz	Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero
EVX	4	100002F6			666	SP.FD	efdctuf	Convert Floating-Point Double-Precision to Unsigned Fraction
EVX	4	100002F4			664	SP.FD	efdctui	Convert Floating-Point Double-Precision to Unsigned Integer
EVX	4	100002EA			665	SP.FD	efdctuidz	Convert Floating-Point Double-Precision to Unsigned Integer Doubleword with Round toward Zero
EVX	4	100002F8			666	SP.FD	efdctuiz	Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero
EVX	4	100002E9			661	SP.FD	efddiv	Floating-Point Double-Precision Divide
EVX	4	100002E8			661	SP.FD	efdmul	Floating-Point Double-Precision Multiply
EVX	4	100002E5			660	SP.FD	efdnabs	Floating-Point Double-Precision Negative Absolute Value
EVX	4	100002E6			660	SP.FD	efdneg	Floating-Point Double-Precision Negate
EVX	4	100002E1			661	SP.FD	efdsub	Floating-Point Double-Precision Subtract
EVX	4	100002FE			663	SP.FD	efdtsteq	Floating-Point Double-Precision Test Equal
EVX	4	100002FC			662	SP.FD	efdtstgt	Floating-Point Double-Precision Test Greater Than
EVX	4	100002FD			663	SP.FD	efdtstlt	Floating-Point Double-Precision Test Less Than
EVX	4	100002C4			653	SP.FS	efsabs	Floating-Point Absolute Value
EVX	4	100002C0			654	SP.FS	efsadd	Floating-Point Add
EVX	4	100002CF			667	SP.FD	efscfd	Floating-Point Single-Precision Convert from Double-Precision
EVX	4	100002D3			658	SP.FS	efscfsf	Convert Floating-Point from Signed Fraction
EVX	4	100002D1			658	SP.FS	efscfsi	Convert Floating-Point from Signed Integer

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100002D2			658	SP.FS	efscfuf	Convert Floating-Point from Unsigned Fraction
EVX	4	100002D0			658	SP.FS	efscfui	Convert Floating-Point from Unsigned Integer
EVX	4	100002CE			656	SP.FS	efscmpeq	Floating-Point Compare Equal
EVX	4	100002CC			655	SP.FS	efscmpgt	Floating-Point Compare Greater Than
EVX	4	100002CD			655	SP.FS	efscmplt	Floating-Point Compare Less Than
EVX	4	100002D7			659	SP.FS	efsctsf	Convert Floating-Point to Signed Fraction
EVX	4	100002D5			658	SP.FS	efsctsi	Convert Floating-Point to Signed Integer
EVX	4	100002DA			659	SP.FS	efsctsiz	Convert Floating-Point to Signed Integer with Round toward Zero
EVX	4	100002D6			659	SP.FS	efsctuf	Convert Floating-Point to Unsigned Fraction
EVX	4	100002D4			658	SP.FS	efsctui	Convert Floating-Point to Unsigned Integer
EVX	4	100002D8			659	SP.FS	efsctuiz	Convert Floating-Point to Unsigned Integer with Round toward Zero
EVX	4	100002C9			654	SP.FS	efsdiv	Floating-Point Divide
EVX	4	100002C8			654	SP.FS	efsmul	Floating-Point Multiply
EVX	4	100002C5			653	SP.FS	efsnabs	Floating-Point Negative Absolute Value
EVX	4	100002C6			653	SP.FS	efsneg	Floating-Point Negate
EVX	4	100002C1			654	SP.FS	efssub	Floating-Point Subtract
EVX	4	100002DE			657	SP.FS	efststeq	Floating-Point Test Equal
EVX	4	100002DC			656	SP.FS	efststgt	Floating-Point Test Greater Than
EVX	4	100002DD			657	SP.FS	efststlt	Floating-Point Test Less Than
XL	31	7C00021C			1043	E.HV	ehpriv	Embedded Hypervisor Privilege
X	31	7C0006AC			790	S	eieio	Enforce In-order Execution of I/O
X	31	7C000238	SR		85	B	eqv[.]	Equivalent
EVX	4	10000208			594	SP	evabs	Vector Absolute Value
EVX	4	10000202			594	SP	evaddiw	Vector Add Immediate Word
EVX	4	100004C9			594	SP	evaddsmiaaw	Vector Add Signed, Modulo, Integer to Accumulator Word
EVX	4	100004C1			595	SP	evaddssiaaw	Vector Add Signed, Saturate, Integer to Accumulator Word
EVX	4	100004C8			595	SP	evaddumiaaw	Vector Add Unsigned, Modulo, Integer to Accumulator Word
EVX	4	100004C0			595	SP	evaddusiaaw	Vector Add Unsigned, Saturate, Integer to Accumulator Word
EVX	4	10000200			595	SP	evaddw	Vector Add Word
EVX	4	10000211			596	SP	evand	Vector AND
EVX	4	10000212			596	SP	evandc	Vector AND with Complement
EVX	4	10000234			596	SP	evcmpeq	Vector Compare Equal
EVX	4	10000231			596	SP	evcmpgts	Vector Compare Greater Than Signed
EVX	4	10000230			597	SP	evcmpgtu	Vector Compare Greater Than Unsigned
EVX	4	10000233			597	SP	evcmplts	Vector Compare Less Than Signed
EVX	4	10000232			597	SP	evcmpltu	Vector Compare Less Than Unsigned
EVX	4	1000020E			598	SP	evcntlsw	Vector Count Leading Signed Bits Word
EVX	4	1000020D			598	SP	evcntlzw	Vector Count Leading Zeros Word
EVX	4	100004C6			598	SP	evdivws	Vector Divide Word Signed
EVX	4	100004C7			599	SP	evdivwu	Vector Divide Word Unsigned
EVX	4	10000219			599	SP	eveqv	Vector Equivalent
EVX	4	1000020A			599	SP	evextsb	Vector Extend Sign Byte
EVX	4	1000020B			599	SP	evextsh	Vector Extend Sign Half Word
EVX	4	10000284			645	SP.FV	evfsabs	Vector Floating-Point Absolute Value
EVX	4	10000280			646	SP.FV	evfsadd	Vector Floating-Point Add
EVX	4	10000293			650	SP.FV	evfscfsf	Vector Convert Floating-Point from Signed Fraction

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000291			650	SP.FV	evfscfsi	Vector Convert Floating-Point from Signed Integer
EVX	4	10000292			650	SP.FV	evfscfuf	Vector Convert Floating-Point from Unsigned Fraction
EVX	4	10000290			650	SP.FV	evfscfui	Vector Convert Floating-Point from Unsigned Integer
EVX	4	1000028E			648	SP.FV	evfscmpeq	Vector Floating-Point Compare Equal
EVX	4	1000028C			647	SP.FV	evfscmpgt	Vector Floating-Point Compare Greater Than
EVX	4	1000028D			647	SP.FV	evfscmplt	Vector Floating-Point Compare Less Than
EVX	4	10000297			652	SP.FV	evfsctsf	Vector Convert Floating-Point to Signed Fraction
EVX	4	10000295			651	SP.FV	evfsctsi	Vector Convert Floating-Point to Signed Integer
EVX	4	1000029A			651	SP.FV	evfsctsiz	Vector Convert Floating-Point to Signed Integer with Round toward Zero
EVX	4	10000296			652	SP.FV	evfsctuf	Vector Convert Floating-Point to Unsigned Fraction
EVX	4	10000294			651	SP.FV	evfsctui	Vector Convert Floating-Point to Unsigned Integer
EVX	4	10000298			651	SP.FV	evfsctuiz	Vector Convert Floating-Point to Unsigned Integer with Round toward Zero
EVX	4	10000289			646	SP.FV	evfsdiv	Vector Floating-Point Divide
EVX	4	10000288			646	SP.FV	evfsmul	Vector Floating-Point Multiply
EVX	4	10000285			645	SP.FV	evfsnabs	Vector Floating-Point Negative Absolute Value
EVX	4	10000286			645	SP.FV	evfsneg	Vector Floating-Point Negate
EVX	4	10000281			646	SP.FV	evfssub	Vector Floating-Point Subtract
EVX	4	1000029E			649	SP.FV	evfststeq	Vector Floating-Point Test Equal
EVX	4	1000029C			648	SP.FV	evfststgt	Vector Floating-Point Test Greater Than
EVX	4	1000029D			649	SP.FV	evfststlt	Vector Floating-Point Test Less Than
EVX	4	10000301			600	SP	evldd	Vector Load Double Word into Double Word
EVX	31	7C00063E		P	1069	E.PD	evlddepx	Vector Load Double Word into Double Word by External PID Indexed
EVX	4	10000300			600	SP	evlddx	Vector Load Double Word into Double Word Indexed
EVX	4	10000305			600	SP	evldh	Vector Load Double into Four Half Words
EVX	4	10000304			600	SP	evldhx	Vector Load Double into Four Half Words Indexed
EVX	4	10000303			601	SP	evldw	Vector Load Double into Two Words
EVX	4	10000302			601	SP	evldwx	Vector Load Double into Two Words Indexed
EVX	4	10000309			601	SP	evlhhesplat	Vector Load Half Word into Half Words Even and Splat
EVX	4	10000308			601	SP	evlhhesplatx	Vector Load Half Word into Half Words Even and Splat Indexed
EVX	4	1000030F			602	SP	evlhhossplat	Vector Load Half Word into Half Word Odd Signed and Splat
EVX	4	1000030E			602	SP	evlhhossplatx	Vector Load Half Word into Half Word Odd Signed and Splat Indexed
EVX	4	1000030D			602	SP	evlhhousplat	Vector Load Half Word into Half Word Odd Unsigned and Splat
EVX	4	1000030C			602	SP	evlhhousplatx	Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed
EVX	4	10000311			603	SP	evlwhe	Vector Load Word into Two Half Words Even
EVX	4	10000310			603	SP	evlwhex	Vector Load Word into Two Half Words Even Indexed
EVX	4	10000317			603	SP	evlwhos	Vector Load Word into Two Half Words Odd Signed (with sign extension)

