Power Architecture
64-Bit ELF V2 ABI Specification

OpenPOWER ABI for Linux Supplement

Advance

21 July 2014
Version 1.0

Notices

This document is based on the following publications:

Power Architecture® 32-bit Application Binary Interface Supplement 1.0 for Linux® by Ryan S. Arnold, Greg Davis, Brian Deitrich, Michael Eager, Emil Medve, Steven J. Munroe, Joseph S. Myers, Steve Papacharalambous, Anmol P. Paralkar, Katherine Stewart, and Edmar Wienskoski

Power Architecture 64-bit Application Binary Interface V2.0 Supplement – OpenPOWER for Linux by Michael Gschwind
2.0 Edition
Published December 2013

Power Architecture 32-bit Application Binary Interface Supplement 1.0 by Ryan S. Arnold, Greg Davis, Brian Deitrich, Michael Eager, Emil Medve, Steven J. Munroe, Joseph S. Myers, Steve Papacharalambous, Anmol P. Paralkar, Katherine Stewart, and Edmar Wienskoski
1.0 Edition
Published April 19, 2011

Portions of *Power Architecture 32-bit Application Binary Interface Supplement 1.0* are derived from the *64-bit PowerPC® ELF Application Binary Interface Supplement 1.8*, originally written by Ian Lance Taylor under contract for IBM, with later revisions by: David Edelsohn, Torbjorn Granlund, Mark Mendell, Kristin Thomas, Alan Modra, Steve Munroe, and Chris Lorenze.

The Thread Local Storage section of this document is an original contribution of IBM written by Alan Modra and Steven Munroe.

Printed in the United States of America July 2014

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®**.

Version 1.0
21 July 2014

Contents

List of Figures

List of Tables

Revision Log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was modified from the previous release of this document.

Revision Date	Pages	Description
21 July 2014	—	Version 1.0, Initial release.

About this Document

This specification defines the OpenPOWER ELF V2 application binary interface (ABI). This ABI is derived from and represents the first major update to the Power ABI since the original release of the IBM® RS/6000® ABI. It was developed to make extensive use of new functions available in OpenPOWER-compliant processors. It expects an OpenPOWER-compliant processor to implement at least Power ISA V2.07 with all OpenPOWER Architecture instruction categories as well as OpenPOWER-defined implementation characteristics for some implementation-specific features.

Specifically, to use this ABI and ABI-compliant programs, OpenPOWER-compliant processors must implement the following categories:

- Base
- 64-Bit
- Server (subject to system-level requirements)
- Floating-Point
- Floating-Point.Record
- Load/Store Quadword x2
- Store Conditional Page Mobility (subject to system-level requirements)
- Stream
- Transactional Memory
- Vector
- Vector.Little-Endian
- Vector-Scalar

For more information about these categories, see “Categories” in Book I of *Power ISA*, version 2.07.

The OpenPOWER ELF V2 ABI is intended for use in little- and big-endian environments.

1. Introduction

The Executable and Linkable Format (ELF) defines a linking interface for executables and shared objects in two parts.

- The first part is the generic System V ABI (http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/normativerefs.html#NORMATIVEREFSSECT).
- The second part is a processor-specific supplement.

This document, the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, is the OpenPOWER-compliant processor-specific supplement for use with ELF V2 on 64-bit IBM Power Architecture® systems. This is not a complete System V ABI supplement because it does not define any library interfaces.

This document establishes both big-endian and little-endian application binary interfaces (see *Section 2.1.2.1 Byte Ordering* on page 19). OpenPOWER-compliant processors in the 64-bit Power Architecture can execute in either big-endian or little-endian mode. Executables and executable-generated data (in general) that subscribe to either byte ordering are not portable to a system running in the other mode.

Note: This ABI specification does not address little-endian byte ordering before Power ISA 2.03.

The OpenPOWER ELF V2 ABI is not the same as either the Power Architecture 32-bit ABI supplement or the 64-bit IBM PowerPC® ELF ABI (ELF V1).

The Power Architecture 64-bit OpenPOWER ELF V2 ABI supplement is intended to use the same structural layout now followed in practice by other processor-specific ABIs.

1.1 Reference Documentation

The archetypal ELF ABI is described by the System V ABI. Supersessions and addenda that are specific to OpenPOWER ELF V2 Power Architecture (64-bit) processors are described in this document.

The following documents are complementary to this document and equally binding:

- *Power Instruction Set Architecture,* Version 2.07, IBM, 2013. http://www.power.org
- *DWARF Debugging Information Format,* Version 4, DWARF Debugging Information Format Workgroup, 2010. http://dwarfstd.org/Dwarf4Std.php
- *ISO/IEC 9899:2011: Programming languages—C*. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=57853
- Itanium C++ ABI: Exception Handling. Rev 1.22, CodeSourcery, 2001. http://www.codesourcery.com/public/cxx-abi/abi-eh.html
- *ISO/IEC TR 24732:2009 - Programming languages, their environments and system software interfaces - Extension for the programming language C to support decimal floating-point arithmetic*, ISO/IEC, January 05, 2009. Available from ISO. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_tc_browse.htm?commid=45202
- *ELF Handling for Thread-Local Storage*, Version 0.20, Ulrich Drepper, Red Hat Inc., December 21, 2005. http://people.redhat.com/drepper/tls.pdf

The following documents are of interest for their historical information but are not normative in any way.

- *64-bit PowerPC ELF Application Binary Interface Supplement 1.9.* http://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi.html
- *IBM PowerOpen™ ABI Application Binary Interface Big-Endian 32-Bit Hardware Implementation.* ftp://www.sourceware.org/pub/binutils/ppc-docs/ppc-poweropen/
- *Power Architecture 32-bit ABI Supplement 1.0 Embedded/Linux/Unified.* https://www.power.org/documentation/power-architecture-32-bit-abi-supplement-1-0-embeddedlinuxunified/
- *ALTIVEC PIM: AltiVec (TM) Technology Programming Interface Manual*, Freescale Semiconductor, 1999. http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf
- ELF Assembly User's Guide, Fourth edition, IBM, 2000. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/109917A251EFD64C872569D900656D07/$file/assem_um.pdf

2. Low-Level System Information

2.1 Machine Interface

The machine interface describes the specific use of the Power ISA 64-bit features to implement the ELF ABI version 2.

2.1.1 Processor Architecture

This ABI is predicated on, at a minimum, Power ISA version 2.07 and contains additional implementation characteristics.

All OpenPOWER instructions that are defined by the Power Architecture can be assumed to be implemented and to work as specified. ABI-conforming implementations must provide these instructions through software emulation if they are not provided by the OpenPOWER-compliant processor.

In addition, the instruction specification must meet additional implementation-defined specifics as commonly required by the OpenPOWER specification.

OpenPOWER-compliant processors may support additional instructions beyond the published Instruction Set Architecture (ISA) and may include optional Power Architecture instructions.

This ABI does not explicitly impose any performance constraints on systems.

2.1.2 Data Representation

2.1.2.1 Byte Ordering

The following standard data formats are recognized:

- 8-bit byte
- 16-bit halfword
- 32-bit word
- 64-bit doubleword
- 128-bit quadword

In big-endian byte ordering, the most-significant byte is located in the lowest addressed byte position in memory (byte 0). This byte ordering is alternately referred to as most-significant byte (MSB) ordering.

In little-endian byte ordering, the least-significant byte is located in the lowest addressed byte position in memory (byte 0). This byte ordering is alternately referred to as least-significant byte (LSB) ordering.

A specific OpenPOWER-compliant processor implementation must state which type of byte ordering is to be used.

MSR[LE|SLE]: Although it may be possible to modify the active byte ordering of an application process that uses application-accessible configuration controls or that uses system calls on some systems, applications that change active byte ordering during the course of execution do not conform to this ABI.

Table 2-2, *Table 2-3*, *Table 2-4*, and *Table 2-5* show the conventions assumed in big-endian and little-endian byte ordering at the bit and byte levels. These conventions are applied to integer and floating-point data types. As shown in *Table 2-1*, byte numbers are indicated in the upper corners, and bit numbers are indicated in the lower corners. Little-endian numbers are indicated on the right side; big-endian numbers are indicated on the left side.

Table 2-1. Bit and Byte Numbering Example

Big-Endian Byte Number	Little-Endian Byte Number
Big-Endian Bit Number	Little-Endian Bit Number

Table 2-2. Bit and Byte Numbering in Halfwords

0 1 MSB 0 7	1 0 LSB 8 15

Table 2-3. Bit and Byte Numbering in Words

0 3 MSB 0 7	1 2 8 15	2 1 16 23	3 0 LSB 24 31

Table 2-4. Bit and Byte Numbering in Doublewords

0 7 MSB 0 7	1 6 8 15	2 5 16 23	3 4 24 31
4 3 32 39	5 2 40 47	6 1 48 55	7 0 LSB 56 63

Table 2-5. Bit and Byte Numbering in Quadwords

0 15 MSB 0 7	1 14 8 15	2 13 16 23	3 12 24 31
4 11 32 39	5 10 40 47	6 9 48 55	7 8 56 63
8 7 64 71	9 6 72 79	10 5 80 87	11 4 88 95
12 3 96 103	13 2 104 111	14 1 112 119	15 0 LSB 120 127

Note: In the Power ISA, the figures are generally only shown in big-endian byte order. The bits in this data format specification are numbered from left to right (MSB to LSB).

FPSCR Formats: As of Power ISA version 2.05, the FPSCR is extended from 32 bits to 64 bits. The fields of the original 32-bit FPSCR are now held in bits 32 - 63 of the 64-bit FPSCR. The assembly instructions that operate upon the 64-bit FPSCR have either a W instruction field added to select the operative word for the instruction (for example, **mtfsfi**) or the instruction is extended to operate upon the entire 64-bit FPSCR, (for example, **mffs**). Fields of the FPSCR that represent 1 or more bits are referred to by field number with an indication of the operative word rather than by bit number.

2.1.2.2 Fundamental Types

The following tables map the data format specifications described in the Power ISA for ISO C scalar types. Each scalar type has a required alignment, which is indicated in the Alignment column. Use of these types in data structures must follow the alignment specified in the order encountered to ensure consistent mapping. When using variables individually, more strict alignment may be imposed if it has optimization benefits.

Table 2-6. Fundamental Types (Page 1 of 2)

Type	ISO C Types	sizeof	Alignment	Description
Boolean	_Bool	1	Byte	Boolean
Character	char	1	Byte	Unsigned byte
	unsigned char			
	signed char	1	Byte	Signed byte
Enumeration	signed enum	4	Word	Signed word
	unsigned enum	4	Word	Unsigned word
Integral	int	4	Word	Signed word
	signed int			
	unsigned int	4	Word	Unsigned word
	long int	8	Doubleword	Signed doubleword
	signed long int	8	Doubleword	Signed doubleword
	unsigned long int	8	Doubleword	Unsigned doubleword
	long long int	8	Doubleword	Signed doubleword
	signed long long int			
	unsigned long long int	8	Doubleword	Unsigned doubleword
	short int	2	Halfword	Signed halfword
	signed short int			
	unsigned short int	2	Halfword	Unsigned halfword
	__int128	16	Quadword	Signed quadword
	signed __int128			
	unsigned __int128	16	Quadword	Unsigned quadword
Pointer	any *	8	Doubleword	Data pointer
	any (*) ()			Function pointer

Table 2-6. Fundamental Types (Page 2 of 2)

Type	ISO C Types	sizeof	Alignment	Description
Binary Floating-Point	float	4	Word	Single-precision float
	double	8	Doubleword	Double-precision float
	long double	16	Quadword	Extended- or quad-precision float

Notes:

- A NULL pointer has all bits set to zero.
- A Boolean value is represented as a byte with a value of 0 or 1. If a byte with a value other than 0 or 1 is evaluated as a boolean value (for example, through the use of unions), the behavior is undefined.
- If an enumerated type contains a negative value, it is compatible with and has the same representation and alignment as int. Otherwise, it is compatible with and has the same representation and alignment as an unsigned int.
- For each real floating-point type, there is a corresponding imaginary type with the same size and alignment and there is a corresponding complex type. The complex type has the same alignment as the real type and is twice the size; the representation is the real part followed by the imaginary part.

Table 2-7. Vector Types

Type	Power SIMD C Types	sizeof	Alignment	Description
vector-128	vector unsigned char	16	Quadword	Vector of 16 unsigned bytes
	vector signed char	16	Quadword	Vector of 16 signed bytes
	vector bool char	16	Quadword	Vector of 16 bytes with a value of either 0 or 2^8 - 1.
	vector unsigned short	16	Quadword	Vector of 8 unsigned halfwords
	vector signed short	16	Quadword	Vector of 8 signed halfwords
	vector bool short	16	Quadword	Vector of 8 halfwords with a value of either 0 or 2^{16} - 1.
	vector unsigned int	16	Quadword	Vector of 4 unsigned words
	vector signed int	16	Quadword	Vector of 4 signed words
	vector bool int	16	Quadword	Vector of 4 words with a value of either 0 or 2^{32} - 1,
	vector unsigned long vector unsigned long long	16	Quadword	Vector of 2 unsigned doublewords
	vector signed long vector signed long long	16	Quadword	Vector of 2 signed doublewords
	vector bool long vector bool long long	16	Quadword	Vector of 2 doublewords with a value of either 0 or 2^{64} - 1.
	vector float	16	Quadword	Vector of 4 single-precision floats
	vector double	16	Quadword	Vector of 2 double-precision doubles
	vector unsigned __int128	16	Quadword	Vector of 1 unsigned quadword
	vector signed __int128	16	Quadword	Vector of 1 signed quadword

Decimal Floating-Point (ISO TR 24732 Support)

The decimal floating-point data type is used to specify variables corresponding to the IEEE 754-2008 densely packed, decimal floating-point format.

Table 2-8. Decimal Floating-Point Types

Type	ISO TR 24732 C Types	sizeof	Alignment	Description
Decimal Floating-Point	_Decimal32	4	Word	Single-precision decimal float
	_Decimal64	8	Doubleword	Double-precision decimal float
	_Decimal128	16	Quadword	Quad-precision decimal float

IBM EXTENDED PRECISION

Table 2-9. IBM EXTENDED PRECISION Type

Type	ISO C Types	sizeof	Alignment	Description
IBM EXTENDED PRECISION	long double	16	Quadword	Two double-precision floats

IEEE BINARY 128 EXTENDED PRECISION

Table 2-10. IEEE BINARY 128 EXTENDED PRECISION Type

Type	ISO C Types	sizeof	Alignment	Description	Notes
IEEE BINARY 128 EXTENDED PRECISION	long double	16	Quadword	IEEE 128-bit quad-precision float	1
IEEE BINARY 128 EXTENDED PRECISION	_float128	16	Quadword	IEEE 128-bit quad-precision float	1

1. Phased in.

IBM EXTENDED PRECISION && IEEE BINARY 128 EXTENDED PRECISION

Availability of the long double data type is subject to conformance to a long double standard where the IBM EXTENDED PRECISION format and the IEEE BINARY 128 EXTENDED PRECISION format are mutually exclusive.

IEEE BINARY 128 EXTENDED PRECISION || IBM EXTENDED PRECISION

This ABI provides the following choices for implementation of long double in compilers and systems. The preferred implementation for long double is the IEEE 128-bit quad-precision binary floating-point type.

- IEEE BINARY 128 EXTENDED PRECISION
  - Long double is implemented as an IEEE 128-bit quad-precision binary floating-point type in accordance with the applicable IEEE floating-point standards.

  - Support is provided for all IEEE standard features.
  - IEEE128 quad-precision values are passed in VMX parameter registers.
  - With some compilers, _float128 or _Float128 can be used to access IEEE128 independent of the floating-point representation chosen for the long double ISO C type. However, this is not part of the C standard.

- IBM EXTENDED PRECISION
  - Support is provided for the IBM EXTENDED PRECISION format. In this format, double-precision numbers with different magnitudes that do not overlap provide an effective precision of 106 bits or more, depending on the value. The high-order double-precision value (the one that comes first in storage) must have the larger magnitude. The high-order double-precision value must equal the sum of the two values, rounded to nearest double (the Linux convention, unlike AIX).
  - IBM EXTENDED PRECISION form provides the same range as double precision (about 10^{-308} to 10^{308}) but more precision (a variable amount, about 31 decimal digits or more).
  - As the absolute value of the magnitude decreases (near the denormal range), the precision available in the low-order double also decreases.
  - When the value represented is in the subnormal or denormal range, this representation provides no more precision than 64-bit (double) floating-point.
  - The actual number of bits of precision can vary. If the low-order part is much less than one unit of least precision (ULP) of the high-order part, significant bits (all 0s) are implied between the significands of high-order and low-order numbers. Some algorithms that rely on having a fixed number of bits in the significand can fail when using extended precision.

  This implementation differs from the IEEE 754 Standard in the following ways:
  - The software support is restricted to round-to-nearest mode. Programs that use extended precision must ensure that this rounding mode is in effect when extended-precision calculations are performed.
  - This implementation does not fully support the IEEE special numbers NaN and INF. These values are encoded in the high-order double value only. The low-order value is not significant, but the low-order value of an infinity must be positive or negative zero.
  - This implementation does not support the IEEE status flags for overflow, underflow, and other conditions. These flags have no meaning in this format.