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000316			603	SP	evlwhosx	Vector Load Word into Two Half Words Odd Signed Indexed (with sign extension)
EVX	4	10000315			604	SP	evlwhou	Vector Load Word into Two Half Words Odd Unsigned (zero-extended)
EVX	4	10000314			604	SP	evlwhoux	Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)
EVX	4	1000031D			604	SP	evlwhsplat	Vector Load Word into Two Half Words and Splat
EVX	4	1000031C			604	SP	evlwhsplatx	Vector Load Word into Two Half Words and Splat Indexed
EVX	4	10000319			605	SP	evlwwsplat	Vector Load Word into Word and Splat
EVX	4	10000318			605	SP	evlwwsplatx	Vector Load Word into Word and Splat Indexed
EVX	4	1000022C			605	SP	evmergehi	Vector Merge High
EVX	4	1000022E			606	SP	evmergehilo	Vector Merge High/Low
EVX	4	1000022D			605	SP	evmergelo	Vector Merge Low
EVX	4	1000022F			606	SP	evmergelohi	Vector Merge Low/High
EVX	4	1000052B			606	SP	evmhegsmfaa	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	100005AB			606	SP	evmhegsmfan	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	10000529			607	SP	evmhegsmiaa	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate
EVX	4	100005A9			607	SP	evmhegsmian	Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	10000528			607	SP	evmhegumiaa	Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	100005A8			607	SP	evmhegumian	Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1000040B			608	SP	evmhesmf	Vector Multiply Half Words, Even, Signed, Modulo, Fractional
EVX	4	1000042B			608	SP	evmhesmfa	Vector Multiply Half Words, Even, Signed, Modulo, Fractional to Accumulator
EVX	4	1000050B			608	SP	evmhesmfaaw	Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1000058B			608	SP	evmhesmfanw	Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	10000409			609	SP	evmhesmi	Vector Multiply Half Words, Even, Signed, Modulo, Integer
EVX	4	10000429			609	SP	evmhesmia	Vector Multiply Half Words, Even, Signed, Modulo, Integer to Accumulator
EVX	4	10000509			609	SP	evmhesmiaaw	Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words
EVX	4	10000589			609	SP	evmhesmianw	Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000403			610	SP	evmhessf	Vector Multiply Half Words, Even, Signed, Saturate, Fractional
EVX	4	10000423			610	SP	evmhessfa	Vector Multiply Half Words, Even, Signed, Saturate, Fractional to Accumulator
EVX	4	10000503			611	SP	evmhessfaaw	Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000583			611	SP	evmhessfanw	Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	10000501			612	SP	evmhessiaaw	Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words
EVX	4	10000581			612	SP	evmhessianw	Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	10000408			613	SP	evmheumi	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer
EVX	4	10000428			613	SP	evmheumia	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer to Accumulator
EVX	4	10000508			613	SP	evmheumiaaw	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	10000588			613	SP	evmheumianw	Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000500			614	SP	evmheusiaaw	Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	10000580			614	SP	evmheusianw	Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	1000052F			615	SP	evmhogsmfaa	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate
EVX	4	100005AF			615	SP	evmhogsmfan	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative
EVX	4	1000052D			615	SP	evmhogsmiaa	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer, and Accumulate
EVX	4	100005AD			615	SP	evmhogsmian	Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative
EVX	4	1000052C			616	SP	evmhogumiaa	Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate
EVX	4	100005AC			616	SP	evmhogumian	Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1000040F			616	SP	evmhosmf	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional
EVX	4	1000042F			616	SP	evmhosmfa	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional to Accumulator
EVX	4	1000050F			617	SP	evmhosmfaaw	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words
EVX	4	1000058F			617	SP	evmhosmfanw	Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words
EVX	4	1000040D			617	SP	evmhosmi	Vector Multiply Half Words, Odd, Signed, Modulo, Integer
EVX	4	1000042D			617	SP	evmhosmia	Vector Multiply Half Words, Odd, Signed, Modulo, Integer to Accumulator
EVX	4	1000050D			618	SP	evmhosmiaaw	Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words
EVX	4	1000058D			617	SP	evmhosmianw	Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000407			619	SP	evmhossf	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000427			619	SP	evmhossfa	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional to Accumulator
EVX	4	10000507			620	SP	evmhossfaaw	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words
EVX	4	10000587			620	SP	evmhossfanw	Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words
EVX	4	10000505			621	SP	evmhossiaaw	Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words
EVX	4	10000585			621	SP	evmhossianw	Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words
EVX	4	1000040C			621	SP	evmhoumi	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer
EVX	4	1000042C			621	SP	evmhoumia	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer to Accumulator
EVX	4	1000050C			622	SP	evmhoumiaaw	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words
EVX	4	1000058C			618	SP	evmhoumianw	Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words
EVX	4	10000504			622	SP	evmhousiaaw	Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words
EVX	4	10000584			622	SP	evmhousianw	Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words
EVX	4	100004C4			623	SP	evmra	Initialize Accumulator
EVX	4	1000044F			623	SP	evmwhsmf	Vector Multiply Word High Signed, Modulo, Fractional
EVX	4	1000046F			623	SP	evmwhsmfa	Vector Multiply Word High Signed, Modulo, Fractional to Accumulator
EVX	4	1000044D			623	SP	evmwhsmi	Vector Multiply Word High Signed, Modulo, Integer
EVX	4	1000046D			623	SP	evmwhsmia	Vector Multiply Word High Signed, Modulo, Integer to Accumulator
EVX	4	10000447			624	SP	evmwhssf	Vector Multiply Word High Signed, Saturate, Fractional
EVX	4	10000467			624	SP	evmwhssfa	Vector Multiply Word High Signed, Saturate, Fractional to Accumulator
EVX	4	1000044C			624	SP	evmwhumi	Vector Multiply Word High Unsigned, Modulo, Integer
EVX	4	1000046C			624	SP	evmwhumia	Vector Multiply Word High Unsigned, Modulo, Integer to Accumulator
EVX	4	10000549			625	SP	evmwlsmiaaw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words
EVX	4	100005C9			625	SP	evmwlsmianw	Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words
EVX	4	10000541			625	SP	evmwlssiaaw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words
EVX	4	100005C1			625	SP	evmwlssianw	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words
EVX	4	10000448			626	SP	evmwlumi	Vector Multiply Word Low Unsigned, Modulo, Integer
EVX	4	10000468			626	SP	evmwlumia	Vector Multiply Word Low Unsigned, Modulo, Integer to Accumulator
EVX	4	10000548			626	SP	evmwlumiaaw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	100005C8			626	SP	evmwlumianw	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words
EVX	4	10000540			627	SP	evmwlusiaaw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words
EVX	4	100005C0			627	SP	evmwlusianw	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words
EVX	4	1000045B			627	SP	evmwsmf	Vector Multiply Word Signed, Modulo, Fractional
EVX	4	1000047B			627	SP	evmwsmfa	Vector Multiply Word Signed, Modulo, Fractional to Accumulator
EVX	4	1000055B			628	SP	evmwsmfaa	Vector Multiply Word Signed, Modulo, Fractional and Accumulate
EVX	4	100005DB			628	SP	evmwsmfan	Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative
EVX	4	10000459			628	SP	evmwsmi	Vector Multiply Word Signed, Modulo, Integer
EVX	4	10000479			628	SP	evmwsmia	Vector Multiply Word Signed, Modulo, Integer to Accumulator
EVX	4	10000559			628	SP	evmwsmiaa	Vector Multiply Word Signed, Modulo, Integer and Accumulate
EVX	4	100005D9			628	SP	evmwsmian	Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative
EVX	4	10000453			629	SP	evmwssf	Vector Multiply Word Signed, Saturate, Fractional
EVX	4	10000473			629	SP	evmwssfa	Vector Multiply Word Signed, Saturate, Fractional to Accumulator
EVX	4	10000553			629	SP	evmwssfaa	Vector Multiply Word Signed, Saturate, Fractional and Accumulate
EVX	4	100005D3			630	SP	evmwssfan	Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative
EVX	4	10000458			630	SP	evmwumi	Vector Multiply Word Unsigned, Modulo, Integer
EVX	4	10000478			630	SP	evmwumia	Vector Multiply Word Unsigned, Modulo, Integer to Accumulator
EVX	4	10000558			631	SP	evmwumiaa	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate
EVX	4	100005D8			631	SP	evmwumian	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative
EVX	4	1000021E			631	SP	evnand	Vector NAND
EVX	4	10000209			631	SP	evneg	Vector Negate
EVX	4	10000218			631	SP	evnor	Vector NOR
EVX	4	10000217			632	SP	evor	Vector OR
EVX	4	1000021B			632	SP	evorc	Vector OR with Complement
EVX	4	10000228			632	SP	evrlw	Vector Rotate Left Word
EVX	4	1000022A			633	SP	evrlwi	Vector Rotate Left Word Immediate
EVX	4	1000020C			633	SP	evrndw	Vector Round Word
EVS	4	10000278			633	SP	evsel	Vector Select
EVX	4	10000224			634	SP	evslw	Vector Shift Left Word
EVX	4	10000226			634	SP	evslwi	Vector Shift Left Word Immediate
EVX	4	1000022B			634	SP	evsplatfi	Vector Splat Fractional Immediate
EVX	4	10000229			634	SP	evsplati	Vector Splat Immediate
EVX	4	10000223			634	SP	evsrwis	Vector Shift Right Word Immediate Signed
EVX	4	10000222			634	SP	evsrwiu	Vector Shift Right Word Immediate Unsigned
EVX	4	10000221			635	SP	evsrws	Vector Shift Right Word Signed
EVX	4	10000220			635	SP	evsrwu	Vector Shift Right Word Unsigned
EVX	4	10000321			635	SP	evstdd	Vector Store Double of Double
EVX	31	7C00073E		P	1069	E.PD	evstddepx	Vector Store Double of Double by External PID Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
EVX	4	10000320			635	SP	evstddx	Vector Store Double of Double Indexed
EVX	4	10000325			636	SP	evstdh	Vector Store Double of Four Half Words
EVX	4	10000324			636	SP	evstdhx	Vector Store Double of Four Half Words Indexed
EVX	4	10000323			636	SP	evstdw	Vector Store Double of Two Words
EVX	4	10000322			636	SP	evstdwx	Vector Store Double of Two Words Indexed
EVX	4	10000331			637	SP	evstwhe	Vector Store Word of Two Half Words from Even
EVX	4	10000330			637	SP	evstwhex	Vector Store Word of Two Half Words from Even Indexed
EVX	4	10000335			637	SP	evstwho	Vector Store Word of Two Half Words from Odd
EVX	4	10000334			637	SP	evstwhox	Vector Store Word of Two Half Words from Odd Indexed
EVX	4	10000339			637	SP	evstwwe	Vector Store Word of Word from Even
EVX	4	10000338			637	SP	evstwwex	Vector Store Word of Word from Even Indexed
EVX	4	1000033D			638	SP	evstwwo	Vector Store Word of Word from Odd
EVX	4	1000033C			638	SP	evstwwox	Vector Store Word of Word from Odd Indexed
EVX	4	100004CB			638	SP	evsubfsmiaaw	Vector Subtract Signed, Modulo, Integer to Accumulator Word
EVX	4	100004C3			638	SP	evsubfssiaaw	Vector Subtract Signed, Saturate, Integer to Accumulator Word
EVX	4	100004CA			639	SP	evsubfumiaaw	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word
EVX	4	100004C2			639	SP	evsubfusiaaw	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word
EVX	4	10000204			639	SP	evsubfw	Vector Subtract from Word
EVX	4	10000206			639	SP	evsubifw	Vector Subtract Immediate from Word
EVX	4	10000216			639	SP	evxor	Vector XOR
X	31	7C000774	SR		85	B	extsb[.]	Extend Sign Byte
X	31	7C000734	SR		85	B	extsh[.]	Extend Sign Halfword
X	31	7C0007B4	SR		89	64	extsw[.]	Extend Sign Word
X	63	FC000210			142	FP[R]	fabs[.]	Floating Absolute Value
A	63	FC00002A			144	FP[R]	fadd[.]	Floating Add
A	59	EC00002A			144	FP[R]	fadds[.]	Floating Add Single
X	63	FC00069C			155	FP[R]	fcfid[.]	Floating Convert From Integer Doubleword
X	59	EC00069C			156	FP[R]	fcfids[.]	Floating Convert From Integer Doubleword Single
X	63	FC00079C			156	FP[R]	fcfidu[.]	Floating Convert From Integer Doubleword Unsigned
X	59	EC00079C			157	FP[R]	fcfidus[.]	Floating Convert From Integer Doubleword Unsigned Single
X	63	FC000040			159	FP	fcmpo	Floating Compare Ordered
X	63	FC000000			159	FP	fcmpu	Floating Compare Unordered
X	63	FC000010			142	FP[R]	fcpsgn[.]	Floating Copy Sign
X	63	FC00065C			151	FP[R]	fctid[.]	Floating Convert To Integer Doubleword
X	63	FC00075C			152	FP[R]	fctidu[.]	Floating Convert To Integer Doubleword Unsigned
X	63	FC00075E			153	FP[R]	fctiduz[.]	Floating Convert To Integer Doubleword Unsigned with round toward Zero
X	63	FC00065E			152	FP[R]	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	FC00001C			153	FP[R]	fctiw[.]	Floating Convert To Integer Word
X	63	FC00011C			154	FP[R]	fctiwu[.]	Floating Convert To Integer Word Unsigned
X	63	FC00011E			155	FP[R]	fctiwuz[.]	Floating Convert To Integer Word Unsigned with round toward Zero

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	63	FC00001E			154	FP[R]	fctiwz[.]	Floating Convert To Integer Word with round to Zero
A	63	FC000024			145	FP[R]	fdiv[.]	Floating Divide
A	59	EC000024			145	FP[R]	fdivs[.]	Floating Divide Single
A	63	FC00003A			149	FP[R]	fmadd[.]	Floating Multiply-Add
A	59	EC00003A			149	FP[R]	fmadds[.]	Floating Multiply-Add Single
X	63	FC000090			142	FP[R]	fmr[.]	Floating Move Register
X	63	FC00078C			143	VSX	fmrgew	Floating Merge Even Word
X	63	FC00068C			143	VSX	fmrgow	Floating Merge Odd Word
A	63	FC000038			149	FP[R]	fmsub[.]	Floating Multiply-Subtract
A	59	EC000038			149	FP[R]	fmsubs[.]	Floating Multiply-Subtract Single
A	63	FC000032			145	FP[R]	fmul[.]	Floating Multiply
A	59	EC000032			145	FP[R]	fmuls[.]	Floating Multiply Single
X	63	FC000110			142	FP[R]	fnabs[.]	Floating Negative Absolute Value
X	63	FC000050			142	FP[R]	fneg[.]	Floating Negate
A	63	FC00003E			150	FP[R]	fnmadd[.]	Floating Negative Multiply-Add
A	59	EC00003E			150	FP[R]	fnmadds[.]	Floating Negative Multiply-Add Single
A	63	FC00003C			150	FP[R]	fnmsub[.]	Floating Negative Multiply-Subtract
A	59	EC00003C			150	FP[R]	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	63	FC000030			146	FP[R].in	fre[.]	Floating Reciprocal Estimate
A	59	EC000030			146	FP[R]	fres[.]	Floating Reciprocal Estimate Single
X	63	FC0003D0			158	FP[R].in	frim[.]	Floating Round To Integer Minus
X	63	FC000310			158	FP[R].in	frin[.]	Floating Round To Integer Nearest
X	63	FC000390			158	FP[R].in	frip[.]	Floating Round To Integer Plus
X	63	FC000350			158	FP[R].in	friz[.]	Floating Round To Integer toward Zero
X	63	FC000018			151	FP[R]	frsp[.]	Floating Round to Single-Precision
A	63	FC000034			147	FP[R]	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	59	EC000034			147	FP[R].in	frsqrtes[.]	Floating Reciprocal Square Root Estimate Single
A	63	FC00002E			160	FP[R]	fsel[.]	Floating Select
A	63	FC00002C			146	FP[R]	fsqrt[.]	Floating Square Root
A	59	EC00002C			146	FP[R]	fsqrts[.]	Floating Square Root Single
A	63	FC000028			144	FP[R]	fsub[.]	Floating Subtract
A	59	EC000028			144	FP[R]	fsubs[.]	Floating Subtract Single
X	63	FC000100			148	FP	ftdiv	Floating Test for software Divide
X	63	FC000140			148	FP	ftsqrt	Floating Test for software Square Root
XL	19	4C000224		H	865	S	hrfid	Return From Interrupt Doubleword Hypervisor
X	31	7C0007AC			762	B	icbi	Instruction Cache Block Invalidate
X	31	7C0007BE		P	1067	E.PD	icbiep	Instruction Cache Block Invalidate by External PID
X	31	7C0001CC		M	1124	ECL	icblc	Instruction Cache Block Lock Clear
X	31	7C000118D		M	1121	ECL	icblq.	Instruction Cache Block Lock Query
X	31	7C00002C			762	E	icbt	Instruction Cache Block Touch
X	31	7C0003CC		M	1123	ECL	icbtls	Instruction Cache Block Touch and Lock Set
X	31	7C00078C		P	1239	E.CI	ici	Instruction Cache Invalidate
X	31	7C0007CC		P	1243	E.CD	icread	Instruction Cache Read
A	31	7C00001E			81	B	isel	Integer Select
XL	19	4C00012C			776	B	isync	Instruction Synchronize
X	31	7C000068			777	B	lbarx	Load Byte And Reserve Indexed
X	31	7C000406			822	DS	lbdx	Load Byte with Decoration Indexed
X	31	7C0000BE		P	1059	E.PD	lbepx	Load Byte and Zero by External PID Indexed
D	34	88000000			48	B	lbz	Load Byte and Zero
X	31	7C0006AA		H	876	S	lbzcix	Load Byte and Zero Caching Inhibited Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
D	35	8C000000			48	B	lbzu	Load Byte and Zero with Update
X	31	7C0000EE			48	B	lbzux	Load Byte and Zero with Update Indexed
X	31	7C0000AE			49	B	lbzx	Load Byte and Zero Indexed
DS	58	E8000000			53	64	ld	Load Doubleword
X	31	7C0000A8			782	64	ldarx	Load Doubleword And Reserve Indexed
X	31	7C000428			60	64	ldbrx	Load Doubleword Byte-Reverse Indexed
X	31	7C0006EA		H	876	S	ldcix	Load Doubleword Caching Inhibited Indexed
X	31	7C0004C6			822	DS	lddx	Load Doubleword with Decoration Indexed
X	31	7C00003A		P	1060	E.PD;64	ldepx	Load Doubleword by External PID Indexed
DS	58	E8000001			53	64	ldu	Load Doubleword with Update
X	31	7C00006A			53	64	ldux	Load Doubleword with Update Indexed
X	31	7C00002A			53	64	ldx	Load Doubleword Indexed
D	50	C8000000			135	FP	lfd	Load Floating-Point Double
X	31	7C000646			822	DS	lfddx	Load Floating Doubleword with Decoration Indexed
X	31	7C0004BE		P	1068	E.PD	lfdepx	Load Floating-Point Double by External PID Indexed
DS	57	E4000000			141	FP.out	lfdp	Load Floating-Point Double Pair
X	31	7C00062E			141	FP.out	lfdpx	Load Floating-Point Double Pair Indexed
D	51	CC000000			135	FP	lfdu	Load Floating-Point Double with Update
X	31	7C0004EE			135	FP	lfdux	Load Floating-Point Double with Update Indexed
X	31	7C0004AE			135	FP	lfdx	Load Floating-Point Double Indexed
X	31	7C0006AE			136	FP	lfiwax	Load Floating-Point as Integer Word Algebraic Indexed
X	31	7C0006EE			136	FP	lfiwzx	Load Floating-Point as Integer Word and Zero Indexed
D	48	C0000000			138	FP	lfs	Load Floating-Point Single
D	49	C4000000			138	FP	lfsu	Load Floating-Point Single with Update
X	31	7C00046E			138	FP	lfsux	Load Floating-Point Single with Update Indexed
X	31	7C00042E			138	FP	lfsx	Load Floating-Point Single Indexed
D	42	A8000000			50	B	lha	Load Halfword Algebraic
X	31	7C0000E8			778	B	lharx	Load Halfword And Reserve Indexed Xform
D	43	AC000000			50	B	lhau	Load Halfword Algebraic with Update
X	31	7C0002EE			50	B	lhaux	Load Halfword Algebraic with Update Indexed
X	31	7C0002AE			50	B	lhax	Load Halfword Algebraic Indexed
X	31	7C00062C			59	B	lhbrx	Load Halfword Byte-Reverse Indexed
X	31	7C000446			822	DS	lhdx	Load Halfword with Decoration Indexed
X	31	7C00023E		P	1059	E.PD	lhepx	Load Halfword and Zero by External PID Indexed
D	40	A0000000			49	B	lhz	Load Halfword and Zero
X	31	7C00066A		H	876	S	lhzcix	Load Halfword and Zero Caching Inhibited Indexed
D	41	A4000000			49	B	lhzu	Load Halfword and Zero with Update
X	31	7C00026E			49	B	lhzux	Load Halfword and Zero with Update Indexed
X	31	7C00022E			49	B	lhzx	Load Halfword and Zero Indexed
D	46	B8000000			61	B	lmw	Load Multiple Word
DQ	56	E0000000		P	58	LSQ	lq	Load Quadword
X	31	7C000228			784	LSQ	lqarx	Load Quadword And Reserve Indexed
X	31	7C0004AA			63	MA	lswi	Load String Word Immediate
X	31	7C00042A			63	MA	lswx	Load String Word Indexed
X	31	7C00000E			184	V	lvebx	Load Vector Element Byte Indexed
X	31	7C00004E			181	V	lvehx	Load Vector Element Halfword Indexed
X	31	7C00024E		P	1070	E.PD	lvepx	Load Vector by External PID Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C00020E		P	1070	E.PD	lvepxl	Load Vector by External PID Indexed Last
X	31	7C00008E			181	V	lvewx	Load Vector Element Word Indexed
X	31	7C00000C			186	V	lvsl	Load Vector for Shift Left
X	31	7C00004C			186	V	lvsr	Load Vector for Shift Right
X	31	7C0000CE			182	V	lvx	Load Vector Indexed
X	31	7C0002CE			182	V	lvxl	Load Vector Indexed Last
DS	58	E8000002			52	64	lwa	Load Word Algebraic
X	31	7C000028			777	B	lwarx	Load Word and Reserve Indexed
X	31	7C0002EA			52	64	lwaux	Load Word Algebraic with Update Indexed
X	31	7C0002AA			52	64	lwax	Load Word Algebraic Indexed
X	31	7C00042C			59	B	lwbrx	Load Word Byte-Reverse Indexed
X	31	7C000486			822	DS	lwdx	Load Word with Decoration Indexed
X	31	7C00003E		P	1060	E.PD	lwepx	Load Word and Zero by External PID Indexed
D	32	80000000			51	B	lwz	Load Word and Zero
X	31	7C00062A		H	876	S	lwzcix	Load Word and Zero Caching Inhibited Indexed
D	33	84000000			51	B	lwzu	Load Word and Zero with Update
X	31	7C00006E			51	B	lwzux	Load Word and Zero with Update Indexed
X	31	7C00002E			51	B	lwzx	Load Word and Zero Indexed
XX1	31	7C000498			393	VSX	lxsdx	Load VSR Scalar Doubleword Indexed
XX1	31	7C000098			393	VSX	lxsiwax	Load VSX Scalar as Integer Word Algebraic Indexed
XX1	31	7C000018			394	VSX	lxsiwzx	Load VSX Scalar as Integer Word and Zero Indexed
XX1	31	7C000418			394	VSX	lxsspx	Load VSX Scalar Single-Precision Indexed
XX1	31	7C000698			395	VSX	lxvd2x	Load VSR Vector Doubleword*2 Indexed
XX1	31	7C000298			395	VSX	lxvdsx	Load VSR Vector Doubleword & Splat Indexed
XX1	31	7C000618			396	VSX	lxvw4x	Load VSR Vector Word*4 Indexed
XO	4	10000158			675	LMA	macchw[.]	Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	10000158			675	LMA	macchwo[.]	Multiply Accumulate Cross Halfword to Word Modulo Signed & record OV
XO	4	100001D8			675	LMA	macchws[.]	Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	100001D8			675	LMA	macchwso[.]	Multiply Accumulate Cross Halfword to Word Saturate Signed & record OV
XO	4	10000198			676	LMA	macchwsu[.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned
XO	4	10000198			676	LMA	macchwsuo[.]	Multiply Accumulate Cross Halfword to Word Saturate Unsigned & record OV
XO	4	10000118			676	LMA	macchwu[.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned
XO	4	10000118			676	LMA	macchwuo[.]	Multiply Accumulate Cross Halfword to Word Modulo Unsigned & record OV
XO	4	10000058			677	LMA	machhw[.]	Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	10000058			677	LMA	machhwo[.]	Multiply Accumulate High Halfword to Word Modulo Signed & record OV
XO	4	100000D8			677	LMA	machhws[.]	Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	100000D8			677	LMA	machhwso[.]	Multiply Accumulate High Halfword to Word Saturate Signed & record OV
XO	4	10000098			678	LMA	machhwsu[.]	Multiply Accumulate High Halfword to Word Saturate Unsigned
XO	4	10000098			678	LMA	machhwsuo[.]	Multiply Accumulate High Halfword to Word Saturate Unsigned & record OV