2.1.2.3 Aggregates and Unions

The following rules for aggregates (structures and arrays) and unions apply to their alignment and size:

- The entire aggregate or union must be aligned to its most strictly aligned member, which corresponds to the member with the largest alignment, including flexible array members.
- Each member is assigned the lowest available offset that meets the alignment requirements of the member. Depending on the previous member, internal padding can be required.
- The entire aggregate or union must have a size that is a multiple of its alignment. Depending on the last member, tail padding may be required.

For *Figure 2-1* through *Figure 2-5*, the big-endian byte offsets are located in the upper left corners, and the little-endian byte offsets are located in the upper right corners.

Figure 2-1. Structure Smaller than a Word

```
struct { char c; };
```

byte aligned, sizeof is 1

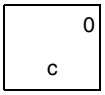
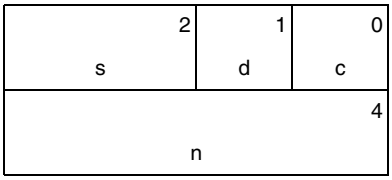


Figure 2-2. Structure with No Padding

```
struct { char c; char d; short s; int n; };
```

word-aligned, sizeof is 8

little endian



big endian

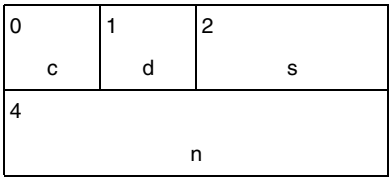
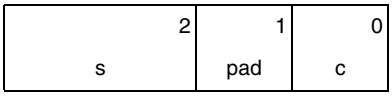


Figure 2-3. Structure with Internal Padding

```
struct { char c; short s; };
```

halfword-aligned, sizeof is 4

little endian



big endian

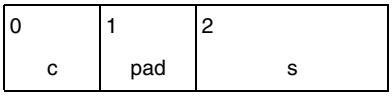
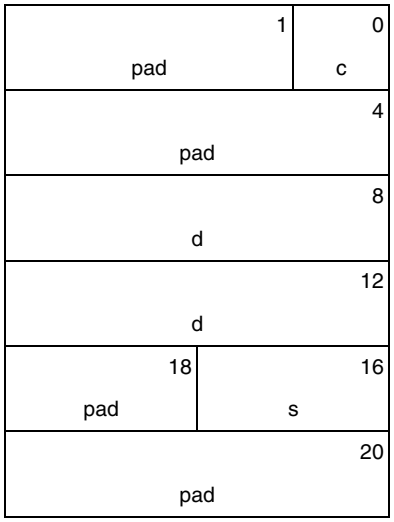


Figure 2-4. Structure with Internal and Tail Padding

```
struct { char c; double d; short s; };
```

doubleword-aligned, sizeof is 24

little endian



big endian

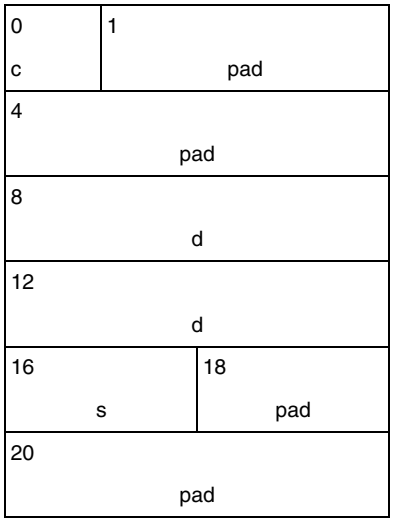
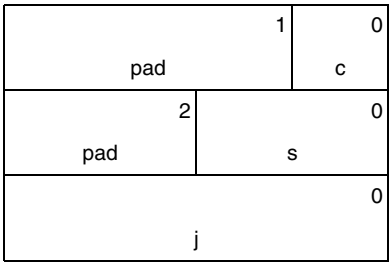


Figure 2-5. Union Allocation

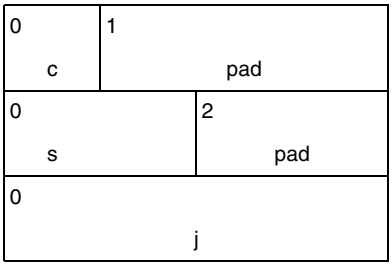
```
union { char c; short s; int j; };
```

word-aligned, sizeof is 4

little endian



big endian



2.1.2.4 Bit Fields

Bit fields can be present in definitions of C structures and unions. These bit fields define whole objects within the structure or union where the number of bits in the bit field is specified.

In *Table 2-11*, a signed range goes from -2^{w-1} to $2^{w-1} - 1$ and an unsigned range goes from 0 to $2^{w} - 1$.

Table 2-11. Bit Field Types (Page 1 of 2)

Bit Field Type	Width (w)
_Bool	1
signed char	1 - 8
unsigned char	
signed short	1 - 16
unsigned short	
signed int	1 - 32
unsigned int	
enum	

Table 2-11. Bit Field Types (Page 2 of 2)

Bit Field Type	Width (w)
signed long	1 - 64
unsigned long	
signed long long	
unsigned long long	
signed __int128	1 - 128
unsigned __int128	

Bit fields can be a signed or unsigned of type short, int, long, or long long. However, bit fields shall have the same range for each corresponding type. For example, signed short must have the same range as unsigned short. All members of structures and unions, including bit fields, must comply with the size and alignment rules. The following list of additional size and alignment rules apply to bit fields:

- The allocation of bit fields is determined by the system endianness. For little-endian implementations, the bit allocation is from the least-significant (right) end to the most-significant (left) end. The reverse is true for big-endian implementations; the bit allocation is from most-significant (left) end to the least-significant (right) end.
- A bit field cannot cross its unit boundary; it must occupy part or all or the storage unit allocated for its declared type.
- If there is enough space within a storage unit, bit fields must share the storage unit with other structure members, including members that are not bit fields. Clearly, all the structure members occupy different parts of the storage unit.
- The types of unnamed bit fields have no effect on the alignment of a structure or union. However, the offsets of an individual bit field's member must comply with the alignment rules. An unnamed bit field of zero width causes sufficient padding (possibly none) to be inserted for the next member, or the end of the structure if there are no more nonzero width members, to have an offset from the start of the structure that is a multiple of the size of the declared type of the zero-width member.

The byte offsets for structure and union members are shown in the following examples. As shown in *Table 2-1* on page 20, the little-endian byte offsets are given in the upper right corners, and the big-endian byte offsets are given in the upper left corners. The bit numbers are given in the lower corners.

Table 2-12. Bit Numbering for 0x01020304

0 3	1 2	2 1	3 0
01	02	03	04
0 7	8 15	16 23	24 31

Figure 2-6. Simple Bit Field Allocation

```
struct {int j : 5; int k : 6; int m : 7; };
```

word-aligned, sizeof is 4

little endian

pad 0 13	m 14 20	k 21 26	0 j 27 31

big endian

0 j 0 4	k 5 10	m 11 17	pad 18 31

Figure 2-7. Bit Field Allocation with Boundary Alignment

```
struct {
    short s : 9;
    int j : 9;
    char c;
    short t : 9;
    short u : 9;
    char d;
};
```

word-aligned, sizeof is 12

little endian

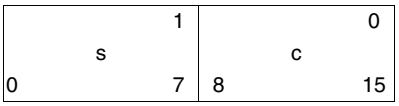
3 c 0 7	pad 8 13	j 14 22	0 s 23 31
pad 0 6	u 7 15	pad 16 22	5 t 23 31
pad 0 23			9 8 d 24 31

Figure 2-8. Bit Field Allocation with Storage Unit Sharing

```
struct { char c; short s : 8; };
```

halfword-aligned, sizeof is 2

little endian



big endian

Figure 2-9. Bit Field Allocation in a Union

```
union { char c; short s : 8; };
```

halfword-aligned, sizeof is 2

little endian

big endian

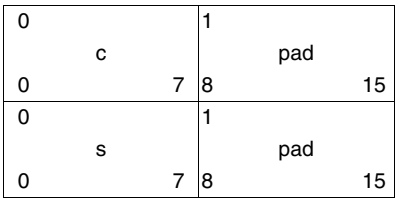


Figure 2-10. Bit Field Allocation with Unnamed Bit Fields

```
struct { char c;
    int : 0;
    char d;
    short : 9;
    char e;
};
```

byte aligned, sizeof is 9

little endian

:0 0 23			1 c 24 31
pad 0 6	6 :9 7 15	pad 16 23	4 d 24 31
			8 e 24 31

big endian

0 c 0 7	1 :0 8 31		
4 d 0 7	pad 8 15	6 :9 16 24	pad 25 31
8 e 0 7			

Note: In *Figure 2-10*, the alignment of the structure is not affected by the unnamed short and int fields. The named members are aligned relative to the start of the structure. However, it is possible that the alignment of the named members is not on optimum boundaries in memory. For instance, in an array of the structure in *Figure 2-10*, the d members will not all be on 4-byte (integer) boundaries.

2.2 Function Calling Sequence

The standard sequence for function calls is outlined in this section. The layout of the stack frame, the parameter passing convention, and the register usage are also described in this section. Standard library functions use these conventions, except as documented for the register save and restore functions.

The conventions given in this section are adhered to by C programs. For more information about the implementation of C, see *Section 2.3 Coding Examples* on page 57.

Note: While it is recommended that all functions use the standard calling sequence, the requirements of the standard calling sequence are only applicable to global functions. Different calling sequences and conventions can be used by local functions that cannot be reached from other compilation units, if they comply with the stack back trace requirements. Some tools may not work with alternate calling sequences and conventions.

2.2.1 Registers

Programs and compilers may freely use all registers except those reserved for system use. The system signal handlers are responsible for preserving the original values upon return to the original execution path. Signals that can interrupt the original execution path are documented in the *System V Interface Definition (SVID)*.

The tables in *Section 2.2.1.1 Register Roles* on page 33 give an overview of the registers that are global during program execution. The tables use three terms to describe register preservation rules:

nonvolatile	A caller can expect that the contents of all registers marked nonvolatile are valid after control returns from a function call. A callee shall save the contents of all registers marked nonvolatile before modification. The callee must restore the contents of all such registers before returning to its caller.
volatile	A caller cannot trust that the contents of registers marked volatile have been preserved across a function call. A callee need not save the contents of registers marked volatile before modification.
limited-access	The contents of registers marked limited-access have special preservation rules. These registers have mutability restricted to certain bit fields as defined by the Power ISA. The individual bits of these bit fields are defined by this ABI to be limited-access. Under normal conditions, a caller can expect that these bits have been preserved across a function call. Under the special conditions indicated in *Section 2.2.1.2 Limited-Access Bits* on page 38, a caller shall expect that these bit will have changed across function calls even if they have not. A callee may only permanently modify these bits without preserving the state upon entrance to the function if the callee satisfies the special conditions indicated in *Section 2.2.1.2*. Otherwise, these bits must be preserved before modification and restored before returning to the caller.
reserved	The contents of registers marked reserved are for exclusive use of system functions, including the ABI. In limited circumstances, a program or program libraries may set or query such registers, but only when explicitly allowed in this document.

2.2.1.1 Register Roles

In the 64-bit OpenPOWER Architecture, there are always 32 general-purpose registers, each 64 bits wide. Throughout this document the symbol rN is used, where N is a register number, to refer to general-purpose register N.

Table 2-13. Register Roles

Register	Preservation Rules	Purpose
r0	Volatile	Optional in function linkage. Used in function prologues.
r1	Nonvolatile	Stack frame pointer.
r2	Nonvolatile[1]	TOC pointer.
r3 - r10	Volatile	Parameter and return values.
r11	Volatile	Optional use in function linkage. Used as an environment pointer in languages that require environment pointers.
r12	Volatile	Optional use in function linkage. Function entry address at global entry point.
r13	Reserved	Thread pointer (see *Section 3.7 Thread Local Storage ABI* on page 102).
r14 - r31[2]	Nonvolatile	Local variables.
LR	Volatile	Link register.
CTR	Volatile	Loop count register.
TAR	Reserved	Reserved for system use. This register should not be read or written by application software.
XER	Volatile	Fixed-point exception register.
CR0 - CR1	Volatile	Condition register fields.
CR2 - CR4	Nonvolatile	Condition register fields.
CR5 - CR7	Volatile	Condition register fields.
DSCR	Limited Access	Data stream prefetch control.
VRSAVE	Reserved	Reserved for system use. This register should not be read or written by application software.

1. Register r2 is nonvolatile with respect to calls between functions in the same compilation unit. It is saved and restored by code inserted by the linker resolving a call to an external function. (For more information, see *TOC Pointer Usage* on page 33.
2. If a function needs a frame pointer, assigning r31 to the role of the frame pointer is recommended.

TOC Pointer Usage

As described in *Section 3.4 Symbol Table* on page 87, the TOC pointer, r2, is commonly initialized by the global function entry point when a function is called through the global entry point. It may be called from a module other than the current function's module or from an unknown call point, such as through a function pointer. (For more information, see *Section 2.3.2.1 Function Prologue* on page 60.)

In those instances, it is the caller's responsibility to store the TOC pointer, r2, in the TOC pointer doubleword of the caller's stack frame. For references external to the compilation unit, this code is inserted by the static linker if a function is to be resolved by the dynamic linker. For references through function pointers, it is the

compiler's or assembler programmer's responsibility to insert appropriate TOC save and restore code. If the function is called from the same module as the callee, the callee must preserve the value of r2. (See *Section 3.6.1 Function Call* on page 100 for a description of function entry conventions.)

When a function calls another function, the TOC pointer must have a legal value pointing to the TOC base, which may be initialized as described in *Section 4.2.3 Global Offset Table* on page 125.

When global data is accessed, the TOC pointer must be available for dereference at the point of all uses of values derived from the TOC pointer in conjunction with the @l operator. This property is used by the linker to optimize TOC pointer accesses. In addition, all reaching definitions for a TOC-pointer-derived access must compute the same definition for code to be ABI compliant. (See the *Section 3.6.3.1 TOC Pointer Usage* on page 100.)

In some implementations, non ABI-compliant code may be processed by providing additional linker options; for example, linker options disabling linker optimization. However, this behavior in support of non-ABI compliant code is not guaranteed to be portable and supported in all systems. For examples of compliant and noncompliant code, see *Section 3.6.3.1 TOC Pointer Usage* on page 100.

Optional Function Linkage

Except as follows, a function cannot depend on the values of those registers that are optional in the function linkage (r0, r11, and r12) because they may be altered by inter-library calls:

- When a function is entered in a way to initialize its environment pointer, register r11 contains the environment pointer. It is used to support languages with access to additional environment context; for example, for languages that support lexical nesting to access its lexically nested outer context.
- When a function is entered through its global entry point, register r12 contains the entry-point address. For more information, see the description of dual entry points in *Section 2.3.2.1 Function Prologue* on page 60 and *Section 2.3.2.2 Function Epilogue* on page 62.

Stack Frame Pointer

The stack pointer always points to the lowest allocated valid stack frame. It must maintain quadword alignment and grow toward the lower addresses. The contents of the word at that address point to the previously allocated stack frame when the code has been compiled to maintain back chains. A called function is permitted to decrement it if required. For more information, see *Section 2.3.8 Dynamic Stack Space Allocation* on page 79.

Link Register

The link register contains the address that a called function normally returns to. It is volatile across function calls.

Condition Register Fields

In the condition register, the bit fields CR2, CR3, and CR4 are nonvolatile. The value on entry must be restored on exit. The other bit fields are volatile.

This ABI requires OpenPOWER-compliant processors to implement **mfocr** instructions in a manner that initializes undefined bits of the RT result register of **mfocr** instructions to one of the following values:

- 0, in accordance with OpenPOWER-compliant processor implementation practice
- The architected value of the corresponding CR field in the **mfocr** instruction

Erratum: When executing an **mfocr** instruction, the POWER8 processor does not implement the behavior described in the “Fixed Point Invalid Forms and Undefined Conditions” section of *POWER8 Processor User’s Manual for the Single-Chip Module*. Instead, it replicates the selected condition register field within the byte that contains it rather than initializing the bits corresponding to the nonselected bits of the byte that contains it to 0. When generating code to save two condition register fields that are stored in the same byte, the compiler must mask the value received from **mfocr** to avoid corruption of the resulting (partial) condition register word.

For more information, see *Power ISA*, version 2.07 and “Fixed Point Invalid Forms and Undefined Conditions” in *POWER8 Processor User’s Manual for the Single-Chip Module*.

In OpenPOWER-compliant processors, floating-point and vector functions are implemented using a unified vector-scalar model. As shown in *Figure 2-11* and *Figure 2-12*, there are 64 vector-scalar registers; each is 128 bits wide.

The vector-scalar registers can be addressed with vector-scalar instructions, for vector and scalar processing of all 64 registers, or with the “classic” Power floating-point instructions to refer to a 32-register subset of 64 bits per register. They can also be addressed with VMX instructions to refer to a 32-register subset of 128-bit wide registers.

Figure 2-11. Floating-Point Registers as Part of VSRs

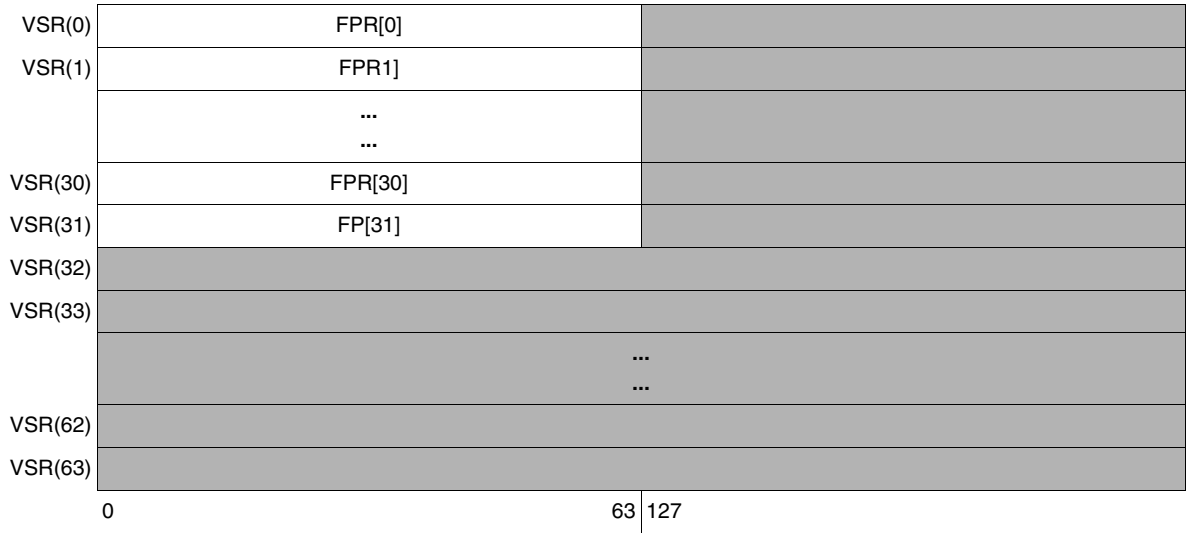
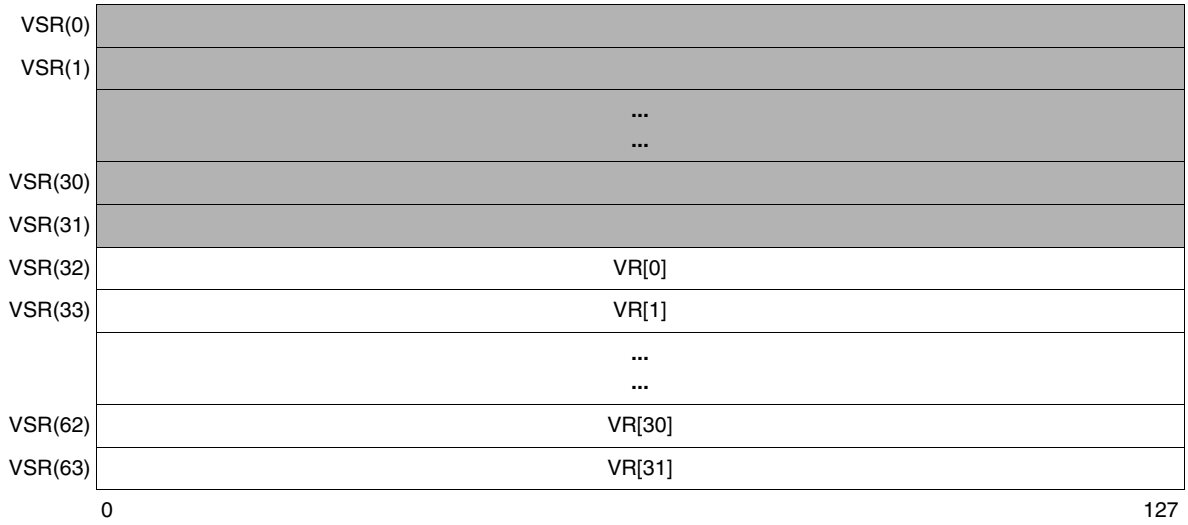


Figure 2-12. Vector Registers as Part of VSRs



The classic floating-point repertoire consists of 32 floating-point registers, each 64 bits wide, and an associated special-purpose register to provide floating-point status and control. Throughout this document, the symbol fN is used, where N is a register number, to refer to floating-point register N.

For the purpose of function calls, the right half of VSX registers, corresponding to the classic floating-point registers (that is, vsr0 - vsr31), is volatile.

Table 2-14. Floating-Point Register Roles for Binary Floating-Point Types

Register	Preservation Rules	Purpose
f0	Volatile	Volatile scratch register.
f1 - f13	Volatile	Used for parameter passing and return values of binary float types.
f14 - f31	Nonvolatile	Local variables.
FPSCR	Limited-access	Floating-point status and control register limited-access bits. Preservation rules governing the limited-access bits for the bit fields [VE], [OE], [UE], [ZE], [XE], and [RN] are presented in *Section 2.2.1.2 Limited-Access Bits* on page 38.

DFP Support

The OpenPOWER ABI supports the decimal floating-point (DFP) format and DFP language extensions. As the DFP repertoire is not part of the OpenPOWER specification, the default implementation of DFP types shall be a software implementation of the IEEE DFP standard (IEEE Standard 754-2008).

The Power ISA decimal floating-point category extends the Power Architecture by adding a decimal floating-point unit. It uses the existing 64-bit floating-point registers and extends the FPSCR register to 64 bits, where it defines a decimal rounding-control field in the extended space. DFP support is defined as an optional category in the OpenPOWER specification. When DFP is supported as a vendor-specific implementation capability, compilers can be used to implement DFP support. The compilers should provide an option to generate DFP instructions or to issue calls to DFP emulation software. The DFP parameters are passed in floating-point registers.

As with other implementation-specific features, the OpenPOWER compliance specification requires that all OpenPOWER compliant programs be able to execute, functionally indistinguishably, on hardware with and without vendor-specific extensions. It is the application's responsibility to transparently adapt to the absence of vendor-specific features by using a library responsive to the presence of DFP hardware, or in conjunction with operating-system dynamic library services, to select from among multiple DFP libraries that contain either a first software implementation or a second hardware implementation.

Single-precision, double-precision, and quad-precision decimal floating-point parameters shall be passed in the floating-point registers. Single-precision decimal floating-point shall occupy the lower half of a floating-point register. Quad-precision floating-point values shall occupy an even/odd register pair. When passing quad-precision decimal floating-point parameters in accordance with this ABI, an odd floating-point register may be skipped in allocation order to align quad-precision parameters and results in an even/odd register pair. When a floating-point register is skipped during input parameter allocation, words in the corresponding GPR or memory doubleword in the parameter list are not skipped.

Table 2-15. Floating-Point Register Roles for Decimal Floating-Point Types

Register	Preservation Rules	Purpose
FPSCR	Limited-access	Floating-point status and control register limited-access bits. Preservation rules governing the limited-access bits for the bit field [DRN] are presented in *Section 2.2.1.2 Limited-Access Bits* on page 38.

The OpenPOWER vector-category instruction repertoire provides the ability to reference 32 vector registers, each 128 bits wide, of the vector-scalar register file, and a special-purpose register VSCR. Throughout this document the symbol vN is used, where N is a register number, to refer to vector register N.

Table 2-16. Vector Register Roles

Register	Preservation Rules	Purpose
v0 - v1	Volatile	Local variables.
v2 - v13	Volatile	Used for parameter passing and return values.
v14 - v19	Volatile	Local variables.
v20 - v31	Nonvolatile	Local variables.
VSCR	Limited-access	32-bit vector status and control register. Preservation rules governing the limited-access bits for the bit field [NJ] are presented in *Section 2.2.1.2 Limited-Access Bits* on page 38.

IEEE BINARY 128 EXTENDED PRECISION

Parameters in IEEE BINARY 128 EXTENDED PRECISION format shall be passed in a single 128-bit vector register as if they were vector values.

IBM EXTENDED PRECISION

Parameters in the IBM EXTENDED PRECISION format with a pair of two double-precision floating-point values shall be passed in two successive floating-point registers.

If only one value can be passed in a floating-point register, the second parameter will be passed in a GPR or memory in accordance with the parameter passing rules for structure aggregates.

2.2.1.2 Limited-Access Bits

The Power ISA identifies a number of registers that have mutability limited to the specific bit fields indicated in the following list:

FPSCR [VE]	The Floating-Point Invalid Operation Exception Enable bit [VE] of the FPSCR register.
FPSCR [OE]	The Floating-Point Overflow Exception Enable bit [OE] of the FPSCR register.
FPSCR [UE]	The Floating-Point Underflow Exception Enable bit [UE] of the FPSCR register.
FPSCR [ZE]	The Floating-Point Zero Divide Exception Enable bit [ZE] of the FPSCR register.
FPSCR [XE]	The Floating-Point Inexact Exception Enable bit [XE] of the FPSCR register.
FPSCR [RN]	The Binary Floating-Point Rounding Control field [RN] of the FPSCR register.
FPSCR [DRN]	The DFP Rounding Control field [DRN] of the 64-bit FPSCR register.
VSCR [NJ]	The Vector Non-Java Mode field [NJ] of the VSCR register.

The bits composing these bit fields are identified as limited access because this ABI manages how they are to be modified and preserved across function calls. Limited-access bits may be changed across function calls only if the called function has specific permission to do so as indicated by the following conditions. A function without permission to change the limited-access bits across a function call shall save the value of the register before modifying the bits and restore it before returning to its calling function.

Limited-Access Conditions

Standard library functions expressly defined to change the state of limited-access bits are not constrained by nonvolatile preservation rules; for example, the fesetround() and feenableexcept() functions. All other standard library functions shall save the old value of these bits on entry, change the bits for their purpose, and restore the bits before returning.

Where a standard library function, such as qsort(), calls functions provided by an application, the following rules shall be observed:

- The limited-access bits, on entry to the first call to such a callback, must have the values they had on entry to the library function.
- The limited-access bits, on entry to a subsequent call to such a callback, must have the values they had on exit from the previous call to such a callback.
- The limited-access bits, on exit from the library function, must have the values they had on exit from the last call to such a callback.

The compiler can directly generate code that saves and restores the limited-access bits.

The values of the limited-access bits are unspecified on entry into a signal handler because a library or user function can temporarily modify the limited-access bits when the signal is taken.

When setjmp() returns from its first call (also known as direct invocation), it does not change the limited access bits. The limited access bits have the values they had on entry to the setjmp() function.

When longjmp() is performed, it appears to be returning from a call to setjmp(). In this instance, the limited access bits are not restored to the values they had on entry to the setjmp() function.

C library functions, such as _FPU_SETCW() defined in <fpu_control.h>, may modify the limited-access bits of the FPSCR. Additional C99 functions that can modify the FPSCR are defined in <fenv.h>.

The vector vec_mtvscr() function may change the limited-access NJ bit.

The unwinder does not modify limited-access bits. To avoid the overhead of saving and restoring the FPSCR on every call, it is only necessary to save it briefly before the call and to restore it after any instructions or groups of instructions that need to change its control flags have been completed. In some cases, that can be avoided by using instructions that override the FPSCR rounding mode.

If an exception and the resulting signal occur while the FPSCR is temporarily modified, the signal handler cannot rely on the default control flag settings and must behave as follows:

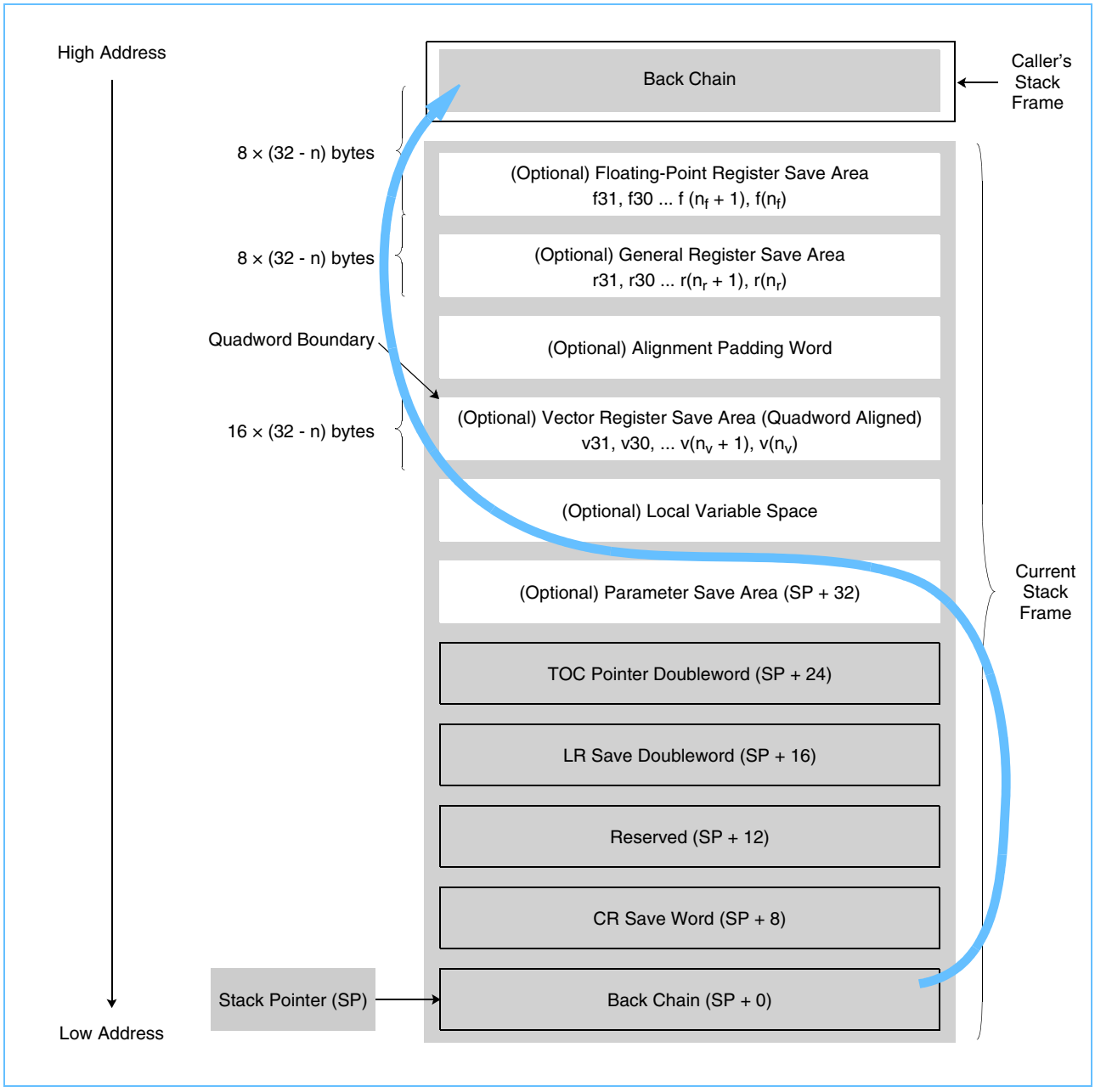
- If the signal handler will unwind the stack, print a traceback, and abort the program, no other special handling is needed.
- If the signal handler will adjust some register values (for example, replace a NaN with a zero or infinity) and then resume execution, no other special handling is needed. There is one exception; if the signal handler changed the control flags, it should restore them.
- If the signal handler will unwind the stack part way and resume execution in a user exception handler, the application should save the FPSCR beforehand and the exception handler should restore its control flags.

2.2.2 The Stack Frame

A function shall establish a stack frame if it requires the use of nonvolatile registers, its local variable usage cannot be optimized into registers and the protected zone, or it calls another function. For more information about the protected zone, see *Section 2.2.2.4* on page 46. It need only allocate space for the required minimal stack frame, consisting of a back-chain doubleword (optionally containing a back-chain pointer), the saved CR word, a reserved word, the saved LR doubleword, and the saved TOC pointer doubleword.

Figure 2-13 shows the relative layout of an allocated stack frame following a nonleaf function call, where the stack pointer points to the back-chain word of the caller's stack frame. By default, the stack pointer always points to the back-chain word of the most recently allocated stack frame. For more information, see *Section 2.2.2.2 Minimum Stack Frame Elements* on page 42.

Figure 2-13. Stack Frame Organization



In *Figure 2-13* the white areas indicate an optional save area of the stack frame. For a description of the optional save areas described by this ABI, see *Section 2.2.2.3 Optional Save Areas* on page 42.

2.2.2.1 General Stack Frame Requirements

The following general requirements apply to all stack frames:

- The stack shall be quadword aligned.

- The minimum stack frame size shall be 32 bytes. A minimum stack frame consists of the first 4 doublewords (back-chain doubleword, CR save word and reserved word, LR save doubleword, and TOC pointer doubleword), with padding to meet the 16-byte alignment requirement.
- There is no maximum stack frame size defined.
- Padding shall be added to the local variable space of the stack frame to maintain the defined stack frame alignment.
- The stack pointer, r1, shall always point to the lowest address doubleword of the most recently allocated stack frame.
- The stack shall start at high addresses and grow downward toward lower addresses.
- The lowest address doubleword (the back-chain word in *Figure 2-13*) shall point to the previously allocated stack frame when a back chain is present. As an exception, the first stack frame shall have a value of 0 (NULL).
- If required, the stack pointer shall be decremented in the called function's prologue and restored in the called function's epilogue.
- The stack pointer shall be updated atomically so that, at all times, it points to a valid back-chain doubleword if a back chain is maintained. This update may be achieved in a number of ways, as described in *Section 2.3.2.3 Rules for Prologue and Epilogue Sequences* on page 62.
- Before a function calls any other functions, it shall save the value of the LR register into the LR save doubleword of the caller's stack frame.