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	4	10000018			678	LMA	machhwu[.]	Multiply Accumulate High Halfword to Word Modulo Unsigned
XO	4	10000018			678	LMA	machhwuo[.]	Multiply Accumulate High Halfword to Word Modulo Unsigned & record OV
XO	4	10000358			679	LMA	maclhw[.]	Multiply Accumulate Low Halfword to Word Modulo Signed
XO	4	10000358			679	LMA	maclhwo[.]	Multiply Accumulate Low Halfword to Word Modulo Signed & record OV
XO	4	100003D8			679	LMA	maclhws[.]	Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	100003D8			679	LMA	maclhwso[.]	Multiply Accumulate Low Halfword to Word Saturate Signed & record OV
XO	4	10000398			680	LMA	maclhwsu[.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned
XO	4	10000398			680	LMA	maclhwsuo[.]	Multiply Accumulate Low Halfword to Word Saturate Unsigned & record OV
XO	4	10000318			680	LMA	maclhwu[.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned
XO	4	10000318			680	LMA	maclhwuo[.]	Multiply Accumulate Low Halfword to Word Modulo Unsigned & record OV
X	31	7C0006AC			790	E	mbar	Memory Barrier
XL	19	4C000000			42	B	mcrf	Move Condition Register Field
X	63	FC000080			161	FP	mcrfs	Move To Condition Register from FPSCR
X	31	7C000400			113	E	mcrxr	Move to Condition Register from XER
XFX	31	7C00025C			44	S	mfbhrbe	Move From Branch History Rolling Buffer
XFX	31	7C000026			108	B	mfcr	Move From Condition Register
XFX	31	7C000286		P	1055	E.DC	mfdcr	Move From Device Control Register
X	31	7C000246			113	E.DC	mfdcrux	Move From Device Control Register User Mode Indexed
X	31	7C000206		P	1055	E.DC	mfdcrx	Move From Device Control Register Indexed
X	63	FC00048E			161	FP[R]	mffs[.]	Move From FPSCR
X	31	7C0000A6		P	888 1055	S E	mfmsr	Move From Machine State Register
XFX	31	7C100026			112	B	mfocrf	Move From One Condition Register Field
XFX	31	7C00029C		O	1257	E.PM	mfpmr	Move from Performance Monitor Register
XFX	31	7C0002A6		O	106 813 885 1054	B	mfspr	Move From Special Purpose Register
X	31	7C0004A6	32	P	927	S	mfsr	Move From Segment Register
X	31	7C000526	32	P	927	S	mfsrin	Move From Segment Register Indirect
XFX	31	7C0002E6			813	S.out	mftb	Move From Time Base
VX	4	10000604			268	V	mfvscr	Move From Vector Status and Control Register
XX1	31	7C000066			109	VSX	mfvsrd	Move From VSR Doubleword
XX1	31	7C0000E6			109	VSX	mfvsrwz	Move From VSR Word and Zero
X	31	7C0001DC		H	1006 1233	S.PC E.PC	msgclr	Message Clear
X	31	7C00015C		P	1007	S.PC	msgclrp	Message Clear Privileged
X	31	7C00019C		H	1006 1233	S.PC E.PC	msgsnd	Message Send
X	31	7C00011C		P	1007	S.PC	msgsndp	Message Send Privileged
XFX	31	7C000120			107	B	mtcrf	Move To Condition Register Fields
XFX	31	7C000386		P	1054	E.DC	mtdcr	Move To Device Control Register
X	31	7C000346			113	E.DC	mtdcrux	Move To Device Control Register User Mode Indexed
X	31	7C000306		P	1054	E.DC	mtdcrx	Move To Device Control Register Indexed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	63	FC00008C			163	FP[R]	mtfsb0[.]	Move To FPSCR Bit 0
X	63	FC00004C			163	FP[R]	mtfsb1[.]	Move To FPSCR Bit 1
XFL	63	FC00058E			162	FP[R]	mtfsf[.]	Move To FPSCR Fields
X	63	FC00010C			162	FP[R]	mtfsfi[.]	Move To FPSCR Field Immediate
X	31	7C000124		P	884 1055	S E	mtmsr	Move To Machine State Register
X	31	7C000164		P	886	S	mtmsrd	Move To Machine State Register Doubleword
XFX	31	7C100120			112	B	mtocrf	Move To One Condition Register Field
XFX	31	7C00039C		O	1257	E.PM	mtpmr	Move To Performance Monitor Register
X	31	7C000126			108	S	mtsle	Move To Split Little Endian
XFX	31	7C0003A6		O	104 884 1053	B	mtspr	Move To Special Purpose Register
X	31	7C0001A4	32	P	926	S	mtsr	Move To Segment Register
X	31	7C0001E4	32	P	926	S	mtsrin	Move To Segment Register Indirect
VX	4	10000644			268	V	mtvscr	Move To Vector Status and Control Register
XX2	31	7C000166			110	VSX	mtvsrd	Move To VSR Doubleword
XX2	31	7C0001A6			110	VSX	mtvsrwa	Move To VSR Word Algebraic
XX2	31	7C0001E6			111	VSX	mtvsrwz	Move To VSR Word and Zero
X	4	10000150			680	LMA	mulchw[.]	Multiply Cross Halfword to Word Signed
X	4	10000110			680	LMA	mulchwu[.]	Multiply Cross Halfword to Word Unsigned
XO	31	7C000092	SR		75	64	mulhd[.]	Multiply High Doubleword
XO	31	7C000012	SR		64	64	mulhdu[.]	Multiply High Doubleword Unsigned
X	4	10000050			681	LMA	mulhhw[.]	Multiply High Halfword to Word Signed
X	4	10000010			681	LMA	mulhhwu[.]	Multiply High Halfword to Word Unsigned
XO	31	7C000096	SR		71	B	mulhw[.]	Multiply High Word
XO	31	7C000016	SR		71	B	mulhwu[.]	Multiply High Word Unsigned
XO	31	7C0001D2	SR		64	64	mulld[.]	Multiply Low Doubleword
XO	31	7C0005D2	SR		64	64	mulldo[.]	Multiply Low Doubleword & record OV
X	4	10000350			681	LMA	mullhw[.]	Multiply Low Halfword to Word Signed
X	4	10000310			681	LMA	mullhwu[.]	Multiply Low Halfword to Word Unsigned
D	7	1C000000			71	B	mulli	Multiply Low Immediate
XO	31	7C0001D6	SR		71	B	mullw[.]	Multiply Low Word
XO	31	7C0005D6	SR		71	B	mullwo[.]	Multiply Low Word & record OV
X	31	7C0003B8	SR		84	B	nand[.]	NAND
XL	19	4C000364		H	867	S	nap	Nap
XO	31	7C0000D0	SR		70	B	neg[.]	Negate
XO	31	7C0004D0	SR		70	B	nego[.]	Negate & record OV
XO	4	1000015C			682	LMA	nmacchw[.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
XO	4	1000015C			682	LMA	nmacchwo[.]	Negative Multiply Accumulate Cross Halfword to Word Modulo Signed & record OV
XO	4	100001DC			682	LMA	nmacchws[.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
XO	4	100001DC			682	LMA	nmacchwso[.]	Negative Multiply Accumulate Cross Halfword to Word Saturate Signed & record OV
XO	4	1000005C			683	LMA	nmachhw[.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed
XO	4	1000005C			683	LMA	nmachhwo[.]	Negative Multiply Accumulate High Halfword to Word Modulo Signed & record OV
XO	4	100000DC			683	LMA	nmachhws[.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed
XO	4	100000DC			683	LMA	nmachhwso[.]	Negative Multiply Accumulate High Halfword to Word Saturate Signed & record OV
XO	4	1000035C			684	LMA	nmaclhw[.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XO	4	1000035C			684	LMA	nmaclhwo[.]	Negative Multiply Accumulate Low Halfword to Word Modulo Signed & record OV
XO	4	100003DC			684	LMA	nmaclhws[.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed
XO	4	100003DC			684	LMA	nmaclhwso[.]	Negative Multiply Accumulate Low Halfword to Word Saturate Signed & record OV
X	31	7C0000F8	SR		85	B	nor[.]	NOR
X	31	7C000378	SR		84	B	or[.]	OR
X	31	7C000338	SR		85	B	orc[.]	OR with Complement
D	24	60000000			82	B	ori	OR Immediate
D	25	64000000			83	B	oris	OR Immediate Shifted
X	31	7C0000F4			87	B	popcntb	Population Count Byte-wise
X	31	7C0003F4			89	64	popcntd	Population Count Doubleword
X	31	7C0002F4			87	B	popcntw	Population Count Words
X	31	7C000174			88	64	prtyd	Parity Doubleword
X	31	7C000134			88	B	prtyw	Parity Word
XL	19	4C000066		P	1041	E	rfci	Return From Critical Interrupt
X	19	4C00004E		P	1042	E.ED	rfdi	Return From Debug Interrupt
XL	19	4C000124			820	S	rfebb	Return from Event Based Branch
XL	19	4C0000CC		P	1043	E.HV	rfgi	Return From Guest Interrupt
XL	19	4C000064		P	1041	E	rfi	Return From Interrupt
XL	19	4C000024		P	864	S	rfid	Return from Interrupt Doubleword
XL	19	4C00004C		P	1042	E	rfmci	Return From Machine Check Interrupt
MDS	30	78000010	SR		95	64	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	78000012	SR		96	64	rldcr[.]	Rotate Left Doubleword then Clear Right
MD	30	78000008	SR		95	64	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	78000000	SR		94	64	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	78000004	SR		94	64	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	7800000C	SR		96	64	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
M	20	50000000	SR		93	B	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21	54000000	SR		91	B	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23	5C000000	SR		92	B	rlwnm[.]	Rotate Left Word then AND with Mask
XL	19	4C0003E4		H	868	S	rvwinkle	Rip Van Winkle
SC	17	44000002			43 863 1040	B	sc	System Call
X	31	7C0007A7	SR	P	923	S	slbfee.	SLB Find Entry ESID
X	31	7C0003E4		P	920	S	slbia	SLB Invalidate All
X	31	7C000364		P	919	S	slbie	SLB Invalidate Entry
X	31	7C000726		P	923	S	slbmfee	SLB Move From Entry ESID
X	31	7C0006A6		P	922	S	slbmfev	SLB Move From Entry VSID
X	31	7C000324		P	921	S	slbmte	SLB Move To Entry
X	31	7C000036	SR		99	64	sld[.]	Shift Left Doubleword
XL	19	4C0003A4		H	868	S	sleep	Sleep
X	31	7C000030	SR		97	B	slw[.]	Shift Left Word
X	31	7C000634	SR		100	64	srad[.]	Shift Right Algebraic Doubleword
XS	31	7C000674	SR		100	64	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	31	7C000630	SR		98	B	sraw[.]	Shift Right Algebraic Word
X	31	7C000670	SR		98	B	srawi[.]	Shift Right Algebraic Word Immediate
X	31	7C000436	SR		99	64	srd[.]	Shift Right Doubleword
X	31	7C000430	SR		97	B	srw[.]	Shift Right Word