Note: An optional frame pointer may be created if necessary (for example, as a result of dynamic allocation on the stack as described in *Section 2.3.8 Dynamic Stack Space Allocation* on page 79) to address arguments or local variables.

An example of a minimum stack frame allocation that meets these requirements is shown in *Figure 2-14*.

Figure 2-14. Minimum Stack Frame Allocation with and without Back Chain

With back chain:

```
stdu r1,-32(r1)                    - Store back chain, decrement SP
mflr r0                            - Copy LR to R0
std r0,r16(r1)                     - Store the LR in the previous LR save doubleword
```

Without back chain:

```
addi r1,r1,-32                      - Decrement SP
mflr r0                             - Copy LR to R0
std r0,r16(r1)                      - Store the LR in the previous LR save doubleword
```

2.2.2.2 Minimum Stack Frame Elements

Back Chain Doubleword

When a back chain is not present, alternate information compatible with the ABI unwind framework to unwind a stack must be provided by the compiler, for all languages, regardless of language features. A compiler that does not provide such system-compatible unwind information must generate a back chain. All compilers shall generate back chain information by default, and default libraries shall contain a back chain.

On systems where system-wide unwind capabilities are not provided, compilers must not generate object files without back-chain generation. A system shall provided a programmatic interface to query unwind information when system-wide unwind capabilities are provided.

CR Save Word

If a function changes the value in any nonvolatile field of the condition register, it shall first save at least the value of those nonvolatile fields of the condition register in the saved CR word.

Reserved Word

This doubleword is reserved for system functions. Modifications of the value contained in this word are prohibited unless explicitly allowed by future ABI amendments.

LR Save Doubleword

This doubleword stores the return address for the present function when another function is called unless a tail call is performed.

TOC Pointer Doubleword

If a function changes the value of the TOC pointer register, it shall first save it in the TOC pointer doubleword.

2.2.2.3 Optional Save Areas

This ABI provides a stack frame with a number of optional save areas. These areas are always present, but may be of size 0. This section indicates the relative position of these save areas in relation to each other and the primary elements of the stack frame.

Because the back-chain word of a stack frame must maintain quadword alignment, a reserved word is introduced above the CR save word to provide a quadword-aligned minimal stack frame and align the doublewords within the fixed stack frame portion at doubleword boundaries.

An optional alignment padding to quadword-boundary element might be necessary above the vector register save area to provide 16-byte alignment, as shown in *Figure 2-13* on page 40.

Floating-Point Register Save Area

If a function changes the value in any nonvolatile floating-point register fN, it shall first save the value in fN in the Floating-Point Register Save Area and restore the register upon function exit.

If full unwind information such as DWARF is present, registers can be saved in arbitrary locations in the stack frame. If the system floating-point register save and restore functions are to be used, the floating-point registers shall be saved in a contiguous range. Floating-point register fN is saved in the doubleword located 8 × (32 – N) bytes before the back-chain word of the previous frame, as shown in *Figure 2-13* on page 40.

The Floating-Point Register Save Area is always doubleword aligned. The size of the Floating-Point Register Save Area depends upon the number of floating-point registers that must be saved. If no floating-point registers are to be saved, the Floating-Point Register Save Area has a zero size.

General-Purpose Register Save Area

If a function changes the value in any nonvolatile general-purpose register rN, it shall first save the value in rN in the General-Purpose Register Save Area and restore the register upon function exit.

If full DWARF unwind information is present, registers can be saved in arbitrary locations in the stack frame. If the system general-purpose register save and restore functions are to be used, the general-purpose registers shall be saved in a contiguous range. General-purpose register rN is saved in the doubleword located 8 × (32 – N) bytes below the floating-point save area, as shown in *Figure 2-13*.

The General-Purpose Register Save Area is always doubleword aligned. The size of the General-Purpose Register Save Area depends upon the number of general registers that must be saved. If no general-purpose registers are to be saved, the General-Purpose Register Save Area has a zero size.

Vector Register Save Area

If a function changes the value in any nonvolatile vector register vN, it shall first save the value in vN in the Vector Register Save Area and restore the register upon function exit.

If full DWARF unwind information is present, registers can be saved in arbitrary locations in the stack frame. If the system vector register save and restore functions are to be used, the vector registers shall be saved in a contiguous range. Vector register vN is saved in the quadword located 16 × (32 - N) bytes before the General-Purpose Register Save Area plus alignment padding, as shown in *Figure 2-13*.

The vector register save area is always quadword aligned. If necessary to ensure suitable alignment of the vector save area, a padding doubleword may be introduced between the vector register and General-Purpose Register Save Areas, and/or the local variable space may be expanded to the next quadword boundary. The size of the vector register save area depends upon the number of vector registers that must be saved. It ranges from 0 bytes to a maximum of 192 bytes (12 × 16). If no vector registers are to be saved, the Vector Register Save Area has a zero size.

Local Variable Space

The Local Variable Space is used for allocation of local variables. The local variable space is located immediately above the parameter save area, at a higher address. There is no restriction on the size of this area.

Note: Sometimes a register spill area is needed. It is typically positioned above the local variable space.

The local variable space also contains any parameters that need to be assigned a memory address when the function's parameter list does not require a save area to be allocated by the caller.

Parameter Save Area

The parameter save area shall be allocated by the caller for function calls unless a prototype is provided for the callee indicating that all parameters can be passed in registers. (This requires a parameter save area to be created for functions where the number and type of parameters exceeds the registers available for parameter passing in registers, for those functions where the prototype contains an ellipsis to indicate a variadic function, and functions are declared without prototype.)

When the caller allocates the parameter save area, it will always be automatically quadword aligned because it must always start at SP + 32. It shall be at least 8 doublewords in length. If a function needs to pass more than 8 doublewords of arguments, the parameter save area shall be large enough to spill all register-based parameters and to contain the arguments that the caller stores in it.

The calling function cannot expect that the contents of this save area are valid when returning from the callee.

The parameter save area, which is located at a fixed offset of 32 bytes from the stack pointer, is reserved in each stack frame for use as an argument list when an in-memory argument list is required. For example, a parameter save area must be allocated by the caller when calling functions with the following characteristics:

- Prototyped functions where the parameters cannot be contained in the parameter registers
- Prototyped functions with variadic arguments
- Functions without a suitable declaration available to the caller to determine the called function's characteristics (for example, functions that use the ISO C and C++ languages)

Under these circumstances, a minimum of 8 doublewords are always reserved. The size of this area must be sufficient to hold the longest argument list being passed by the function that owns the stack frame. Although not all arguments for a particular call are located in storage, when an in-memory parameter list is required, consider the parameters to be forming a list in this area. Each argument occupies one or more doublewords.

More arguments might be passed than can be stored in the parameter registers. In that case, the remaining arguments are stored in the parameter save area. The values passed on the stack are identical to the values placed in registers. Therefore, the stack contains register images for the values that are not placed into registers.

This ABI uses a simple va_list type for variable lists to point to the memory location of the next parameter. Therefore, regardless of type, variable arguments must always be in the same location so that they can be found at runtime. The first 8 doublewords are located in general registers r3 - r10. Any additional doublewords are located in the stack parameter save area. Alignment requirements such as those for vector types may require the va_list pointer to first be aligned before accessing a value.

Follow these rules for parameter passing:

- Map each argument to enough doublewords in the parameter save area to hold its value.
- Map single-precision floating-point values to the least-significant word in a single doubleword.
- Map double-precision floating-point values to a single doubleword.
- Map simple integer types (char, short, int, long, enum) to a single doubleword. Sign or zero extend values shorter than a doubleword to a doubleword based on whether the source data type is signed or unsigned.
- When 128-bit integer types are passed by value, map each to two consecutive GPRs, two consecutive doublewords, or a GPR and a doubleword.[1] The required alignment of int128 data types is 16 bytes. Therefore, by-value parameters must be copied to a new location in the local variable area of the callee's stack frame before the address of the type can be provided (for example, using the address-of operator,

or when the variable is to be passed by reference), when the incoming parameter is not aligned at a 16-byte boundary.

- If extended precision floating-point values in IEEE BINARY 128 EXTENDED PRECISION format are supported (see *IEEE BINARY 128 EXTENDED PRECISION* on page 23), map them to a single quadword, quadword aligned. This might result in skipped doublewords in the parameter save area.
- If extended precision floating-point values in IBM EXTENDED PRECISION format are supported (see *IBM EXTENDED PRECISION* on page 23), map them to two consecutive doublewords. The required alignment of IBM EXTENDED PRECISION data types is 16 bytes. Therefore, by-value parameters must be copied to a new location in the local variable area of the callee's stack frame before the address of the type can be provided (for example, using the address-of operator, or when the variable is to be passed by reference), when the incoming parameter is not aligned at a 16-byte boundary.
- Map complex floating-point and complex integer types as if the argument was specified as separate real and imaginary parts.
- Map pointers to a single doubleword.
- Map vectors to a single quadword, quadword aligned. This might result in skipped doublewords in the parameter save area.
- Map fixed-size aggregates and unions passed by value to as many doublewords of the parameter save area as the value uses in memory. Align aggregates and unions as follows:
  - Aggregates that contain qualified floating-point or vector arguments are normally aligned at the alignment of their base type. For more information about qualified arguments, see *Section 2.2.3 Parameter Passing in Registers* on page 46.
  - Other aggregates are normally aligned in accordance with the aggregate's defined alignment.
  - The alignment will never be larger than the stack frame alignment (16 bytes).

  This might result in doublewords being skipped for alignment. When a doubleword in the parameter save area (or its GPR copy) contains at least a portion of a structure, that doubleword must contain all other portions mapping to the same doubleword. (That is, a doubleword can either be completely valid, or completely invalid, but not partially valid and invalid, except in the last doubleword where invalid padding may be present.)
- Pad an aggregate or union smaller than one doubleword in size so that it is in the least-significant bits of the doubleword. Pad all others, if necessary, at their tail. Variable size aggregates or unions are passed by reference.
- Map other scalar values to the number of doublewords required by their size.
- Future data types that have an architecturally defined quadword-required alignment will be aligned at a quadword boundary.
- If the callee has a known prototype, arguments are converted to the type of the corresponding parameter when loaded to their parameter registers or when being mapped into the parameter save area. For example, if a long is used as an argument to a float double parameter, the value is converted to double-precision and mapped to a doubleword in the parameter save area.

1. In big-endian environments, the most-significant doubleword of the quadword (__int128) parameter is stored in the lower numbered GPR or parameter word. The least-significant doubleword of the quadword (__int128) is stored in the higher numbered GPR or parameter word. In little-endian environments, the least-significant doubleword of the quadword (__int128) parameter is stored in the lower numbered GPR or parameter word. The most-significant doubleword of the quadword (__int128) is stored in the higher numbered GPR or parameter word.

2.2.2.4 Protected Zone

The 288 bytes below the stack pointer are available as volatile program storage that is not preserved across function calls. Interrupt handlers and any other functions that might run without an explicit call must take care to preserve a protected zone, also referred to as the red zone, of 512 bytes that consists of:

- The 288-byte volatile program storage region that is used to hold saved registers and local variables
- An additional 224 bytes below the volatile program storage region that is set aside as a volatile system storage region for system functions

If a function does not call other functions and does not need more stack space than is available in the volatile program storage region (that is, 288 bytes), it does not need to have a stack frame. The 224-byte volatile system storage region is not available to compilers for allocation to saved registers and local variables.

2.2.3 Parameter Passing in Registers

For the OpenPOWER Architecture, it is more efficient to pass arguments to functions in registers rather than through memory. For more information about passing parameters through memory, see *Parameter Save Area* on page 44. For the OpenPOWER ABI, the following parameters can be passed in registers:

- Up to eight arguments can be passed in general-purpose registers r3 - r10.
- Up to thirteen qualified floating-point arguments can be passed in floating-point registers f1 - f13 or up to twelve in vector registers v2 - v13.
- Up to thirteen single-precision or double-precision decimal floating-point arguments can be passed in floating-point registers f1 - f13.
- Up to six quad-precision decimal floating-point arguments can be passed in even-odd floating-point register pairs f2 - f13.
- Up to 12 qualified vector arguments can be passed in v2 - v13.

A qualified floating-point argument corresponds to:

- A scalar floating-point data type
- Each member of a complex floating-point type
- A member of a homogeneous aggregate of multiple like data types passed in up to eight floating-point registers

  A homogeneous aggregate can consist of a variety of nested constructs including structures and array members, which shall be traversed to determine the types and number of members of the base floating-point type. (A complex floating-point data type is treated as if two separate scalar values of the base type were passed.)

  Homogeneous floating-point aggregates can have up to four IBM EXTENDED PRECISION members, four _Decimal128 members, or eight members of other floating-point types. They are passed in floating-point registers if parameters of that type would be passed in floating-point registers. They are passed in vector registers if parameters of that type would be passed in vector registers. They are passed as if each member was specified as a separate parameter.

A qualified vector argument corresponds to:

- A vector data type
- A member of a homogeneous aggregate of multiple like data types passed in up to eight vector registers

- Any future type requiring 16-byte alignment (see *Section 2.2.2.3 Optional Save Areas* on page 42) or processed in vector registers

  A homogeneous aggregate can consist of a variety of nested constructs including structures and array members, which shall be traversed to determine the types and number of members of the base vector type. Homogeneous vector aggregates with up to eight members are passed in in up to eight vector registers as if each member was specified as a separate parameter.

Note: Floating-point and vector aggregates that contain padding words and integer fields with a width of 0 should not be treated as homogeneous aggregates.

Binary extended precision numbers in IEEE BINARY 128 EXTENDED PRECISION format (see page 37) are passed using a VMX register. Binary extended precision numbers in IBM EXTENDED PRECISION format (see page 23) are passed using two successive floating-point registers. Single-precision decimal floating-point numbers (see *DFP Support* on page 36) are passed in the lower half of a floating-point register. Quad-precision decimal floating-point numbers (see *DFP Support* on page 36) are passed using a paired even/odd floating-point register pair. A floating-point register might be skipped to allocate an even/odd register pair when necessary. When a floating-point register is skipped, no corresponding memory word is skipped in the natural home location; that is, the corresponding GPR or memory doubleword in the parameter list.

All other aggregates are passed in consecutive GPRs, in GPRs and in memory, or in memory.

When a parameter is passed in a floating-point or vector register, a number of GPRs are skipped, in allocation order, commensurate to the size of the corresponding in-memory representation of the passed argument's type.

Each parameter is allocated to at least one doubleword.

Full doubleword rule:

When a doubleword in the parameter save area (or its GPR copy) contains at least a portion of a structure, that doubleword must contain all other portions mapping to the same doubleword. (That is, a doubleword can either be completely valid, or completely invalid, but not partially valid and invalid, except in the last doubleword where invalid padding may be present.)

IEEE BINARY 128 EXTENDED PRECISION

Up to 12 quad-precision parameters can be passed in v2 - v13.

IBM EXTENDED PRECISION

IBM EXTENDED PRECISION format parameters are passed as if they were a struct consisting of separate double parameters.

IBM EXTENDED PRECISION format parameters shall be considered as a distinct type for the determination of homogeneous aggregates.

If fewer arguments are needed, the unused registers defined previously will contain undefined values on entry to the called function.

If there are more arguments than registers or no function prototype is provided, a function must provide space for all arguments in its stack frame. When this happens, only the minimum storage needed to contain all arguments (including allocating space for parameters passed in registers) needs to be allocated in the stack frame.

General-purpose registers r3 - r10 correspond to the allocation of parameters to the first 8 doublewords of the parameter save area. Specifically, this requires a suitable number of general-purpose registers to be skipped to correspond to parameters passed in floating-point and vector registers.

If a parameter corresponds to an unnamed parameter that corresponds to the ellipsis, a caller shall promote float values to double. If a parameter corresponds to an unnamed parameter that corresponds to the ellipsis, the parameter shall be passed in a GPR or in the parameter save area.

If no function prototype is available, the caller shall promote float values to double and pass floating-point parameters in both available floating-point registers and in the parameter save area. If no function prototype is available, the caller shall pass vector parameters in both available vector registers and in the parameter save area. (If the callee expects a float parameter, the result will be incorrect.)

It is the callee's responsibility to allocate storage for the stored data in the local variable area. When the callee's parameter list indicates that the caller must allocate the parameter save area (because at least one parameter must be passed in memory or an ellipsis is present in the prototype), the callee may use the preallocated parameter save area to save incoming parameters.

2.2.3.1 Parameter Passing Register Selection Algorithm

The following algorithm describes where arguments are passed for the C language. In this algorithm, arguments are assumed to be ordered from left (first argument) to right. The actual order of evaluation for arguments is unspecified.

- gr contains the number of the next available general-purpose register.
- fr contains the number of the next available floating-point register.
- vr contains the number of the next available vector register.

Note: The following types refer to the type of the argument as declared by the function prototype. The argument values are converted (if necessary) to the types of the prototype arguments before passing them to the called function.

If a prototype is not present, or it is a variable argument prototype and the argument is after the ellipsis, the type refers to the type of the data objects being passed to the called function.

- INITIALIZE: If the function return type requires a storage buffer, set gr = 4; else set gr = 3.

```
Set fr = 1
Set vr = 2
```

- SCAN: If there are no more arguments, terminate. Otherwise, allocate as follows based on the class of the function argument:

```
switch(class(argument))

unnamed parameter:
             if gr > 10
                    goto mem_argument
```

```
                size = size_in_DW(argument)
                reg_size = size-(11-gr)
                pass (GPR, gr, first_n_DW (argument, reg_size));

                if remaining_members
                       argument = after_n_DW(argument,reg_size))
                       goto mem_argument

                break;

   integer:  // up to 64b
   pointer:  // this also includes all pass by reference values

                if gr > 10
                       goto mem_argument
                pass (GPR, gr, argument);
                gr++

                break;


aggregate:
                if (homogeneous(argument,float) and regs_needed(members(argument)) <= 8)
                       if (register_type_used (type (argument)) == vr)
                          goto use_vrs;
                         n_fregs = n_fregs_for_type(member_type(argument,0))
                         agg_size = members(argument) * n_fregs
                         reg_size = agg_size-(15-fr)
                         pass(FPR,fr,first_n_DW(argument,reg_size)
                         fr += reg_size;
                         gr += size_in_DW (first_n_DW(argument,reg_size))

                       if remaining_members
                           argument = after_n_DW(argument,reg_size))
                           goto gpr_struct

                break;

                if (homogeneous(argument,vector) and members(argument) <= 8)
                       use_vrs:
                         agg_size = members(argument)
                         reg_size = agg_size-(14-vr)
                         if (gr&1 = 0)  // align vector in memory
                           gr++
                         pass(VR,vr,first_n_elements(argument,reg_size);
                         vr += reg_size
                         gr += size_in_DW (first_n_elements(argument,reg_size)

                       if remaining_members
                           argument = after_n_elements(argument,reg_size))
```

```
                        goto gpr_struct

            break;

            if gr > 10
                    goto mem_argument

            size = size_in_DW(argument)

gpr_struct:
            reg_size = size-(11-gr)
            pass (GPR, gr, first_n_DW (argument, reg_size));

            if remaining_members
                    argument = after_n_DW(argument,reg_size))
                    goto mem_argument

            break;


float:

// float is passed in one FPR.
// double is passed in one FPR.
// IBM EXTENDED PRECISION is passed in the next two FPRs.
// IEEE BINARY 128 EXTENDED PRECISION is passed in one VR.
// _Decimal32 is passed in the lower half of one FPR.
// _Decimal64 is passed in one FPR.
// _Decimal128 is passed in an even-odd FPR pair, skipping an FPR if necessary.

            if (register_type_used (type (argument)) == vr)    // Assumes == vr is true for
              IEEE BINARY 128 EXTENDED PRECISION.
                    goto use_vr;
            fr += align_pad(fr,type(argument))  // Assumes align_pad = 8 for _Decimal128 if
              fr is odd; otherwise = 0.

            if fr > 14
                    goto mem_argument

            n_fregs = n_fregs_for_type(argument)    // Assumes n_fregs_for_type == 2 for IBM
              EXTENDED PRECISION or _Decimal128, == 1 for float, double, _Decimal32 or
              _Decimal64.
            pass(FPR,fr,argument)
            fr += n_fregs
            gr += size_in_DW(argument)

            break;


vector:
          Use vr:
```

```
                if vr > 13
                        goto mem_argument

                if (gr&1 = 0)  // align vector in memory
                        gr++

                pass(VR,vr,argument)
                vr ++
                gr += 2

                break;

    next argument;

    mem_argument:
                need_save_area = TRUE
                pass (stack, gr, argument)
                gr += size_in_DW(argument)

    next argument;
```

All complex data types are handled as if two scalar values of the base type were passed as separate parameters.

If the callee takes the address of any of its parameters, values passed in registers are stored to memory. It is the callee's responsibility to allocate storage for the stored data in the local variable area. When the callee's parameter list indicates that the caller must allocate the parameter save area (because at least one parameter must be passed in memory, or an ellipsis is present in the prototype), the callee may use the preallocated parameter save area to save incoming parameters. (If an ellipsis is present, using the preallocated parameter save area ensures that all arguments are contiguous.) If the compilation unit for the caller contains a function prototype, but the callee has a mismatching definition, this may result in the wrong values being stored.

Note: If the declaration of a function that is used by the caller does not match the definition for the called function, corruption of the caller's stack space can occur.

2.2.3.2 Parameter Passing Examples

This section provides some examples that use the algorithm described in *Section 2.2.3.1 Parameter Passing Register Selection Algorithm* on page 48.

Figure 2-15 shows how parameters are passed for a function that passes arguments in GPRs, FPRs, and memory.

Figure 2-15. Passing Arguments in GPRs, FPRs, and Memory

```
typedef struct {
  int    a;
  double dd;
} sparm;
sparm   s, t;
int     c, d, e;
```

```
long double ld;/* IBM EXTENDED PRECISION format */
double  ff, gg, hh;

x = func(c, ff, d, ld, s, gg, t, e, hh);
Parameter     Register     Offset in parameter save area
c             r3           0-7    (not stored in parameter save area)
ff            f1           8-15   (not stored)
d             r5           16-23  (not stored)
ld            f2,f3        24-39  (not stored)
s             r8,r9        40-55  (not stored)
gg            f4           56-63  (not stored)
t             (none)       64-79  (stored in parameter save area)
e             (none)       80-87  (stored)
hh            f5           88-95  (not stored)
```

Note: If a prototype is not in scope:

- The floating-point argument ff is also passed in r4.
- The long double argument ld is also passed in r6 and r7.
- The floating-point argument gg is also passing in r10.
- The floating-point arguments gg and hh are also stored into the parameter save area.

If a prototype containing an ellipsis describes any of these floating-point arguments as being part of the variable argument part, the general registers and parameter save area are used as when no prototype is in scope. The floating-point registers are not used.

Figure 2-16 shows the definitions that are used in the remaining examples of parameter passing.

Figure 2-16. Parameter Passing Definitions

```
typedef struct {
  double a
  double b;
} dpfp2;

typedef struct
  float a
  float b;
} spfp2;

double a1,a4;
dpfp2 a2,a3  ;
spfp a6,a7;
double func2 (double a, dpfp2 p1, dpfp p2, double b, int x);
double func3 (double a, dpfp2 p1, dpfp p2, double b, int x, spfp2 p3,spfpp4);

struct three_floats { float a,b,c;}
struct two_floats { float a,b;}
```

Figure 2-17 shows how parameters are passed for a function that passes homogenous floating-point aggregates and integer parameters in registers without allocating a parameter save area because all the parameters can be contained in the registers.

Figure 2-17. Passing Homogeneous Floating-Point Aggregates and Integer Parameters in Registers without a Parameter Save Area

```
x = func2(a1,a2,a3,a4, 5);
Parameter    Register      Offset in parameter save area
a1                  f1     n/a
a2.a                f2     n/a
a2.b                f3     n/a
a3.a                f4     n/a
a3.b                f5     n/a
a4                  f6     n/a
5                   r9     n/a
```

Figure 2-18 shows how parameters are passed for a function that passes homogenous floating-point aggregates and integer parameters in registers without allocating a parameter save area because all parameters can be passed in registers.

Figure 2-18. Passing Homogeneous Floating-Point Aggregates and Integer Parameters in Registers without a Parameter Save Area

```
x = func3(a1,a2,a3,a4, 5,a6,a7);
Parameter    Register Offset in parameter save area
a1           f1     n/a
a2.a         f2     n/a
a2.b         f3     n/a
a3.a         f4     n/a
a3.b         f5     n/a
a4           f6     n/a
5            r9     n/a
a6.a         f7     n/a
a6.b         f8     n/a
a7.a         f9     n/a
a7.b         f10    n/a
```

Figure 2-19 shows how parameters are passed for a function that passes floating-point scalars and homogeneous floating-point aggregates in registers and memory because the number of available parameter registers has been exceeded. It demonstrate the full doubleword rule.

Figure 2-19. Passing Floating-Point Scalars and Homogeneous Floating-Point Aggregates in Registers and Memory

```
x = oddity (float d1, float d2, float d3, float d4, float d5,
                    float d6, float d7, float d8, float d9, float d10,
                    float d11, float d12, struct three_floats x)
```

```
Parameter    Register      Offset in parameter save area
d1                  f1     0   (not stored)
d2                  f2     8   (not stored)
d3                  f3     16  (not stored)
d4                  f4     24  (not stored)
d5                  f5     32  (not stored)
d6                  f6     40  (not stored)
d7                  f7     48  (not stored)
d8                  f8     56  (not stored)
d9                  f9     64  (not stored)
d10                 f10    72  (not stored)
d11                 f11    80  (not stored)
d12                 f12    88  (not stored)
x.a                 f13    96  (store because of no partial DW rule)
x.b                 -      100 (stored)
x.c                 -      104 (stored)
```

Figure 2-20 shows how parameters are passed for a function that passes homogeneous floating-point aggregates and floating-point scalars in general-purpose registers because the number of available floating-point registers has been exceeded. In this figure, a parameter area is not allocated because all the parameters can be passed in registers.

Figure 2-20. Passing Floating-Point Scalars and Homogeneous Floating-Point Aggregates in FPRs and GPRs without a Parameter Save Area

```
x = oddity2 (struct two_floats s1, struct two_floats s2, struct two_floats s3, struct
two_floats s4, struct two_floats s5, struct two_floats s6, struct two_floats s7, struct
two_floats s8)

Parameter    Register      Offset in parameter save area
s1.a                f1     n/a
s1.b                f2     n/a
s2.a                f3     n/a
s2.b                f4     n/a
s3.a                f5     n/a
s3.b                f6     n/a
s4.a                f7     n/a
s4.b                f8     n/a
s5.a                f9     n/a
s5.b                f10    n/a
s6.a                f11    n/a
s6.b                f12    n/a
s7.a                f13    n/a
s7.b                -      n/a
s7                  gpr9   n/a
s8                  gpr10  n/a
```

Figure 2-21 shows how parameters are passed for a function that passes homogeneous floating-point aggregates in FPRs, GPRs, and memory because the number of available floating-point and integer parameter registers has been exceeded. In this figure, a parameter area is allocated because all the parameters cannot be passed in the registers. This figure also demonstrates the full doubleword rule applied to GPR7.

Figure 2-21. Passing Homogeneous Floating-Point Aggregates in FPRs, GPRs, and Memory with a Parameter Save Area

```
x = oddity3 (struct two_floats s1, struct two_floats s2, struct two_floats s3, struct
two_floats s4, struct two_floats s5, struct two_floats s6, struct two_floats s7, struct
two_floats s8, struct two_floats s9)

Parameter    Register      Offset in parameter save area
s1.a                f1     0   (not stored)
s1.b                f2     4   (not stored)
s2.a                f3     8   (not stored)
s2.b                f4     12 (not stored)
s3.a                f5     16 (not stored)
s3.b                f6     20 (not stored)
s4.a                f7     24 (not stored)
s4.b                f8     28 (not stored)
s5.a                f9     32 (not stored)
s5.b                f10    36 (not stored)
s6.a                f11    40 (not stored)
s6.b                f12    44 (not stored)
s7.a                f13    48 (not stored, SPFP in FPR)
s7.b                -      52 (not stored)
s7                  gpr9   48 (not stored, full gpr)
s8                  gpr10  56 (not stored, full gpr)
s9                         64 (stored)
```

Figure 2-22 shows how parameters are passed for a function that passes vector data types in VRs, GPRs, and FPRs. In this figure, a parameter area is not allocated.

Figure 2-22. Passing Vector Data Types without Parameter Save Area

```
x =func4(int s1, vector float s2, float s3, vector int s4, vector char s5)
Parameter    Register   Offset in parameter save area
s1           gpr3       n/a
s2           v2         n/a
s3           f1         n/a
s4           v3         n/a
s5           v4         n/a
```

Figure 2-23 on page 56 shows how parameters are passed for a function that passes vector data types in VRs, GPRs, and FPRs. In this figure, a parameter area is allocated.

Figure 2-23. Passing Vector Data Types with a Parameter Save Area

```
x =func5(int s1, vector float s2, float s3, vector int s4, int s5, char s6)
Parameter    Register   Offset in parameter save area
s1           gpr3       0 (not stored)
s2           v2         16 (not stored)
s3           f1         32 (not stored)
s4           v3         48 (not stored)
s5           -          64 (stored)
s6           -          72 (stored)
```

When a function takes the address of at least one of its arguments, it is the callee's responsibility to store function parameters in memory and provide a suitable memory address for parameters passed in registers.

For functions where all parameters can be contained in the parameter registers and without an ellipsis, the caller shall allocate saved parameters in the local variable save area because the caller may not have allocated a register save area. This can be performed, for example, in the prologue. For functions where the caller must allocate a register save area because at least one parameter must be passed in memory, or has an ellipsis in the prototype to indicate the presence of a variadic function, references to named parameters may be spilled to the register save area.

2.2.4 Variable Argument Lists

C programs that are intended to be portable across different compilers and architectures must use the header file <stdarg.h> to deal with variable argument lists. This header file contains a set of macro definitions that define how to step through an argument list. The implementation of this header file may vary across different architectures, but the interface is the same.

C programs that do not use this header file for the variable argument list and assume that all the arguments are passed on the stack in increasing order on the stack are not portable, especially on architectures that pass some of the arguments in registers. The Power Architecture is one of the architectures that passes some of the arguments in registers.

The parameter list may be zero length and is only allocated when parameters are spilled, when a function has unnamed parameters, or when no prototype is provided. When the parameter save area is allocated, the parameter save area must be large enough to accommodate all parameters, including parameters passed in registers.

2.2.5 Return Values

Functions that return a value shall place the result in the same registers as if the return value was the first named input argument to a function unless the return value is a nonhomogeneous aggregate larger than 2 doublewords or a homogeneous aggregate with more than eight registers.[1] (Homogeneous aggregates are arrays or structs of a homogeneous type and a known fixed size.) Therefore, IBM EXTENDED PRECISION functions are returned in f1:f2.

1. For a definition of homogeneous aggregates, see *Section 2.2.3 Parameter Passing in Registers* on page 46.

Homogeneous floating-point or vector aggregate return values that consist of up to eight registers with up to eight elements will be returned in floating-point or vector registers that correspond to the parameter registers that would be used if the return value type were the first input parameter to a function.

Aggregates that are not returned by value are returned in a storage buffer provided by the caller. The address is provided as a hidden first input argument in general-purpose register r3.

Note: Quadword decimal floating-point return values shall be returned in the first paired floating-point register parameter pair; that is, f2:f3.

Functions that return values of the following types shall place the result in register r3 as signed or unsigned integers, as appropriate, and sign extended or zero extended to 64 bits where necessary:

- char
- enum
- short
- int
- long
- pointer to any type
- _Bool

2.3 Coding Examples

The following ISO C coding examples are provided as illustrations of how operations may be done, not how they shall be done, for calling functions, accessing static data, and transferring control from one part of a program to another. They are shown as code fragments with simplifications to explain addressing modes. They do not necessarily show the optimal code sequences or compiler output. The small data area is not used in any of them. For more information, see *Section 3.4.2 Use of the Small Data Area* on page 89.

The previous sections explicitly specify what a program, operating system, and processor may and may not assume and are the definitive reference to be used.

In these examples, absolute code and position-independent code are referenced.

When instructions hold absolute addresses, a program must be loaded at a specific virtual address to permit the absolute code model to work.

When instructions hold relative addresses, a program library can be loaded at various positions in virtual memory and is referred to as a position-independent code model.

2.3.1 Code Model Overview

Executable modules can be built to use either position-dependent or position-independent memory references. Position-dependent references generally result in better performing programs.

Static modules representing the base executables and libraries intended to be statically linked into a base executable can be compiled and linked using either position-dependent or position-independent code.

Dynamically shared objects (DSOs) intended to be used as shared libraries and position-independent executables must be compiled and linked as position-independent code.

2.3.1.1 Position-Dependent Code

Static objects are preferably built by using position-dependent code. Position-dependent code can reference data in one of the following ways:

- Directly by creating absolute memory addresses using a combination of instructions such as lis, addi, and memory instructions:

```
lis     r16, symbol@ha
ld      r12, symbol@l(r16)

lis     r16, symbol2@ha
addi    r16, r16, symbol2@l
lvx     v1, r0, r16
```

- By instantiating the TOC pointer in r2 and using TOC-pointer relative addressing. (For more information, see *Section 3.3 TOC* on page 86.)

```
<load TOC base to r2>

ld      r12, symbol@toc(r2)


li      r16, symbol2@toc
lvx     v1, r2, r16
```

- By instantiating the TOC pointer in r2 and using GOT-indirect addressing:

```
<load TOC base to r2>

ld      r12, symbol@got(r2)
ld      r12, 0(r12)

ld      r12, symbol2@got(r2)
lvx     v1, 0, r12
```

In the OpenPOWER ELF V2 ABI, position-dependent code built with this addressing scheme may have a Global Offset Table (GOT) in the data segment that holds addresses. (For more information, see *Section 4.2.3 Global Offset Table* on page 125.) For position-dependent code, GOT entries are typically updated to reflect the absolute virtual addresses of the reference objects at static link time. Any remaining GOT entries are updated by the loader to reflect the absolute virtual addresses that were assigned for the process. These data segments are private, while the text segments are shared. In systems based on the Power Architecture, the GOT can be addressed with a single instruction if the GOT size is less than 65,536 bytes. A larger GOT requires more general code to access all of its entries.