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
D	38	98000000			54	B	stb	Store Byte
X	31	7C0007AA		H	877	S	stbcix	Store Byte Caching Inhibited Indexed
X	31	7C00056D			779	B	stbcx.	Store Byte Conditional Indexed
X	31	7C000506			823	DS	stbdx	Store Byte with Decoration Indexed
X	31	7C0001BE		P	1061	E.PD	stbepx	Store Byte by External PID Indexed
D	39	9C000000			54	B	stbu	Store Byte with Update
X	31	7C0001EE			54	B	stbux	Store Byte with Update Indexed
X	31	7C0001AE			54	B	stbx	Store Byte Indexed
DS	62	F8000000			57	64	std	Store Doubleword
X	31	7C000528			60	64	stdbrx	Store Doubleword Byte-Reverse Indexed
X	31	7C0007EA		H	877	S	stdcix	Store Doubleword Caching Inhibited Indexed
X	31	7C0001AD			782	64	stdcx.	Store Doubleword Conditional Indexed & record CR0
X	31	7C0005C6			823	DS	stddx	Store Doubleword with Decoration Indexed
X	31	7C00013A		P	1062	E.PD;64	stdepx	Store Doubleword by External PID Indexed
DS	62	F8000001			57	64	stdu	Store Doubleword with Update
X	31	7C00016A			57	64	stdux	Store Doubleword with Update Indexed
X	31	7C00012A			57	64	stdx	Store Doubleword Indexed
D	54	D8000000			139	FP	stfd	Store Floating-Point Double
X	31	7C000746			823	DS	stfddx	Store Floating Doubleword with Decoration Indexed
X	31	7C0005BE		P	1068	E.PD	stfdepx	Store Floating-Point Double by External PID Indexed
DS	61	F4000000			141	FP.out	stfdp	Store Floating-Point Double Pair
X	31	7C00072E			141	FP.out	stfdpx	Store Floating-Point Double Pair Indexed
D	55	DC000000			139	FP	stfdu	Store Floating-Point Double with Update
X	31	7C0005EE			139	FP	stfdux	Store Floating-Point Double with Update Indexed
X	31	7C0005AE			139	FP	stfdx	Store Floating-Point Double Indexed
X	31	7C0007AE			140	FP	stfiwx	Store Floating-Point as Integer Word Indexed
D	52	D0000000			138	FP	stfs	Store Floating-Point Single
D	53	D4000000			138	FP	stfsu	Store Floating-Point Single with Update
X	31	7C00056E			138	FP	stfsux	Store Floating-Point Single with Update Indexed
X	31	7C00052E			138	FP	stfsx	Store Floating-Point Single Indexed
D	44	B0000000			55	B	sth	Store Halfword
X	31	7C00072C			59	B	sthbrx	Store Halfword Byte-Reverse Indexed
X	31	7C00076A		H	877	S	sthcix	Store Halfword and Zero Caching Inhibited Indexed
X	31	7C0005AD			780	B	sthcx.	Store Halfword Conditional Indexed Xform
X	31	7C000546			823	DS	sthdx	Store Halfword with Decoration Indexed
X	31	7C00033E		P	1061	E.PD	sthepx	Store Halfword by External PID Indexed
D	45	B4000000			55	B	sthu	Store Halfword with Update
X	31	7C00036E			55	B	sthux	Store Halfword with Update Indexed
X	31	7C00032E			55	B	sthx	Store Halfword Indexed
D	47	BC000000			61	B	stmw	Store Multiple Word
DS	62	F8000002		P	58	LSQ	stq	Store Quadword
X	31	7C00016D			785	LSQ	stqcx.	Store Quadword Conditional Indexed and record CR0
X	31	7C0005AA			64	MA	stswi	Store String Word Immediate
X	31	7C00052A			64	MA	stswx	Store String Word Indexed
X	31	7C00010E			184	V	stvebx	Store Vector Element Byte Indexed
X	31	7C00014E			184	V	stvehx	Store Vector Element Halfword Indexed
X	31	7C00064E		P	1071	E.PD	stvepx	Store Vector by External PID Indexed
X	31	7C00060E		P	1071	E.PD	stvepxl	Store Vector by External PID Indexed Last

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C00018E			185	V	stvewx	Store Vector Element Word Indexed
X	31	7C0001CE			182	V	stvx	Store Vector Indexed
X	31	7C0003CE			185	V	stvxl	Store Vector Indexed Last
D	36	90000000			56	B	stw	Store Word
X	31	7C00052C			59	B	stwbrx	Store Word Byte-Reverse Indexed
X	31	7C00072A		H	877	S	stwcix	Store Word and Zero Caching Inhibited Indexed
X	31	7C00012D			781	B	stwcx.	Store Word Conditional Indexed & record CR0
X	31	7C000586			823	DS	stwdx	Store Word with Decoration Indexed
X	31	7C00013E		P	1062	E.PD	stwepx	Store Word by External PID Indexed
D	37	94000000			56	B	stwu	Store Word with Update
X	31	7C00016E			56	B	stwux	Store Word with Update Indexed
X	31	7C00012E			56	B	stwx	Store Word Indexed
XX1	31	7C000598			396	VSX	stxsdx	Store VSR Scalar Doubleword Indexed
XX1	31	7C000118			394	VSX	stxsiwx	Store VSX Scalar as Integer Word Indexed
XX1	31	7C000518			394	VSX	stxsspx	Store VSR Scalar Word Indexed
XX1	31	7C000798			398	VSX	stxvd2x	Store VSR Vector Doubleword*2 Indexed
XX1	31	7C000718			398	VSX	stxvw4x	Store VSR Vector Word*4 Indexed
XO	31	7C000050	SR		67	B	subf[.]	Subtract From
XO	31	7C000010	SR		68	B	subfc[.]	Subtract From Carrying
XO	31	7C000410	SR		68	B	subfco[.]	Subtract From Carrying & record OV
XO	31	7C000110	SR		69	B	subfe[.]	Subtract From Extended
XO	31	7C000510	SR		69	B	subfeo[.]	Subtract From Extended & record OV
D	8	20000000	SR		68	B	subfic	Subtract From Immediate Carrying
XO	31	7C0001D0	SR		69	B	subfme[.]	Subtract From Minus One Extended
XO	31	7C0005D0	SR		69	B	subfmeo[.]	Subtract From Minus One Extended & record OV
XO	31	7C000450	SR		67	B	subfo[.]	Subtract From & record OV
XO	31	7C000190	SR		70	B	subfze[.]	Subtract From Zero Extended
XO	31	7C000590	SR		70	B	subfzeo[.]	Subtract From Zero Extended & record OV
X	31	7C0004AC			786	B	sync	Synchronize
X	31	7C00071D			808	TM	tabort.	Transaction Abort
X	31	7C00065D			809	TM	tabortdc.	Transaction Abort Doubleword Conditional
X	31	7C0006DD			810	TM	tabortdci.	Transaction Abort Doubleword Conditional Immediate
X	31	7C00061D			809	TM	tabortwc.	Transaction Abort Word Conditional
X	31	7C00069D			809	TM	tabortwci.	Transaction Abort Word Conditional Immediate
X	31	7C00051D			806	TM	tbegin.	Transaction Begin
X	31	7C00059C			811	TM	tcheck	Transaction Check
X	31	7C000088			81	64	td	Trap Doubleword
D	2	08000000			81	64	tdi	Trap Doubleword Immediate
X	31	7C00055C			807	TM	tend.	Transaction End
X	31	7C0002E4		H	933	S	tlbia	TLB Invalidate All
X	31	7C000264	64	H	928	S	tlbie	TLB Invalidate Entry
X	31	7C000224	64	P	931	S	tlbiel	TLB Invalidate Entry Local
X	31	7C000024		P	1134	E	tlbilx	TLB Invalidate Local Indexed
X	31	7C000624		P	1132	E	tlbivax	TLB Invalidate Virtual Address Indexed
X	31	7C000764		P	1139	E	tlbre	TLB Read Entry
X	31	7C0006A5		P	1138	E.TWC	tlbsrx.	TLB Search and Reserve Indexed
X	31	7C000724		P	1136	E	tlbsx	TLB Search Indexed
X	31	7C00046C		H PH	934 1141	S E	tlbsync	TLB Synchronize
X	31	7C0007A4		P	1141	E	tlbwe	TLB Write Entry
X	31	7C0007DD			880	TM	trechkpt.	Transaction Recheckpoint

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
X	31	7C00075D			879	TM	treclaim.	Transaction Reclaim
X	31	7C0005DC			810	TM	tsr.	Transaction Suspend or Resume
X	31	7C000008			80	B	tw	Trap Word
D	3	0C000000			80	B	twi	Trap Word Immediate
VX	4	10000140			206	V	vaddcuq	Vector Add & write Carry Unsigned Quadword
VX	4	10000180			202	V	vaddcuw	Vector Add and Write Carry-Out Unsigned Word
VA	4	1000003D			206	V	vaddecuq	Vector Add Extended & write Carry Unsigned Quadword
VA	4	1000003C			206	V	vaddeuqm	Vector Add Extended Unsigned Quadword Modulo
VX	4	1000000A			244	V	vaddfp	Vector Add Single-Precision
VX	4	10000300			202	V	vaddsbs	Vector Add Signed Byte Saturate
VX	4	10000340			202	V	vaddshs	Vector Add Signed Halfword Saturate
VX	4	10000380			203	V	vaddsws	Vector Add Signed Word Saturate
VX	4	10000000			203	V	vaddubm	Vector Add Unsigned Byte Modulo
VX	4	10000200			205	V	vaddubs	Vector Add Unsigned Byte Saturate
VX	4	100000C0			203	V	vaddudm	Vector Add Unsigned Doubleword Modulo
VX	4	10000040			203	V	vadduhm	Vector Add Unsigned Halfword Modulo
VX	4	10000240			205	V	vadduhs	Vector Add Unsigned Halfword Saturate
VX	4	10000100			206	V	vadduqm	Vector Add Unsigned Quadword Modulo
VX	4	10000080			204	V	vadduwm	Vector Add Unsigned Word Modulo
VX	4	10000280			205	V	vadduws	Vector Add Unsigned Word Saturate
VX	4	10000404			238	V	vand	Vector Logical AND
VX	4	10000444			238	V	vandc	Vector Logical AND with Complement
VX	4	10000502			226	V	vavgsb	Vector Average Signed Byte
VX	4	10000542			226	V	vavgsh	Vector Average Signed Halfword
VX	4	10000582			226	V	vavgsw	Vector Average Signed Word
VX	4	10000402			227	V	vavgub	Vector Average Unsigned Byte
VX	4	10000442			227	V	vavguh	Vector Average Unsigned Halfword
VX	4	10000482			227	V	vavguw	Vector Average Unsigned Word
VX	4	1000054C			265	V	vbpermq	Vector Bit Permute Quadword
VX	4	1000054C			265	V	vbpermq	Vector Bit Permute Quadword
VX	4	1000034A			248	V	vcfsx	Vector Convert From Signed Fixed-Point Word To Single-Precision
VX	4	1000030A			248	V	vcfux	Vector Convert From Unsigned Fixed-Point Word To Single-Precision
VX	4	10000508			256	V.Crypto	vcipher	Vector AES Cipher
VX	4	10000509			256	V.Crypto	vcipherlast	Vector AES Cipher Last
VX	4	10000702			263	V	vclzb	Vector Count Leading Zeros Byte
VX	4	100007C2			263	V	vclzd	Vector Count Leading Zeros Doubleword
VX	4	10000742			263	V	vclzh	Vector Count Leading Zeros Halfword
VX	4	10000782			263	V	vclzw	Vector Count Leading Zeros Word
VC	4	100003C6			251	V	vcmpbfp[.]	Vector Compare Bounds Single-Precision
VC	4	100000C6			252	V	vcmpeqfp[.]	Vector Compare Equal To Single-Precision
VC	4	10000006			232	V	vcmpequb[.]	Vector Compare Equal To Unsigned Byte
VC	4	100000C7			233	V	vcmpequd[.]	Vector Compare Equal To Unsigned Doubleword
VC	4	10000046			233	V	vcmpequh[.]	Vector Compare Equal To Unsigned Halfword
VC	4	10000086			233	V	vcmpequw[.]	Vector Compare Equal To Unsigned Word
VC	4	100001C6			252	V	vcmpgefp[.]	Vector Compare Greater Than or Equal To Single-Precision
VC	4	100002C6			253	V	vcmpgtfp[.]	Vector Compare Greater Than Single-Precision
VC	4	10000306			234	V	vcmpgtsb[.]	Vector Compare Greater Than Signed Byte

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VC	4	100003C7			234	V	vcmpgtsd[.]	Vector Compare Greater Than Signed Doubleword
VC	4	10000346			234	V	vcmpgtsh[.]	Vector Compare Greater Than Signed Halfword
VC	4	10000386			235	V	vcmpgtsw[.]	Vector Compare Greater Than Signed Word
VC	4	10000206			236	V	vcmpgtub[.]	Vector Compare Greater Than Unsigned Byte
VC	4	100002C7			236	V	vcmpgtud[.]	Vector Compare Greater Than Unsigned Doubleword
VC	4	10000246			236	V	vcmpgtuh[.]	Vector Compare Greater Than Unsigned Halfword
VC	4	10000286			237	V	vcmpgtuw[.]	Vector Compare Greater Than Unsigned Word
VX	4	100003CA			247	V	vctsxs	Vector Convert From Single-Precision To Signed Fixed-Point Word Saturate
VX	4	1000038A			247	V	vctuxs	Vector Convert From Single-Precision To Unsigned Fixed-Point Word Saturate
VX	4	10000684			238	V	veqv	Vector Equivalence
VX	4	1000018A			254	V	vexptefp	Vector 2 Raised to the Exponent Estimate Single-Precision
VX	4	1000050C			262	V	vgbbd	Vector Gather Bits by Byte by Doubleword
VX	4	100001CA			254	V	vlogefp	Vector Log Base 2 Estimate Single-Precision
VA	4	1000002E			245	V	vmaddfp	Vector Multiply-Add Single-Precision
VX	4	1000040A			246	V	vmaxfp	Vector Maximum Single-Precision
VX	4	10000102			228	V	vmaxsb	Vector Maximum Signed Byte
VX	4	100001C2			228	V	vmaxsd	Vector Maximum Signed Doubleword
VX	4	10000142			228	V	vmaxsh	Vector Maximum Signed Halfword
VX	4	10000182			228	V	vmaxsw	Vector Maximum Signed Word
VX	4	10000002			228	V	vmaxub	Vector Maximum Unsigned Byte
VX	4	100000C2			228	V	vmaxud	Vector Maximum Unsigned Doubleword
VX	4	10000042			228	V	vmaxuh	Vector Maximum Unsigned Halfword
VX	4	10000082			229	V	vmaxuw	Vector Maximum Unsigned Word
VA	4	10000020			218	V	vmhaddshs	Vector Multiply-High-Add Signed Halfword Saturate
VA	4	10000021			218	V	vmhraddshs	Vector Multiply-High-Round-Add Signed Halfword Saturate
VX	4	1000044A			246	V	vminfp	Vector Minimum Single-Precision
VX	4	10000302			230	V	vminsb	Vector Minimum Signed Byte
X	4	100003C2			230	V	vminsd	Vector Minimum Signed Doubleword
VX	4	10000342			230	V	vminsh	Vector Minimum Signed Halfword
VX	4	10000382			231	V	vminsw	Vector Minimum Signed Word
VX	4	10000202			230	V	vminub	Vector Minimum Unsigned Byte
VX	4	100002C2			230	V	vminud	Vector Minimum Unsigned Doubleword
VX	4	10000242			230	V	vminuh	Vector Minimum Unsigned Halfword
VX	4	10000282			231	V	vminuw	Vector Minimum Unsigned Word
VA	4	10000022			219	V	vmladduhm	Vector Multiply-Low-Add Unsigned Halfword Modulo
VX	4	1000078C			196	VSX	vmrgew	Vector Merge Even Word
VX	4	1000000C			194	V	vmrghb	Vector Merge High Byte
VX	4	1000004C			194	V	vmrghh	Vector Merge High Halfword
VX	4	1000008C			195	V	vmrghw	Vector Merge High Word
VX	4	1000010C			194	V	vmrglb	Vector Merge Low Byte
VX	4	1000014C			194	V	vmrglh	Vector Merge Low Halfword
VX	4	1000018C			195	V	vmrglw	Vector Merge Low Word
VX	4	1000068C			196	VSX	vmrgow	Vector Merge Odd Word
VA	4	10000025			220	V	vmsummbm	Vector Multiply-Sum Mixed Byte Modulo
VA	4	10000028			220	V	vmsumshm	Vector Multiply-Sum Signed Halfword Modulo

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VA	4	10000029			221	V	vmsumshs	Vector Multiply-Sum Signed Halfword Saturate
VA	4	10000024			219	V	vmsumubm	Vector Multiply-Sum Unsigned Byte Modulo
VA	4	10000026			221	V	vmsumuhm	Vector Multiply-Sum Unsigned Halfword Modulo
VA	4	10000027			222	V	vmsumuhs	Vector Multiply-Sum Unsigned Halfword Saturate
VX	4	10000308			214	V	vmulesb	Vector Multiply Even Signed Byte
VX	4	10000348			215	V	vmulesh	Vector Multiply Even Signed Halfword
VX	4	10000388			216	V	vmulesw	Vector Multiply Even Signed Word
VX	4	10000208			214	V	vmuleub	Vector Multiply Even Unsigned Byte
VX	4	10000248			215	V	vmuleuh	Vector Multiply Even Unsigned Halfword
VX	4	10000288			216	V	vmuleuw	Vector Multiply Even Unsigned Word
VX	4	10000108			214	V	vmulosb	Vector Multiply Odd Signed Byte
VX	4	10000148			215	V	vmulosh	Vector Multiply Odd Signed Halfword
VX	4	10000188			216	V	vmulosw	Vector Multiply Odd Signed Word
VX	4	10000008			214	V	vmuloub	Vector Multiply Odd Unsigned Byte
VX	4	10000048			215	V	vmulouh	Vector Multiply Odd Unsigned Halfword
VX	4	10000088			216	V	vmulouw	Vector Multiply Odd Unsigned Word
VX	4	10000089			217	V	vmuluwm	Vector Multiply Unsigned Word Modulo
VX	4	10000584			238	V	vnand	Vector NAND
VX	4	10000548			257	V.Crypto	vncipher	Vector AES Inverse Cipher
VX	4	10000549			257	V.Crypto	vncipherlast	Vector AES Inverse Cipher Last
VA	4	1000002F			245	V	vnmsubfp	Vector Negative Multiply-Subtract Single-Precision
VX	4	10000504			239	V	vnor	Vector Logical NOR
VX	4	10000484			239	V	vor	Vector Logical OR
VX	4	10000544			239	V	vorc	Vector OR with Complement
VA	4	1000002B			198	V	vperm	Vector Permute
VA	4	1000002D			261	V.RAID	vpermxor	Vector Permute and Exclusive-OR
VX	4	1000030E			187	V	vpkpx	Vector Pack Pixel
VX	4	100005CE			187	V	vpksdss	Vector Pack Signed Doubleword Signed Saturate
VX	4	1000054E			188	V	vpksdus	Vector Pack Signed Doubleword Unsigned Saturate
VX	4	1000018E			188	V	vpkshss	Vector Pack Signed Halfword Signed Saturate
VX	4	1000010E			189	V	vpkshus	Vector Pack Signed Halfword Unsigned Saturate
VX	4	100001CE			189	V	vpkswss	Vector Pack Signed Word Signed Saturate
VX	4	1000014E			190	V	vpkswus	Vector Pack Signed Word Unsigned Saturate
VX	4	1000044E			190	V	vpkudum	Vector Pack Unsigned Doubleword Unsigned Modulo
VX	4	100004CE			190	V	vpkudus	Vector Pack Unsigned Doubleword Unsigned Saturate
VX	4	1000000E			190	V	vpkuhum	Vector Pack Unsigned Halfword Unsigned Modulo
VX	4	1000008E			191	V	vpkuhus	Vector Pack Unsigned Halfword Unsigned Saturate
VX	4	1000004E			191	V	vpkuwum	Vector Pack Unsigned Word Unsigned Modulo
VX	4	100000CE			191	V	vpkuwus	Vector Pack Unsigned Word Unsigned Saturate
VX	4	10000408			259	V	vpmsumb	Vector Polynomial Multiply-Sum Byte
VX	4	100004C8			259	V	vpmsumd	Vector Polynomial Multiply-Sum Doubleword
VX	4	10000448			260	V	vpmsumh	Vector Polynomial Multiply-Sum Halfword
VX	4	10000488			260	V	vpmsumw	Vector Polynomial Multiply-Sum Word
VX	4	10000703			264	V	vpopcntb	Vector Population Count Byte

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	100007C3			264	V	vpopcntd	Vector Population Count Doubleword
VX	4	10000743			264	V	vpopcnth	Vector Population Count Halfword
VX	4	10000783			264	V	vpopcntw	Vector Population Count Word
VX	4	1000010A			255	V	vrefp	Vector Reciprocal Estimate Single-Precision
VX	4	100002CA			250	V	vrfim	Vector Round to Single-Precision Integer toward -Infinity
VX	4	1000020A			249	V	vrfin	Vector Round to Single-Precision Integer Nearest
VX	4	1000028A			249	V	vrfip	Vector Round to Single-Precision Integer toward +Infinity
VX	4	1000024A			249	V	vrfiz	Vector Round to Single-Precision Integer toward Zero
VX	4	10000004			240	V	vrlb	Vector Rotate Left Byte
VX	4	100000C4			240	V	vrld	Vector Rotate Left Doubleword
VX	4	10000044			240	V	vrlh	Vector Rotate Left Halfword
VX	4	10000084			240	V	vrlw	Vector Rotate Left Word
VX	4	1000014A			255	V	vrsqrtefp	Vector Reciprocal Square Root Estimate Single-Precision
VX	4	100005C8			257	V.Crypto	vsbox	Vector AES S-Box
VA	4	1000002A			199	V	vsel	Vector Select
VX	4	100006C2			258	V.Crypto	vshasigmad	Vector SHA-512 Sigma Doubleword
VX	4	10000682			258	V.Crypto	vshasigmaw	Vector SHA-256 Sigma Word
VX	4	100001C4			200	V	vsl	Vector Shift Left
VX	4	10000104			241	V	vslb	Vector Shift Left Byte
VX	4	100005C4			241	V	vsld	Vector Shift Left Doubleword
VA	4	1000002C			200	V	vsldoi	Vector Shift Left Double by Octet Immediate
VX	4	10000144			241	V	vslh	Vector Shift Left Halfword
VX	4	1000040C			200	V	vslo	Vector Shift Left by Octet
VX	4	10000184			241	V	vslw	Vector Shift Left Word
VX	4	1000020C			197	V	vspltb	Vector Splat Byte
VX	4	1000024C			197	V	vsplth	Vector Splat Halfword
VX	4	1000030C			198	V	vspltisb	Vector Splat Immediate Signed Byte
VX	4	1000034C			198	V	vspltish	Vector Splat Immediate Signed Halfword
VX	4	1000038C			198	V	vspltisw	Vector Splat Immediate Signed Word
VX	4	1000028C			197	V	vspltw	Vector Splat Word
VX	4	100002C4			201	V	vsr	Vector Shift Right
VX	4	10000304			243	V	vsrab	Vector Shift Right Algebraic Byte
VX	4	100003C4			243	V	vsrad	Vector Shift Right Algebraic Doubleword
VX	4	10000344			243	V	vsrah	Vector Shift Right Algebraic Halfword
VX	4	10000384			243	V	vsraw	Vector Shift Right Algebraic Word
VX	4	10000204			242	V	vsrb	Vector Shift Right Byte
VX	4	100006C4			242	V	vsrd	Vector Shift Right Doubleword
VX	4	10000244			242	V	vsrh	Vector Shift Right Halfword
VX	4	1000044C			201	V	vsro	Vector Shift Right by Octet
VX	4	10000284			242	V	vsrw	Vector Shift Right Word
VX	4	10000540			212	V	vsubcuq	Vector Subtract & write Carry Unsigned Quadword
VX	4	10000580			208	V	vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
VA	4	1000003F			212	V	vsubecuq	Vector Subtract Extended & write Carry Unsigned Quadword
VA	4	1000003E			212	V	vsubeuqm	Vector Subtract Extended Unsigned Quadword Modulo
VX	4	1000004A			244	V	vsubfp	Vector Subtract Single-Precision
VX	4	10000700			208	V	vsubsbs	Vector Subtract Signed Byte Saturate