OpenPOWER-compliant processor hardware implementation and linker optimizations described here work together to optimize efficient code generation for applications with large GOTs. They use instruction fusion to combine multiple ISA instructions into a single internal operation.

Offsets from the TOC register can be generated using either:

- 16-bit offsets (small code model), with a maximum addressing reach of 64 KB for TOC-based relative addressing or GOT accesses
- 32-bit offsets (medium or large code model) with a maximum addressing reach of 4 GB

Efficient implementation of the OpenPOWER ELF V2 ABI medium code model is supported by additional optimizations present in OpenPOWER-compliant processor implementations and the OpenPOWER ABI tool-chain (see *Section 2.3.1.3 Code Models* on page 60).

Position-dependent code is most efficient if the application is loaded in the first 2 GB of the address space because direct address references and TOC-pointer initializations can be performed using a two-instruction sequence.

2.3.1.2 Position-Independent Code

A shared object file is mapped with virtual addresses to avoid conflicts with other segments in the process. Because of this mapping, shared objects use position-independent code, which means that the instructions do not contain any absolute addresses. Avoiding the use of absolute addresses allows shared objects to be loaded into different virtual address spaces without code modification, which can allow multiple processes to share the same text segment for a shared object file.

Two techniques are used to deal with position-independent code:

- First, branch instructions use an offset to the current effective address (EA) or use registers to hold addresses. The Power Architecture provides both EA-relative branch instructions and branch instructions that use registers. In both cases, absolute addressing is not required.
- Second, when absolute addressing is required, the value can be computed with a Global Offset Table (GOT), which holds the information for address computation. Static and const references can be accessed using a TOC pointer relative addressing model, while (shared) extern references must be accessed using the GOT-indirect addressing scheme. Both addressing schemes require a TOC pointer to be initialized.

DSOs can access data as follows:

- By instantiating the TOC pointer in r2 and using TOC pointer relative addressing (for private data).

```
<load TOC base to r2>

ld      r12, symbol@toc(r2)

li      r16, symbol2@toc
lvx     v1, r2, r16
```

- By instantiating the TOC pointer in r2 and using GOT-indirect addressing (for shared data or for very large data sections):

```
<load TOC base to r2>

ld      r12, symbol@got(r2)

ld      r12, 0(r12)

ld      r12 symbol2@got(r2)
lvx     v1, 0, r12
```

Position-independent executables or shared objects have a GOT in the data segment that holds addresses. When the system creates a memory image from the file, the GOT entries are updated to reflect the absolute virtual addresses that were assigned for the process. These data segments are private, while the text segments are shared. In systems based on the Power Architecture, the GOT can be addressed with a single instruction if the GOT size is less than 65,536 bytes. A larger GOT requires more general code to access all of its entries.

The OpenPOWER-compliant processor hardware implementation and linker optimizations described here work together to optimize efficient code generation for applications with large GOTs. They use instruction fusion to combine multiple ISA instructions into a single internal operation.

2.3.1.3 Code Models

Compilers may provide different code models depending on the expected size of the TOC and the size of the entire executable or shared library.

- Small code model: The TOC is accessed using 16-bit offsets from the TOC pointer. This limits the size of a single TOC to 64 KB. Position-independent code uses GOT-indirect addressing to access other objects in the binary.
- Large code model: The TOC is accessed using 32-bit offsets from the TOC pointer, except for .sdata and .sbss, which are accessed using 16-bit offsets from the TOC pointer. This allows a TOC of at least 2 GB. Position-independent code uses GOT-indirect addressing to access other objects in the binary.
- Medium code model: Like the large code model, the TOC is accessed using 32-bit offsets from the TOC pointer, except for .sdata and .sbss, which are accessed using 16-bit offsets. In addition, accesses to module-local code and data objects use TOC pointer relative addressing with 32-bit offsets. Using TOC pointer relative addressing removes a level of indirection, resulting in faster access and a smaller GOT. However. it limits the size of the entire binary to between 2 GB and 4 GB, depending on the placement of the TOC base.

  Note: The medium code model is the default for compilers, and it is applicable to most programs and libraries. The code examples in this document generally use the medium code model.

When linking medium and large code model relocatable objects, the linker should place the .sdata and .sbss sections near to the TOC base.

A linker must allow linking of relocatable object files using different code models. This may be accomplished by sorting the constituent sections of the TOC so that sections that are accessed using 16-bit offsets are placed near to the TOC base, by using multiple TOCs, or by some other method. The suggested allocation order of sections is provided in *Section 3.3 TOC* on page 86.

2.3.2 Function Prologue and Epilogue

A function’s prologue and epilogue are described in this section.

2.3.2.1 Function Prologue

A function's prologue establishes addressability by initializing a TOC pointer in register r2, if necessary, and a stack frame, if necessary, and may save any nonvolatile registers it uses.

All functions have a global entry point (GEP) available to any caller and pointing to the beginning of the prologue. Some functions may have a secondary entry point to optimize the cost of TOC pointer management. In particular, functions within a common module sharing the same TOC base value in r2 may be entered using a secondary entry point (the local entry point or LEP) that may bypass the code that loads a suitable TOC pointer value into the r2 register. When a dynamic or global linker transfers control from a function to another function in the same module, it *may* choose (but is not required) to use the local entry point when the r2 register is known to hold a valid TOC base value. Function pointers shared between modules shall always use the global entry point to specify the address of a function.

When a linker causes control to transfer to a global entry point, it must insert a glue code sequence that loads r12 with the global entry point address. Code at the global entry point can assume that register r12 points to the GEP.

Addresses between the global and local entry points must not be branch targets, either for function entry or referenced by program logic of the function, because a linker may rewrite the code sequence establishing addressability to a different, more optimized form.

For example, while linking a static module with a known load address in the first 2 GB of the address space, the following code sequence may be rewritten:

```
addis r2, r12, .TOC.-func@ha
addi r2, r2, .TOC.-func@l
```

It may be rewritten by a linker or assembler to an equivalent form that is faster due to instruction fusion, such as:

```
lis r2, .TOC.@ha
addi r2, r2, .TOC.@l
```

In addition to establishing addressability, the function prologue is responsible for the following functions:

- Creating a stack frame when required
- Saving any nonvolatile registers that are used by the function
- Saving any limited-access bits that are used by the function, per the rules described in *Section 2.2.1.2* on page 38

This ABI shall be used in conjunction with the Power Architecture that implements the **mfocrf** architecture level. Further, OpenPOWER-compliant processors shall implement implementation-defined bits in a manner to allow the combination of multiple **mfocrf** results with an OR instruction; for example, to yield a word in r0 including all three preserved CRs as follows:

```
mfocrf  r0, crf2
mfocrf  r1, crf3
or r0, r0, r1
mfocrf r1, crf4
or r0, r0, r1
```

Specifically, this allows each OpenPOWER-compliant processor implementation to set each field to hold either 0 or the correct in-order value of the corresponding CR field at the point where the **mfocrf** instruction is performed.

Assembly Language Syntax for Defining Entry Points

When a function has two entry points, the global entry point is defined as a symbol. The local entry point is defined with the .localentry assembler pseudo op.

```
my_func:
       addis r2, r12, (.TOC.-my_func)@ha
       addi r2, r2, (.TOC.-my_func)@l
       .localentry my_func, .-my_func
       ... ; function definition
       blr
```

Section 3.4.1 Symbol Values on page 87 shows how to represent dual entry points in symbol tables in an ELF object file. It also defines the meaning of the second parameter, which is put in the three most-significant bits of the st_other field in the ELF Symbol Table entry.

2.3.2.2 Function Epilogue

The purpose of the epilogue is to perform the following functions:

- Restore all registers and limited-access bits that were saved by the function's prologue.
- Restore the last stack frame.
- Return to the caller.

2.3.2.3 Rules for Prologue and Epilogue Sequences

Set function prologue and function epilogue code sequences are not imposed by this ABI. There are several rules that must be adhered to in order to ensure reliable and consistent call chain backtracing:

- Before a function calls any other function, it shall establish its own stack frame, whose size shall be a multiple of 16 bytes.
  - In instances where a function's prologue creates a stack frame, the back-chain word of the stack frame shall be updated atomically with the value of the stack pointer (r1) when a back chain is implemented. (This must be supported as default by all ELF V2 ABI-compliant environments.) This task can be done by using one of the following Store Doubleword with Update instructions:
    - Store Doubleword with Update instruction with relevant negative displacement for stack frames that are smaller than 32 KB
    - Store Doubleword with Update Indexed instruction where the negative size of the stack frame has been computed, using **addis** and **addi** or **ori** instructions, and then loaded into a volatile register, for stack frames that are 32 KB or greater
  - The function shall save the link register that contains its return address in the LR save doubleword of its caller's stack frame before calling another function.
- The deallocation of a function's stack frame must be an atomic operation. This task can be accomplished by one of the following methods:
  - Increment the stack pointer by the identical value that it was originally decremented by in the prologue when the stack frame was created.
  - Load the stack pointer (r1) with the value in the back-chain word in the stack frame, if a back chain is present.

- The calling sequence does not restrict how languages leverage the local variable space of the stack frame. There is no restriction on the size of this section.
- The parameter save area shall be allocated by the caller. It shall be large enough to contain the parameters needed by the caller if a parameter save area is needed (as described in *Section 2.2.2.3 Optional Save Areas* on page 42). Its contents are not saved across function calls.
- If any nonvolatile registers are to be used by the function, the contents of the register must be saved into a register save area. See *Section 2.2.2.3 Optional Save Areas* on page 42 for information on all of the optional register save areas.

Saving and/or restoring nonvolatile registers used by the function can be accomplished by using in-line code. Alternatively, one of the system subroutines described in *Section 2.3.3 Register Save and Restore Functions* on page 63 may offer a more efficient alternative to in-line code, especially in cases where there are many registers to be saved or restored.

2.3.3 Register Save and Restore Functions

This section describes functions that can be used to save and restore the contents of nonvolatile registers. The use of these routines, rather than performing these saves and restores inline in the prologue and epilogue of functions, can help reduce the code footprint. The calling conventions of these functions are not standard, and the executables or shared objects that use these functions must statically link them.

The register save and restore functions affect consecutive registers from register N through register 31, where N represents a number between 14 and 31. Higher-numbered registers are saved at higher addresses within a save area. Each function described in this section is a family of functions with identical behavior except for the number and kind of registers affected.

Systems must provide three pairs of functions to save and restore general-purpose, floating-point, and vector registers. They may be implemented as multiple-entry-point routines or as individual routines. The specific calling conventions for each of these functions are described in *Section 2.3.3.1 GPR Save and Restore Functions* on page 63, *Section 2.3.3.2 FPR Save and Restore Functions* on page 65, and *Section 2.3.3.3 Vector Save and Restore Functions* on page 67. Visibility rules are described in *Section 5.1.2 Save and Restore Routines* on page 131.

2.3.3.1 GPR Save and Restore Functions

Each _savegpr0_*N* routine saves the general registers from r*N* - r31, inclusive. Each routine also saves the LR. The stack frame must not have been allocated yet. When the routine is called, r1 contains the address of the word immediately beyond the end of the general register save area, and r0 must contain the value of LR on function entry.

The _restgpr0_*N* routines restore the general registers from r*N* - r31, and then return to their caller's caller. The caller's stack frame must already have been deallocated. When the routine is called, r1 contains the address of the word immediately beyond the end of the general register save area, and LR must contain the return address.

A sample implementation of _savegpr0_*N* and _restgpr0_*N* follows:

```
_savegpr0_14:  std  r14,-144(r1)
_savegpr0_15:  std  r15,-136(r1)
_savegpr0_16:  std  r16,-128(r1)
_savegpr0_17:  std  r17,-120(r1)
```

```
_savegpr0_18:  std  r18,-112(r1)
_savegpr0_19:  std  r19,-104(r1)
_savegpr0_20:  std  r20,-96(r1)
_savegpr0_21:  std  r21,-88(r1)
_savegpr0_22:  std  r22,-80(r1)
_savegpr0_23:  std  r23,-72(r1)
_savegpr0_24:  std  r24,-64(r1)
_savegpr0_25:  std  r25,-56(r1)
_savegpr0_26:  std  r26,-48(r1)
_savegpr0_27:  std  r27,-40(r1)
_savegpr0_28:  std  r28,-32(r1)
_savegpr0_29:  std  r29,-24(r1)
_savegpr0_30:  std  r30,-16(r1)
_savegpr0_31:  std  r31,-8(r1)
               std  r0, 16(r1)
               blr

_restgpr0_14:  ld   r14,-144(r1)
_restgpr0_15:  ld   r15,-136(r1)
_restgpr0_16:  ld   r16,-128(r1)
_restgpr0_17:  ld   r17,-120(r1)
_restgpr0_18:  ld   r18,-112(r1)
_restgpr0_19:  ld   r19,-104(r1)
_restgpr0_20:  ld   r20,-96(r1)
_restgpr0_21:  ld   r21,-88(r1)
_restgpr0_22:  ld   r22,-80(r1)
_restgpr0_23:  ld   r23,-72(r1)
_restgpr0_24:  ld   r24,-64(r1)
_restgpr0_25:  ld   r25,-56(r1)
_restgpr0_26:  ld   r26,-48(r1)
_restgpr0_27:  ld   r27,-40(r1)
_restgpr0_28:  ld   r28,-32(r1)
_restgpr0_29:  ld   r0, 16(r1)
               ld   r29,-24(r1)
               mtlr r0
               ld   r30,-16(r1)
               ld   r31,-8(r1)
               blr
_restgpr0_30:  ld   r30,-16(r1)
_restgpr0_31:  ld   r0, 16(r1)
               ld   r31,-8(r1)
               mtlr r0
               blr
```

Each _savegpr1_N routine saves the general registers from rN - r31, inclusive. When the routine is called, r12 contains the address of the word just beyond the end of the general register save area.

The _restgpr1_N routines restore the general registers from rN - r31. When the routine is called, r12 contains the address of the word just beyond the end of the general register save area, superseding the normal use of r12 on a call.

A sample implementation of _savegpr1_N and _restgpr1_N follows:

```
_savegpr1_14:  std  r14,-144(r12)
_savegpr1_15:  std  r15,-136(r12)
_savegpr1_16:  std  r16,-128(r12)
_savegpr1_17:  std  r17,-120(r12)
_savegpr1_18:  std  r18,-112(r12)
_savegpr1_19:  std  r19,-104(r12)
_savegpr1_20:  std  r20,-96(r12)
_savegpr1_21:  std  r21,-88(r12)
_savegpr1_22:  std  r22,-80(r12)
_savegpr1_23:  std  r23,-72(r12)
_savegpr1_24:  std  r24,-64(r12)
_savegpr1_25:  std  r25,-56(r12)
_savegpr1_26:  std  r26,-48(r12)
_savegpr1_27:  std  r27,-40(r12)
_savegpr1_28:  std  r28,-32(r12)
_savegpr1_29:  std  r29,-24(r12)
_savegpr1_30:  std  r30,-16(r12)
_savegpr1_31:  std  r31,-8(r12)
               blr

_restgpr1_14:  ld   r14,-144(r12)
_restgpr1_15:  ld   r15,-136(r12)
_restgpr1_16:  ld   r16,-128(r12)
_restgpr1_17:  ld   r17,-120(r12)
_restgpr1_18:  ld   r18,-112(r12)
_restgpr1_19:  ld   r19,-104(r12)
_restgpr1_20:  ld   r20,-96(r12)
_restgpr1_21:  ld   r21,-88(r12)
_restgpr1_22:  ld   r22,-80(r12)
_restgpr1_23:  ld   r23,-72(r12)
_restgpr1_24:  ld   r24,-64(r12)
_restgpr1_25:  ld   r25,-56(r12)
_restgpr1_26:  ld   r26,-48(r12)
_restgpr1_27:  ld   r27,-40(r12)
_restgpr1_28:  ld   r28,-32(r12)
_restgpr1_29:  ld   r29,-24(r12)
_restgpr1_30:  ld   r30,-16(r12)
_restgpr1_31:  ld   r31,-8(r12)
               blr
```

2.3.3.2 FPR Save and Restore Functions

Each _savefpr_*N* routine saves the floating-point registers from f*N* - f31, inclusive. When the routine is called, r1 contains the address of the word immediately beyond the end of the floating-point register save area, which means that the stack frame must not have been allocated yet. Register r0 must contain the value of LR on function entry.

The _restfpr_*N* routines restore the floating-point registers from f*N* - f31, inclusive. When the routine is called, r1 contains the address of the word immediately beyond the end of the floating-point register save area, which means that the stack frame must not have been allocated yet.

It is incorrect to call both _savefpr_M and _savegpr0_M in the same prologue, or _restfpr_M and _restgpr0_M in the same epilogue. It is correct to call _savegpr1_M and _savefpr_M in either order, and to call _restgpr1_M and then _restfpr_M.