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
VX	4	10000740			208	V	vsubshs	Vector Subtract Signed Halfword Saturate
VX	4	10000780			209	V	vsubsws	Vector Subtract Signed Word Saturate
VX	4	10000400			210	V	vsububm	Vector Subtract Unsigned Byte Modulo
VX	4	10000600			211	V	vsububs	Vector Subtract Unsigned Byte Saturate
VX	4	100004C0			210	V	vsubudm	Vector Subtract Unsigned Doubleword Modulo
VX	4	10000440			210	V	vsubuhm	Vector Subtract Unsigned Halfword Modulo
VX	4	10000640			210	V	vsubuhs	Vector Subtract Unsigned Halfword Saturate
VX	4	10000500			212	V	vsubuqm	Vector Subtract Unsigned Quadword Modulo
VX	4	10000480			210	V	vsubuwm	Vector Subtract Unsigned Word Modulo
VX	4	10000680			211	V	vsubuws	Vector Subtract Unsigned Word Saturate
VX	4	10000688			223	V	vsum2sws	Vector Sum across Half Signed Word Saturate
VX	4	10000708			224	V	vsum4sbs	Vector Sum across Quarter Signed Byte Saturate
VX	4	10000648			224	V	vsum4shs	Vector Sum across Quarter Signed Halfword Saturate
VX	4	10000608			225	V	vsum4ubs	Vector Sum across Quarter Unsigned Byte Saturate
VX	4	10000788			223	V	vsumsws	Vector Sum across Signed Word Saturate
VX	4	1000034E			190	V	vupkhpx	Vector Unpack High Pixel
VX	4	1000020E			193	V	vupkhsb	Vector Unpack High Signed Byte
VX	4	1000024E			193	V	vupkhsh	Vector Unpack High Signed Halfword
VX	4	1000064E			193	V	vupkhsw	Vector Unpack High Signed Word
VX	4	100003CE			192	V	vupklpx	Vector Unpack Low Pixel
VX	4	1000028E			193	V	vupklsb	Vector Unpack Low Signed Byte
VX	4	100002CE			193	V	vupklsh	Vector Unpack Low Signed Halfword
VX	4	100006CE			193	V	vupklsw	Vector Unpack Low Signed Word
VX	4	100004C4			239	V	vxor	Vector Logical XOR
X	31	7C00007C			791	WT	wait	Wait for Interrupt
X	31	7C000106		P	1056	E	wrtee	Write External Enable
X	31	7C000146		P	1057	E	wrteei	Write External Enable Immediate
X	26	68000000				B	xnop	Executed No Operation
X	31	7C000278	SR		84	B	xor[.]	XOR
D	26	68000000			83	B	xori	XOR Immediate
D	27	6C000000			83	B	xoris	XOR Immediate Shifted
XX2	60	F0000564			399	VSX	xsabsdp	VSX Scalar Absolute Value Double-Precision
XX3	60	F0000100			400	VSX	xsadddp	VSX Scalar Add Double-Precision
XX3	60	F0000000			405	VSX	xsaddsp	VSX Scalar Add Single-Precision
XX3	60	F0000158			407	VSX	xscmpodp	VSX Scalar Compare Ordered Double-Precision
XX3	60	F0000118			409	VSX	xscmpudp	VSX Scalar Compare Unordered Double-Precision
XX3	60	F0000580			411	VSX	xscpsgndp	VSX Scalar Copy Sign Double-Precision
XX2	60	F0000424			412	VSX	xscvdpsp	VSX Scalar Convert Double-Precision to Single-Precision
XX2	60	F000042C			413	VSX	xscvdpspn	VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling
XX2	60	F0000560			422	VSX	xscvdpsxds	VSX Scalar Convert Double-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000160			413	VSX	xscvdpsxws	VSX Scalar Convert Double-Precision to Signed Fixed-Point Word Saturate
XX2	60	F0000520			416	VSX	xscvdpuxds	VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000120			418	VSX	xscvdpuxws	VSX Scalar Convert Double-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F0000524			420	VSX	xscvspdp	VSX Scalar Convert Single-Precision to Double-Precision (p=1)

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F000052C			422	VSX	xscvspdpn	Scalar Convert Single-Precision to Double-Precision format Non-signalling
XX2	60	F00005E0			423	VSX	xscvsxddp	VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision
XX2	60	F00004E0			423	VSX	xscvsxdsp	VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision
XX2	60	F00005A0			424	VSX	xscvuxddp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Double-Precision
XX2	60	F00004A0			424	VSX	xscvuxdsp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision
XX3	60	F00001C0			425	VSX	xsdivdp	VSX Scalar Divide Double-Precision
XX3	60	F00000C0			427	VSX	xsdivsp	VSX Scalar Divide Single-Precision
XX3	60	F0000108			429	VSX	xsmaddadp	VSX Scalar Multiply-Add Type-A Double-Precision
XX3	60	F0000008			432	VSX	xsmaddasp	VSX Scalar Multiply-Add Type-A Single-Precision
XX3	60	F0000148			429	VSX	xsmaddmdp	VSX Scalar Multiply-Add Type-M Double-Precision
XX3	60	F0000048			432	VSX	xsmaddmsp	VSX Scalar Multiply-Add Type-M Single-Precision
XX3	60	F0000500			435	VSX	xsmaxdp	VSX Scalar Maximum Double-Precision
XX3	60	F0000540			437	VSX	xsmindp	VSX Scalar Minimum Double-Precision
XX3	60	F0000188			439	VSX	xsmsubadp	VSX Scalar Multiply-Subtract Type-A Double-Precision
XX3	60	F0000088			442	VSX	xsmsubasp	VSX Scalar Multiply-Subtract Type-A Single-Precision
XX3	60	F00001C8			439	VSX	xsmsubmdp	VSX Scalar Multiply-Subtract Type-M Double-Precision
XX3	60	F00000C8			442	VSX	xsmsubmsp	VSX Scalar Multiply-Subtract Type-M Single-Precision
XX3	60	F0000180			445	VSX	xsmuldp	VSX Scalar Multiply Double-Precision
XX3	60	F0000080			447	VSX	xsmulsp	VSX Scalar Multiply Single-Precision
XX2	60	F00005A4			449	VSX	xsnabsdp	VSX Scalar Negative Absolute Value Double-Precision
XX2	60	F00005E4			449	VSX	xsnegdp	VSX Scalar Negate Double-Precision
XX3	60	F0000508			450	VSX	xsnmaddadp	VSX Scalar Negative Multiply-Add Type-A Double-Precision
XX3	60	F0000408			455	VSX	xsnmaddasp	VSX Scalar Negative Multiply-Add Type-A Single-Precision
XX3	60	F0000548			450	VSX	xsnmaddmdp	VSX Scalar Negative Multiply-Add Type-M Double-Precision
XX3	60	F0000448			455	VSX	xsnmaddmsp	VSX Scalar Negative Multiply-Add Type-M Single-Precision
XX3	60	F0000588			458	VSX	xsnmsubadp	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision
XX3	60	F0000488			461	VSX	xsnmsubasp	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
XX3	60	F00005C8			458	VSX	xsnmsubmdp	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision
XX3	60	F00004C8			461	VSX	xsnmsubmsp	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision
XX2	60	F0000124			464	VSX	xsrdpi	VSX Scalar Round to Double-Precision Integer
XX2	60	F00001AC			465	VSX	xsrdpic	VSX Scalar Round to Double-Precision Integer using Current rounding mode
XX2	60	F00001E4			466	VSX	xsrdpim	VSX Scalar Round to Double-Precision Integer toward -Infinity

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F00001A4			466	VSX	xsrdpip	VSX Scalar Round to Double-Precision Integer toward +Infinity
XX2	60	F0000164			467	VSX	xsrdpiz	VSX Scalar Round to Double-Precision Integer toward Zero
XX3	60	F0000168			468	VSX	xsredp	VSX Scalar Reciprocal Estimate Double-Precision
XX3	60	F0000068			469	VSX	xsresp	VSX Scalar Reciprocal Estimate Single-Precision
XX2	60	F0000464			470	VSX	xsrsp	VSX Scalar Round to Single-Precision
XX2	60	F0000128			471	VSX	xsrsqrtedp	VSX Scalar Reciprocal Square Root Estimate Double-Precision
XX2	60	F0000028			472	VSX	xsrsqrtesp	VSX Scalar Reciprocal Square Root Estimate Single-Precision
XX2	60	F000012C			473	VSX	xssqrtdp	VSX Scalar Square Root Double-Precision
XX2	60	F000002C			474	VSX	xssqrtsp	VSX Scalar Square Root Single-Precision
XX3	60	F0000140			475	VSX	xssubdp	VSX Scalar Subtract Double-Precision
XX3	60	F0000040			477	VSX	xssubsp	VSX Scalar Subtract Single-Precision
XX3	60	F00001E8			479	VSX	xstdivdp	VSX Scalar Test for software Divide Double-Precision
XX2	60	F00001A8			480	VSX	xstsqrtdp	VSX Scalar Test for software Square Root Double-Precision
XX2	60	F0000764			480	VSX	xvabsdp	VSX Vector Absolute Value Double-Precision
XX2	60	F0000664			481	VSX	xvabssp	VSX Vector Absolute Value Single-Precision
XX3	60	F0000300			482	VSX	xvadddp	VSX Vector Add Double-Precision
XX3	60	F0000200			486	VSX	xvaddsp	VSX Vector Add Single-Precision
XX3	60	F0000318			488	VSX	xvcmpeqdp	VSX Vector Compare Equal To Double-Precision
XX3	60	F0000718			488	VSX	xvcmpeqdp.	VSX Vector Compare Equal To Double-Precision & record CR6
XX3	60	F0000218			489	VSX	xvcmpeqsp	VSX Vector Compare Equal To Single-Precision
XX3	60	F0000618			489	VSX	xvcmpeqsp.	VSX Vector Compare Equal To Single-Precision & record CR6
XX3	60	F0000398			490	VSX	xvcmpgedp	VSX Vector Compare Greater Than or Equal To Double-Precision
XX3	60	F0000798			490	VSX	xvcmpgedp.	VSX Vector Compare Greater Than or Equal To Double-Precision & record CR6
XX3	60	F0000298			491	VSX	xvcmpgesp	VSX Vector Compare Greater Than or Equal To Single-Precision
XX3	60	F0000698			491	VSX	xvcmpgesp.	VSX Vector Compare Greater Than or Equal To Single-Precision & record CR6
XX3	60	F0000358			492	VSX	xvcmpgtdp	VSX Vector Compare Greater Than Double-Precision
XX3	60	F0000758			492	VSX	xvcmpgtdp.	VSX Vector Compare Greater Than Double-Precision & record CR6
XX3	60	F0000258			493	VSX	xvcmpgtsp	VSX Vector Compare Greater Than Single-Precision
XX3	60	F0000658			493	VSX	xvcmpgtsp.	VSX Vector Compare Greater Than Single-Precision & record CR6
XX3	60	F0000780			494	VSX	xvcpsgndp	VSX Vector Copy Sign Double-Precision
XX3	60	F0000680			494	VSX	xvcpsgnsp	VSX Vector Copy Sign Single-Precision
XX2	60	F0000624			495	VSX	xvcvdpsp	VSX Vector Convert Double-Precision to Single-Precision
XX2	60	F0000760			496	VSX	xvcvdpsxds	VSX Vector Convert Double-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000360			498	VSX	xvcvdpsxws	VSX Vector Convert Double-Precision to Signed Fixed-Point Word Saturate