A sample implementation of _savefpr_*N* and _restfpr_*N* follows:

```
_savefpr_14:  stfd f14,-144(r1)
_savefpr_15:  stfd f15,-136(r1)
_savefpr_16:  stfd f16,-128(r1)
_savefpr_17:  stfd f17,-120(r1)
_savefpr_18:  stfd f18,-112(r1)
_savefpr_19:  stfd f19,-104(r1)
_savefpr_20:  stfd f20,-96(r1)
_savefpr_21:  stfd f21,-88(r1)
_savefpr_22:  stfd f22,-80(r1)
_savefpr_23:  stfd f23,-72(r1)
_savefpr_24:  stfd f24,-64(r1)
_savefpr_25:  stfd f25,-56(r1)
_savefpr_26:  stfd f26,-48(r1)
_savefpr_27:  stfd f27,-40(r1)
_savefpr_28:  stfd f28,-32(r1)
_savefpr_29:  stfd f29,-24(r1)
_savefpr_30:  stfd f30,-16(r1)
_savefpr_31:  stfd f31,-8(r1)
              std  r0, 16(r1)
              blr

_restfpr_14:  lfd  f14,-144(r1)
_restfpr_15:  lfd  f15,-136(r1)
_restfpr_16:  lfd  f16,-128(r1)
_restfpr_17:  lfd  f17,-120(r1)
_restfpr_18:  lfd  f18,-112(r1)
_restfpr_19:  lfd  f19,-104(r1)
_restfpr_20:  lfd  f20,-96(r1)
_restfpr_21:  lfd  f21,-88(r1)
_restfpr_22:  lfd  f22,-80(r1)
_restfpr_23:  lfd  f23,-72(r1)
_restfpr_24:  lfd  f24,-64(r1)
_restfpr_25:  lfd  f25,-56(r1)
_restfpr_26:  lfd  f26,-48(r1)
_restfpr_27:  lfd  f27,-40(r1)
_restfpr_28:  lfd  f28,-32(r1)
_restfpr_29:  ld   r0, 16(r1)
              lfd  f29,-24(r1)
              mtlr r0
              lfd  f30,-16(r1)
              lfd  f31,-8(r1)
```

```
              blr
 _restfpr_30: lfd  f30,-16(r1)
 _restfpr_31: ld   r0, 16(r1)
              lfd  f31,-8(r1)
              mtlr r0
              blr
```

2.3.3.3 Vector Save and Restore Functions

Each _savevr_M routine saves the vector registers from vM - v31 inclusive. On entry to this function, r0 contains the address of the word just beyond the end of the vector register save area. The routines leave r0 undisturbed. They modify the value of r12.

The _restvr_M routines restore the vector registers from vM - v31 inclusive. On entry to this function, r0 contains the address of the word just beyond the end of the vector register save area. The routines leave r0 undisturbed. They modify the value of r12. The following code is an example of restoring a vector register.

It is valid to call _savevr_M before any of the other register save functions, or after _savegpr1_M. It is valid to call _restvr_M before any of the other register restore functions, or after _restgpr1_M.

A sample implementation of _savevr_M and _restvr_M follows:

```
_savevr_20:    addi   r12,r0,-192
               stvx   v20,r12,r0          # save v20
_savevr_21:    addi   r12,r0,-176
               stvx   v21,r12,r0          # save v21
_savevr_22:    addi   r12,r0,-160
               stvx   v22,r12,r0          # save v22
_savevr_23:    addi   r12,r0,-144
               stvx   v23,r12,r0          # save v23
_savevr_24:    addi   r12,r0,-128
               stvx   v24,r12,r0          # save v24
_savevr_25:    addi   r12,r0,-112
               stvx   v25,r12,r0          # save v25
_savevr_26:    addi   r12,r0,-96
               stvx   v26,r12,r0          # save v26
_savevr_27:    addi   r12,r0,-80
               stvx   v27,r12,r0          # save v27
_savevr_28:    addi   r12,r0,-64
               stvx   v28,r12,r0          # save v28
_savevr_29:    addi   r12,r0,-48
               stvx   v29,r12,r0          # save v29
_savevr_30:    addi   r12,r0,-32
               stvx   v30,r12,r0          # save v30
_savevr_31:    addi   r12,r0,-16
               stvx   v31,r12,r0          # save v31
               blr                        # return to epilogue

_restvr_20:    addi    r12,r0,-192
               lvx     v20,r12,r0        # restore v20
_restvr_21:    addi    r12,r0,-176
```

```
              lvx     v21,r12,r0          # restore v21
_restvr_22:   addi    r12,r0,-160
              lvx     v22,r12,r0          # restore v22
_restvr_23:   addi    r12,r0,-144
              lvx     v23,r12,r0          # restore v23
_restvr_24:   addi    r12,r0,-128
              lvx     v24,r12,r0          # restore v24
_restvr_25:   addi    r12,r0,-112
              lvx     v25,r12,r0          # restore v25
_restvr_26:   addi    r12,r0,-96
              lvx     v26,r12,r0          # restore v26
_restvr_27:   addi    r12,r0,-80
              lvx     v27,r12,r0          # restore v27
_restvr_28:   addi    r12,r0,-64
              lvx     v28,r12,r0          # restore v28
_restvr_29:   addi    r12,r0,-48
              lvx     v29,r12,r0          # restore v29
_restvr_30:   addi    r12,r0,-32
              lvx     v30,r12,r0          # restore v30
_restvr_31:   addi    r12,r0,-16
              lvx     v31,r12,r0          # restore v31
              blr                         #return to epilogue
```

2.3.4 Function Pointers

A function's address is defined to be its global entry point. Function pointers shall contain the global entry point address.

2.3.5 Static Data Objects

Data objects with static storage duration are described here. Stack-resident data objects are omitted because the virtual addresses of stack-resident data objects are derived relative to the stack or frame pointers. Heap data objects are omitted because they are accessed via a program pointer.

The only instructions that can access memory in the Power Architecture are load and store instructions. Programs typically access memory by placing the address of the memory location into a register and accessing the memory location indirectly through the register because Power Architecture instructions cannot hold 64-bit addresses directly. The values of symbols or their absolute virtual addresses are placed directly into instructions for symbolic references in absolute code.

Absolute addresses are not permitted in position-independent modules. The signed offset into the Global Offset Table of the symbol is held in position-independent instructions that reference symbols. Then, the absolute address of the table entry for the particular symbol can be derived by adding the offset to the appropriate Global Offset Table address using the r2 TOC register in a load instruction. *Figure 2-25* on page 70 shows an example of this method.

Examples of absolute and position-independent compilations are shown in *Figure 2-24*, *Figure 2-25*, and *Figure 2-26*. These examples show the C language statements together with the generated assembly language. The assumption for these figures is that only executables can use absolute addressing while

shared objects must use position-independent code addressing. The figures are intended to demonstrate the compilation of each C statement independent of its context; hence, there can be redundant operations in the code.

Absolute addressing efficiency depends on the memory-region addresses:

Top 32 KB	Addressed directly with load and store D forms.
Top 2 GB	Addressed by a two-instruction sequence consisting of an lis with load and store D forms.
Remaining addresses	More than two instructions.
Bottom 2 GB	Addressed by a two-instruction sequence consisting of an lis with load and store D forms.
Bottom 32 KB	Addressed directly with load and store D forms.

Figure 2-24. Absolute Load and Store Example

C Code	Assembly Code

`extern int src;`
`extern int dst;`
`extern int *ptr;`
 	
`dst = src;`
	`.extern src`
	`.extern dst`
	`.extern ptr`
	`.section ".text"`
	`lis r9,src@ha`
	`lwz r9,src@l(r9)`
	`lis r11,dst@ha`
	`stw r9,dst@l(r11)`

`ptr = &dst;`
 	
`*ptr = src;`
	`lis r11,ptr@ha`
	`lis r9,dst@ha`
	`la r9,dst@l(r9)`
	`std r9,ptr@l(r11)`
	`lis r11,ptr@ha`
	`lwz r11,ptr@l(r11)`
	`lis r9,src@ha`
	`lwz r9,src@l(r9)`
	`stw r9,0(r11)`

Figure 2-25. Small Model Position-Independent Load and Store (DSO)

C Code	Assembly Code
`extern int src;`	`.extern src`
`extern int dst;`	`.extern dst`
`extern int *ptr;`	`.extern ptr`
	`.section ".text"`
	`# TOC base in r2`
`dst = src;`	`ld r9,src@got(2)`
	`lwz r0,0(r9)`
	`ld r9,dst@got(r2)`
	`stw r0,0(r9)`
`ptr = &dst;`	`ld r9,ptr@got(r2)`
	`ld r0,dst@got(r2)`
	`std r0,0(r9)`
`*ptr = src;`	`ld r9,ptr@got(r2)`
	`ld r11,0(r9)`
	`ld r9,src@got(r2)`
	`lwz r0,0(r9)`
	`stw r0,0(r11)`

Figure 2-26. Medium or Large Model Position-Independent Load and Store (DSO) (Page 1 of 2)

C Code	Assembly Code
`extern int src;`	`.extern src`
`extern int dst;`	`.extern dst`
`int *ptr;`	`.extern ptr`
	`.section ".text"`
	`# AssumesTOC pointer in r2`
`dst = src;`	`addis r6,r2,src@got@ha`
	`ld r6,src@got@l(r6)`
	`addis r7,r2,dst@got@ha`
	`ld r7,dst@got@l(r7)`
	`lwz r0,0(r6)`
	`stw r0,0(r7)`
`ptr = & dst;`	`addis r6,r2,dst@got@ha`

Figure 2-26. Medium or Large Model Position-Independent Load and Store (DSO) (Page 2 of 2)

C Code	Assembly Code

*ptr = src;	ld r6,dst@got@l(r6)
	addis r7,r2,ptr@got@ha
	ld r7,ptr@got@l(r7)
	stw r6,0(r7)
	addis r6,r2,src@got@ha
	ld r6,src@got@l(r6)
	addis r7,r2,ptr@got@ha
	ld r7,ptr@got@l(r7)
	ld r7,0(r7)
	lwz r0,0(r6)
	stw r0,0,(r7)

Notes:

- Due to fusion hardware support, the preferred code forms are destructive[a] addressing forms with an addis specifying a set of high-order bits followed immediately by a destructive load using the same target register as the addis instruction to load data from a signed 32-bit offset from a base register.
- For PIC code (see *Figure 2-25* and *Figure 2-26*), the offset in the Global Offset Table where the value of the symbol is stored is given by the assembly syntax symbol@got. This syntax represents the address of the variable named “symbol”.

  The offset for this assembly syntax cannot be any larger than 16 bits. In cases where the offset is greater than 16 bits, the following assembly syntax is used for offsets up to 32 bits:

  – High (32-bit) adjusted part of the offset: symbol@got@ha
  Causes a linker error if the offset is larger than 32 bits.

  – High (32-bit) part of the offset: symbol@got@h
  Causes a linker error if the offset is larger than 32 bits.

  – Low part of the offset: symbol@got@l

  To obtain the multiple 16-bit segments of a 64-bit offset, the following operators may be used:

  – Highest (most-significant 16 bits) adjusted part of the offset: symbol@highesta

  – Highest (most-significant 16 bits) part of the offset: symbol@highest

  – Higher (next significant 16 bits) adjusted part of the offset: symbol@highera

  – Higher (next significant 16 bits) part of the offset: symbol@higher

  – High (next significant 16 bits) adjusted part of the offset: symbol@higha

  – High (next significant 16 bits) part of the offset: symbol@high

  – Low part of the offset: symbol@l

  If the instruction using symbol@got@l has a signed immediate operand (for example, addi), use symbol@got@**ha** (high adjusted) for the high part of the offset. If it has an unsigned immediate operand (for example, ori), use symbol@got@**h**. For a description of high-adjusted values, see *Section 3.5.2 Relocation Notations* on page 91.

a. Destructive in this context refers to a code sequence where the first intermediate result computed by a first instruction is overwritten (that is, “destroyed”) by the result of a second instruction so that only one result register is produced. Fusion can then give the same performance as a single load instruction with a 32-bit displacement.

2.3.6 Function Calls

Direct function calls are made in programs with the Power Architecture bl instruction. A bl instruction can reach 32 MB backwards or forwards from the current position due to a self-relative branch displacement in the instruction. Therefore, the size of the text segment in an executable or shared object is constrained when a bl instruction is used to make a function call. When the distance of the called function exceeds the displacement reach of the bl instruction, a linker implementation may either introduce branch trampoline code to extend function call distances or issue a link error.

As shown in *Figure 2-27*, the bl instruction is generally used to call a local function.

Two possibilities exist for the location of the function with respect to the caller:

1. The called function is in the same executable or shared object as the caller. In this case, the symbol is resolved by the link editor and the bl instruction branches directly to the called function as shown in *Figure 2-27*.

Figure 2-27. Direct Function Call

C Code	Assembly Code
`extern void function();` `function();`	`bl function` `nop`

2. The called function is not in the same executable or shared object as the caller. In this case, the symbol cannot be directly resolved by the link editor. The link editor generates a branch to glue code that loads the address of the function from the Procedure Linkage Table. See *Section 4.2.5 Procedure Linkage Table* on page 127.

For indirect function calls, the address of the function to be called is placed in r12 and the CTR register, and a bctrl instruction is used to perform the indirect branch as shown in *Figure 2-28*, *Figure 2-29*, and *Figure 2-30*. The ELF V2 ABI requires the address of the called function to be in r12 when a cross-module function call is made.

Figure 2-28. Indirect Function Call (Absolute Medium Model)

C Code	Assembly Code

`extern void function();`
`extern void (*ptrfunc) ();`
 	
`ptrfunc = function;`
 	
`(*ptrfunc)();`
	`.section .text`
	`lis r11,ptrfunc@ha`
	`lis r9,function@ha`
	`ld r9,function@l(r9)`
	`std r9,ptrfunc@l(r11)`
	`lis r12,ptrfunc@ha`
	`ld r12,ptrfunc@l(r12)`
	`mtctr r12`
	`bctrl`

Figure 2-29 on page 73 shows how to make an indirect function call using small model position-independent code.

Figure 2-29. Small Model Position-Independent Indirect Function Call (Page 1 of 2)

C Code	Assembly Code
`extern void function();` `extern void (*ptrfunc) ();`	`.section .text`

Figure 2-29. Small Model Position-Independent Indirect Function Call (Page 2 of 2)

C Code	Assembly Code
	/* TOC pointer is in r2 */
ptrfunc = function;	ld r9,ptrfunc@got(r2)
	ld r0,function@got(r2)
	std r0,0(r9)
...	...
(*ptrfunc) ();	ld r9,ptrfunc@got(r2)
	ld r12,0(r9)
	mtctr r12
	std r2,24(r1)
	bctrl
	ld r2,24(r1)

Figure 2-30 shows how to make an indirect function call using large model position-independent code.

Figure 2-30. Large Model Position-Independent Indirect Function Call

C Code	Assembly Code
extern void function();	
extern void (*ptrfunc) ();	
ptrfunc=function;	addis r9,r2,ptrfunc@got@ha
	ld r9,ptrfunc@got@l(r9)
	addis r12,r2,function@got@ha
	ld r12,function@got@l(r12)
	std r12,0(r9)
(*ptrfunc) ();	addis r9,r2,ptrfunc@got@ha
	ld r9,ptrfunc@got@l(r9)
	ld r12,0(r9)
	std r2,24(r1)
	mtctr r12
	bctrl
	ld r2,24(r1)

Function calls need to be performed in conjunction with establishing, maintaining, and restoring addressability through the TOC pointer register, r2. When a function is called, the TOC pointer register may be modified. The caller must provide a nop after the bl instruction performing a call, if r2 is not known to have the

same value in the callee. This is generally true for external calls. The linker will replace the nop with an r2 restoring instruction if the caller and callee use different r2 values, The linker leaves it unchanged if they use the same r2 value. This scheme avoids having a compiler generate an overconservative r2 save and restore around every external call.

For calls to functions resolved at runtime, the linker must generate stub code to load the function address from the PLT.

The stub code also must save r2 to 24(r1) unless the call is marked with an R_PPC64_TOCSAVE relocation that points to a nop provided in the caller's prologue. In that case, the stub code can omit the r2 save. Instead, the linker replaces the prologue nop with an r2 save.

The linker may assume that r2 is valid at the point of a call. Thus, stub code may use r2 to load an address from the PLT unless the call is marked with an R_PPC64_REL24_NOTOC relocation to indicate that r2 is not available.

The nop instruction must be:

```
ori r0,r0,0
```

For more information, see *Section 2.3.2.1 Function Prologue* on page 60, *Section 3.4.1 Symbol Values* on page 87, and *Table 3-2* on page 94.

2.3.7 Branching

The flow of execution in a program is controlled by the use of branch instructions. Unconditional branch instructions can jump to locations up to 32 MB in either direction because they hold a signed value with a 64 MB range that is relative to the current location of the program execution.

Figure 2-31 shows the model for branch instructions.

Figure 2-31. Branch Instruction Model

C Code	Assembly Code

`label:`
`...`
`goto label;`
	`.L01:`
	`...`
	`b .L01`

Selecting one of multiple branches is accomplished in C with switch statements. An address table is used by the compiler to implement the switch statement selections in cases where the case labels satisfy grouping constraints. In the examples that follow, details that are not relevant are avoided by the use of the following simplifying assumptions:

- r12 holds the selection expression.
- Case label constants begin at zero.
- The assembler names .Lcasei, .Ldefault, and .Ltab are used for the case labels, the default, and the address table respectively.