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F0000720			500	VSX	xvcvdpuxds	VSX Vector Convert Double-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000320			502	VSX	xvcvdpuxws	VSX Vector Convert Double-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F0000724			504	VSX	xvcvspdp	VSX Vector Convert Single-Precision to Double-Precision
XX2	60	F0000660			505	VSX	xvcvspsxds	VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate
XX2	60	F0000260			507	VSX	xvcvspsxws	VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate
XX2	60	F0000620			509	VSX	xvcvspuxds	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Doubleword Saturate
XX2	60	F0000220			511	VSX	xvcvspuxws	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Word Saturate
XX2	60	F00007E0			513	VSX	xvcvsxddp	VSX Vector Convert Signed Fixed-Point Doubleword to Double-Precision
XX2	60	F00006E0			513	VSX	xvcvsxdsp	VSX Vector Convert Signed Fixed-Point Doubleword to Single-Precision
XX2	60	F00003E0			514	VSX	xvcvsxwdp	VSX Vector Convert Signed Fixed-Point Word to Double-Precision
XX2	60	F00002E0			514	VSX	xvcvsxwsp	VSX Vector Convert Signed Fixed-Point Word to Single-Precision
XX2	60	F00007A0			515	VSX	xvcvuxddp	VSX Vector Convert Unsigned Fixed-Point Doubleword to Double-Precision
XX2	60	F00006A0			515	VSX	xvcvuxdsp	VSX Vector Convert Unsigned Fixed-Point Doubleword to Single-Precision
XX2	60	F00003A0			516	VSX	xvcvuxwdp	VSX Vector Convert Unsigned Fixed-Point Word to Double-Precision
XX2	60	F00002A0			516	VSX	xvcvuxwsp	VSX Vector Convert Unsigned Fixed-Point Word to Single-Precision
XX3	60	F00003C0			517	VSX	xvdivdp	VSX Vector Divide Double-Precision
XX3	60	F00002C0			519	VSX	xvdivsp	VSX Vector Divide Single-Precision
XX3	60	F0000308			521	VSX	xvmaddadp	VSX Vector Multiply-Add Type-A Double-Precision
XX3	60	F0000208			521	VSX	xvmaddasp	VSX Vector Multiply-Add Type-A Single-Precision
XX3	60	F0000348			524	VSX	xvmaddmdp	VSX Vector Multiply-Add Type-M Double-Precision
XX3	60	F0000248			524	VSX	xvmaddmsp	VSX Vector Multiply-Add Type-M Single-Precision
XX3	60	F0000700			527	VSX	xvmaxdp	VSX Vector Maximum Double-Precision
XX3	60	F0000600			529	VSX	xvmaxsp	VSX Vector Maximum Single-Precision
XX3	60	F0000740			531	VSX	xvmindp	VSX Vector Minimum Double-Precision
XX3	60	F0000640			533	VSX	xvminsp	VSX Vector Minimum Single-Precision
XX3	60	F0000388			535	VSX	xvmsubadp	VSX Vector Multiply-Subtract Type-A Double-Precision
XX3	60	F0000288			535	VSX	xvmsubasp	VSX Vector Multiply-Subtract Type-A Single-Precision
XX3	60	F00003C8			538	VSX	xvmsubmdp	VSX Vector Multiply-Subtract Type-M Double-Precision
XX3	60	F00002C8			538	VSX	xvmsubmsp	VSX Vector Multiply-Subtract Type-M Single-Precision
XX3	60	F0000380			541	VSX	xvmuldp	VSX Vector Multiply Double-Precision
XX3	60	F0000280			543	VSX	xvmulsp	VSX Vector Multiply Single-Precision
XX2	60	F00007A4			545	VSX	xvnabsdp	VSX Vector Negative Absolute Value Double-Precision
XX2	60	F00006A4			545	VSX	xvnabssp	VSX Vector Negative Absolute Value Single-Precision

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX2	60	F00007E4			546	VSX	xvnegdp	VSX Vector Negate Double-Precision
XX2	60	F00006E4			546	VSX	xvnegsp	VSX Vector Negate Single-Precision
XX3	60	F0000708			547	VSX	xvnmaddadp	VSX Vector Negative Multiply-Add Type-A Double-Precision
XX3	60	F0000608			547	VSX	xvnmaddasp	VSX Vector Negative Multiply-Add Type-A Single-Precision
XX3	60	F0000748			552	VSX	xvnmaddmdp	VSX Vector Negative Multiply-Add Type-M Double-Precision
XX3	60	F0000648			552	VSX	xvnmaddmsp	VSX Vector Negative Multiply-Add Type-M Single-Precision
XX3	60	F0000788			555	VSX	xvnmsubadp	VSX Vector Negative Multiply-Subtract Type-A Double-Precision
XX3	60	F0000688			555	VSX	xvnmsubasp	VSX Vector Negative Multiply-Subtract Type-A Single-Precision
XX3	60	F00007C8			558	VSX	xvnmsubmdp	VSX Vector Negative Multiply-Subtract Type-M Double-Precision
XX3	60	F00006C8			558	VSX	xvnmsubmsp	VSX Vector Negative Multiply-Subtract Type-M Single-Precision
XX2	60	F0000324			561	VSX	xvrdpi	VSX Vector Round to Double-Precision Integer
XX2	60	F00003AC			561	VSX	xvrdpic	VSX Vector Round to Double-Precision Integer using Current rounding mode
XX2	60	F00003E4			562	VSX	xvrdpim	VSX Vector Round to Double-Precision Integer toward -Infinity
XX2	60	F00003A4			562	VSX	xvrdpip	VSX Vector Round to Double-Precision Integer toward +Infinity
XX2	60	F0000364			563	VSX	xvrdpiz	VSX Vector Round to Double-Precision Integer toward Zero
XX3	60	F0000368			564	VSX	xvredp	VSX Vector Reciprocal Estimate Double-Precision
XX3	60	F0000268			565	VSX	xvresp	VSX Vector Reciprocal Estimate Single-Precision
XX2	60	F0000224			566	VSX	xvrspi	VSX Vector Round to Single-Precision Integer
XX2	60	F00002AC			566	VSX	xvrspic	VSX Vector Round to Single-Precision Integer using Current rounding mode
XX2	60	F00002E4			567	VSX	xvrspim	VSX Vector Round to Single-Precision Integer toward -Infinity
XX2	60	F00002A4			567	VSX	xvrspip	VSX Vector Round to Single-Precision Integer toward +Infinity
XX2	60	F0000264			568	VSX	xvrspiz	VSX Vector Round to Single-Precision Integer toward Zero
XX2	60	F0000328			568	VSX	xvrsqrtedp	VSX Vector Reciprocal Square Root Estimate Double-Precision
XX2	60	F0000228			570	VSX	xvrsqrtesp	VSX Vector Reciprocal Square Root Estimate Single-Precision
XX2	60	F000032C			571	VSX	xvsqrtdp	VSX Vector Square Root Double-Precision
XX2	60	F000022C			572	VSX	xvsqrtsp	VSX Vector Square Root Single-Precision
XX3	60	F0000340			573	VSX	xvsubdp	VSX Vector Subtract Double-Precision
XX3	60	F0000240			575	VSX	xvsubsp	VSX Vector Subtract Single-Precision
XX3	60	F00003E8			577	VSX	xvtdivdp	VSX Vector Test for software Divide Double-Precision
XX3	60	F00002E8			578	VSX	xvtdivsp	VSX Vector Test for software Divide Single-Precision
XX2	60	F00003A8			579	VSX	xvtsqrtdp	VSX Vector Test for software Square Root Double-Precision
XX2	60	F00002A8			579	VSX	xvtsqrtsp	VSX Vector Test for software Square Root Single-Precision
XX3	60	F0000410			580	VSX	xxland	VSX Logical AND

Format	Opcode		Mode Dep.[1]	Privilege[1]	Page	Category[1]	Mnemonic	Instruction
	Primary	Base (hex)						
XX3	60	F0000450			580	VSX	xxlandc	VSX Logical AND with Complement
XX3	60	F00005D0			581	VSX	xxleqv	VSX Logical Equivalence
XX3	60	F0000590			581	VSX	xxlnand	VSX Logical NAND
XX3	60	F0000510			582	VSX	xxlnor	VSX Logical NOR
XX3	60	F0000490			583	VSX	xxlor	VSX Logical OR
XX3	60	F0000550			582	VSX	xxlorc	VSX Logical OR with Complement
XX3	60	F00004D0			583	VSX	xxlxor	VSX Logical XOR
XX3	60	F0000090			584	VSX	xxmrghw	VSX Merge High Word
XX3	60	F0000190			584	VSX	xxmrglw	VSX Merge Low Word
XX3	60	F0000050			585	VSX	xxpermdi	VSX Permute Doubleword Immediate
XX4	60	F0000030			585	VSX	xxsel	VSX Select
XX3	60	F0000010			586	VSX	xxsldwi	VSX Shift Left Double by Word Immediate
XX3	60	F0000290			586	VSX	xxspltw	VSX Splat Word

[1] See the key to the mode dependency and privilege columns on page 1484 and the key to the category column in Section 1.3.5 of Book I.

Mode Dependency and Privilege Abbreviations

Except as described below and in Section 1.10.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

Key to Mode Dependency Column

Mode Dep.	Description
CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
SR	The setting of status registers (such as XER and CR0) is mode-dependent.
32	The instruction can be executed only in 32-bit mode.
64	The instruction can be executed only in 64-bit mode.

Key to Privilege Column

Priv.	Description
P	Denotes a privileged instruction.
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for ***mtspr***), depending on the SPR or PMR number.
H	Denotes an instruction that can be executed only in hypervisor state <S,E.HV>
PH	Denotes a hypervisor privileged instruction if Category Embedded.Hypervisor is implemented; otherwise denotes a privileged instruction.
M	Denotes an instruction that is treated as privileged or nonprivileged, depending on the value of the UCLE bit in the MSR.

Index

E

F

G

H

I

K

L

M

S

T

U

V

W

X

Z

Last Page - End of Document