For position-dependent code (for example, the main module of an application) loaded into the low or high address range, absolute addressing of a branch table yields the best performance.

Figure 2-32. Absolute Switch Code (Within) for static modules located in low or high 2 GB of address space

C Code	Assembly Code	
switch(j)	cmplwi	r12, 4
{	bge	.Ldefault
case 0:	slwi	r12, 2
...	addis	r12, r12, .Ltab@ha
case 1:	lwa	r12, .Ltab@l(r12)
...	mtctr	r12
case 3:	bctr	
...	.rodata	
default:	.Ltab:	
...	.long	.Lcase0
}	.long	.Lcase1
	.long	.Ldefault
	.long	.Lcase3
	.text	

Note: A faster variant of this code may be used to locate branch targets in the bottom 2 GB of the address space in conjunction with the lwz instruction in place of the lwa instruction.

Figure 2-33. Absolute Switch Code (Beyond) (Page 1 of 2) for static modules beyond the top or bottom 2 GB of the address space

C Code	Assembly Code	
switch(j)	cmplwi	r12, 4
{	bge	.Ldefault
case 0:	slwi	r12, 2
...	addis	r12, r12, .Ltab@ha
case 1:	ld	r12, .Ltab@l(r12)
...	mtctr	r12
case 3:	bctr	
...	.rodata	
default:	.Ltab:	
...	.quad	.Lcase0
}	.quad	.Lcase1

Figure 2-33. Absolute Switch Code (Beyond) (Page 2 of 2) for static modules beyond the top or bottom 2 GB of the address space

C Code	Assembly Code	
	.quad	.Ldefault
	.quad	.Lcase3
	.text	

For position-independent code targeted at being dynamically loaded to different address ranges as DSO, the preferred code pattern uses TOC-relative addressing by taking advantage of the fact that the TOC pointer points to a fixed offset from the code segment. The use of relative offsets from the start address of the branch table ensures position-independence when code is loaded at different addresses.

Figure 2-34. Position-Independent Switch Code for Small/Medium Models (preferred with TOC-relative addressing)

C Code	Assembly Code	
switch(j)	cmplwi	r12, 4
{	bge	.Ldefault
case 0:	addis	r10,r2,(.Ltab-.TOC.)@ha
...	addi	r10,r10,(.Ltab-.TOC.)@l
case 1:	slwi	r12,2
...	lwax	r8,r10,r12
case 3:	add	r10,r8,r10
...	mtctr	r10
default:	bctr	
...	.Ltab:	
}	.word	(.Lcase0-.Ltab)
	.word	(.Lcase1-.Ltab)
	.word	(.Ldefault-.Ltab)
	.word	(.Lcase3-.Ltab)

For position-independent code targeted at being dynamically loaded to different address ranges as a DSO or a position-independent executable (PIE), the preferred code pattern uses TOC-indirect addresses for code models where the distance between the TOC and the branch table exceeds 2 GB. The use of relative offsets from the start address of the branch table ensures position independence when code is loaded at different addresses.

Figure 2-35. Position-Independent Switch Code for All Models (alternate, with GOT-indirect addressing)

C Code	Assembly Code	
switch(j)	cmplwi	r12, 4
{	bge	.Ldefault
case 0:	addis	r10,r2,.Ltab@got@ha
...	ld	r10,.Ltab@got@l(r10)
case 1:	slwi	r12,2
...	lwax	r8,r10,r8
case 3:	add	r10,r8,r12
...	mtctr	r10
default:	bctr	
...	.Ltab:	
}	.word	(.Lcase0-.Ltab)
	.word	(.Lcase1-.Ltab)
	.word	(.Ldefault-.Ltab)
	.word	(.Lcase3-.Ltab)

Figure 2-36 shows how, in the medium code model, PIC code can be used to avoid using the lwa instruction, which may result in lower performance in some POWER processor implementation.

Figure 2-36. PIC Code that Avoids the lwa Instruction

```
    .text
f1:
     addis r9,r2,.Ltab@ha
     sldi r10,r3,2
     addi r9,r9,.Ltab@l
     lwzx r10,r10,r9
     sub r10,r2,r10
     mtctr r10
     bctr

.Ltab:
     .long .TOC. - Lcase0
     .long .TOC. - Lcase1
     .long .TOC. - Ldefault
     .long .TOC. - Lcase13
```

2.3.8 Dynamic Stack Space Allocation

When allocated, a stack frame may be grown or shrunk dynamically as many times as necessary across the lifetime of a function. Standard calling conventions must be maintained because a subfunction can be called after the current frame is grown and that subfunction may stack, grow, shrink, and tear down a frame between dynamic stack frame allocations of the caller. The following constraints apply when dynamically growing or shrinking a stack frame:

- Maintain 16-byte alignment.
- Stack pointer adjustments shall be performed atomically so that at all times the value of the back-chain word is valid, when a back chain is used.
- Maintain addressability to the previously allocated local variables in the presence of multiple dynamic allocations or conditional allocations.
- Ensure that other linkage information is correct, so that the function can return or its stack space can be deallocated by exception handling without deallocating any dynamically allocated space.

Note: Using a frame pointer is the recognized method for maintaining addressability to arguments or local variables. (This may be a pointer to the top of the stack frame, typically in r31.) For correct behavior in the cases of setjmp() and longjmp(), the frame pointer shall be allocated in a nonvolatile general-purpose register.

Figure 2-37 on page 80 shows the organization of a stack frame before a dynamic allocation.

Figure 2-37. Before Dynamic Stack Allocation

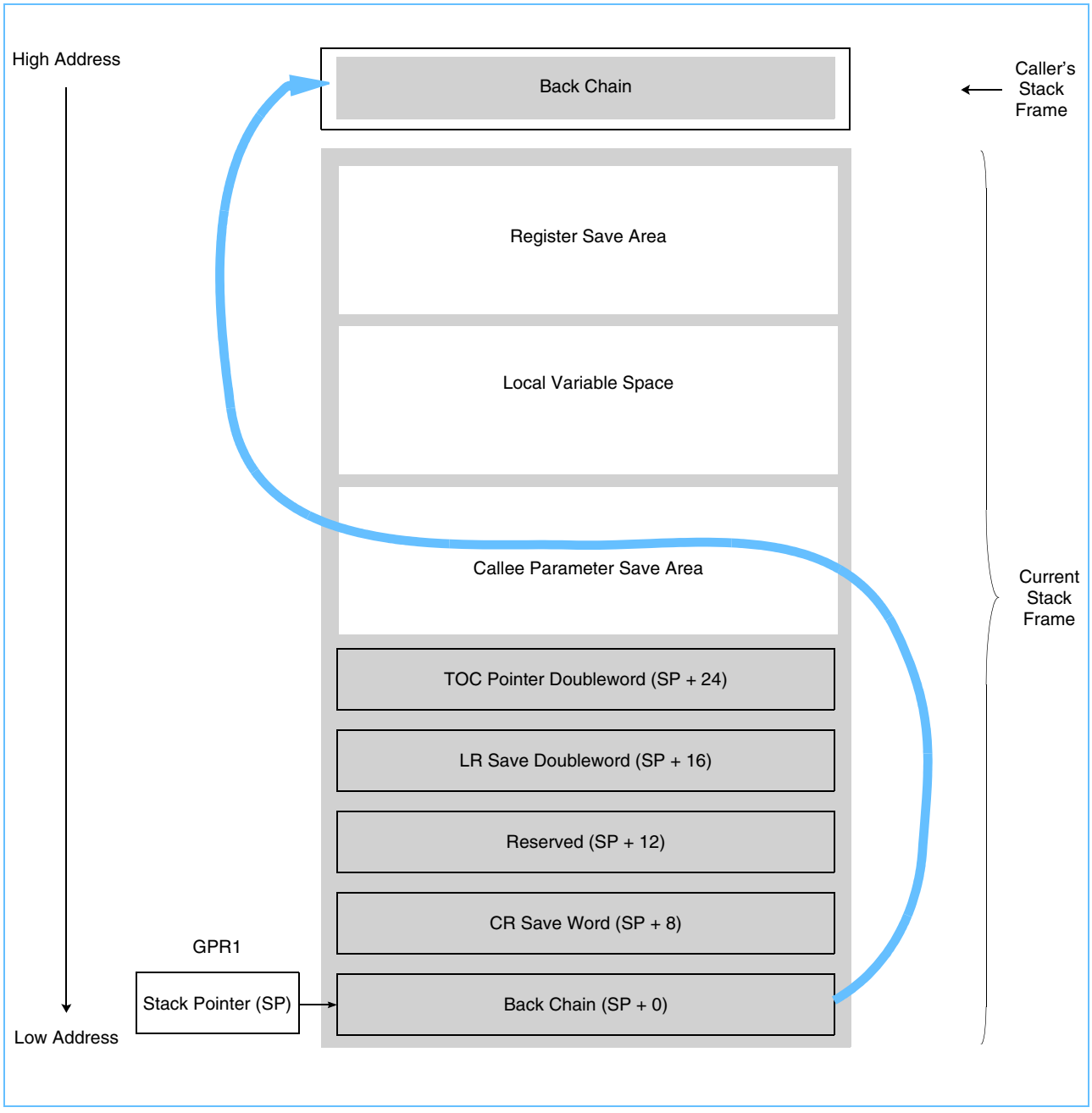


Figure 2-38. Example Code to Allocate n Bytes

```
#define n 13
; char *a = alloca(n);
; rnd(x) = round x to be multiple of stack alignment
; psave = size of parameter save area (may be zero).
p = 32 + rnd(sizeof(psave)+15); Offset to the start of the dynamic allocation
```

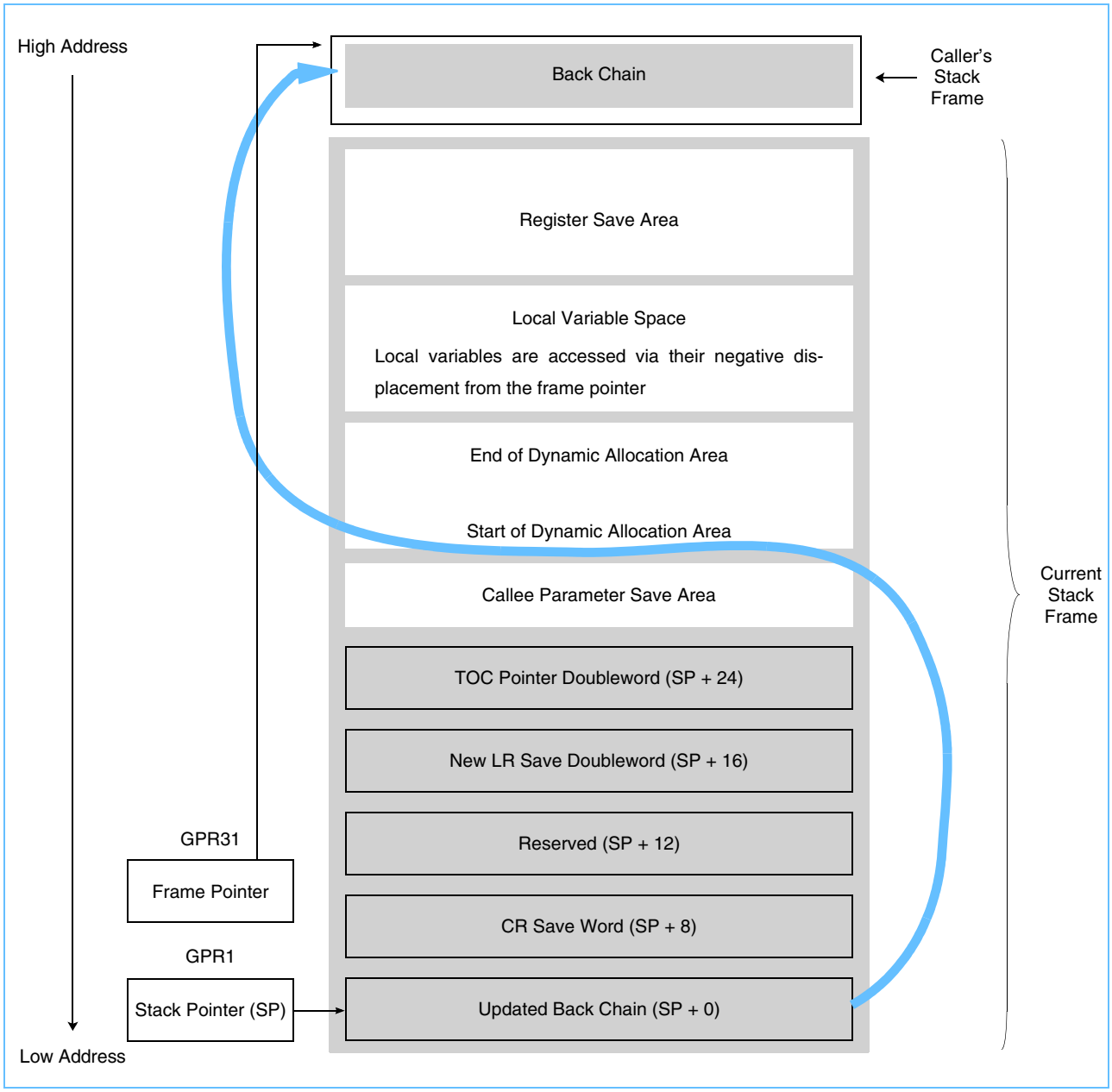
```
ld              r0,0(r1)         ; Load
stdu            r0,-rnd(n+15)(r1); Store new back chain, quadword-aligned.
addi            r3,r1,p          ; R3 = new data area following parameter save area.
```

Because it is allowed (and common) to return without first deallocating this dynamically allocated memory, all the linkage information in the new location must be valid. Therefore, it is also necessary to copy the CR save word and the TOC pointer doubleword from their old locations to the new. It is not necessary to copy the LR save doubleword because, until this function makes a call, it does not contain a value that needs to be preserved. In the future, if it is defined and if the function uses the Reserved word, the LR save doubleword must also be copied.

Note: Additional instructions will be necessary for an allocation of variable size. If a dynamic deallocation will occur, the r1 stack pointer must be saved before the dynamic allocation, and r1 reset to that by the deallocation. The deallocation does not need to copy any stack locations because the old ones should still be valid.

Figure 2-39 on page 82 shows an example organization of a stack frame after a dynamic allocation.

Figure 2-39. After Dynamic Stack Allocation



2.4 DWARF Definition

Although this ABI itself does not define a debugging format, debug with arbitrary record format (DWARF) is defined here for systems that implement the DWARF specification. For information about how to locate the specification, see *Section 1.1 Reference Documentation* on page 17.

The DWARF specification is used by compilers and debuggers to aid source-level or symbolic debugging. However, the format is not biased toward any particular compiler or debugger. Per the DWARF specification, a mapping from Power Architecture registers to register numbers is required as described in *Table 2-17*.

Special Purpose Registers (SPRs) are mapped into DWARF as 100 plus their SPR number. Performance Monitor Registers (PMRs) are mapped into DWARF as 2048 plus the PMR number.

All instances of the Power Architecture use the mapping shown in *Table 2-17* for encoding registers into DWARF. DWARF register numbers 32 - 63 and 77 - 106 are also used to indicate the location of variables in VSX registers vsr0 - vsr31 and vsr32 - vsr63, respectively, in DWARF debug information.

Table 2-17. Mappings of Common Registers

DWARF	Register Number	Register Name	Register Width (Bytes)
Reg	0 - 31	r0 - r31	8
Reg	32 - 63	f0 - f31	8
Reg	64	Reserved	N/A
Reg	65	lr	8
Reg	66	ctr	8
Reg	67	Reserved	N/A
Reg	68 - 75	cr0 - cr7	0.5[1]
Reg	76	xer	4
Reg	77 - 108	vr0 - vr31	16
Reg	109	Reserved	N/A
Reg	110	vscr	8
Reg	111	Reserved	N/A
Reg	112	Reserved	N/A
Reg	113	Reserved	N/A
Reg	114	tfhar	8
Reg	115	tfiar	8
Reg	116	texasr	8

1. The CRx registers correspond to 4-bit fields within a word where the offset of the 4-bit group within a word is a function of the CRFx number (x).

DWARF for the OpenPOWER ABI defines the address class codes described in *Table 2-18*.

Table 2-18. Address Class Codes

Code	Value	Meaning
ADDR_none	0	No class specified

2.5 Exception Handling

Where exceptions can be thrown or caught by a function, or thrown through that function, or where a thread can be canceled from within a function, the locations where nonvolatile registers have been saved must be described with unwind information. The format of this information is based on the DWARF call frame information with extensions.

Any implementation that generates unwind information must also provide exception handling functions that are the same as those described in the Itanium C++ ABI, the normative text on the issue. For information about how to locate this material, see *Section 1.1 Reference Documentation* on page 17.

3. Object Files

3.1 ELF Header

The file class member of the ELF header identification array, e_ident[EI_CLASS], identifies the ELF file as 64-bit encoded by holding the value ELFCLASS64.

For a big-endian encoded ELF file, the data encoding member of the ELF header identification array, e_ident[EI_DATA], holds the value 2, defined as data encoding ELFDATA2MSB. For a little-endian encoded ELF file, it holds the value 1, defined as data encoding ELFDATA2LSB.

```
e_ident[EI_CLASS]   ELFCLASS64   For all 64-bit implementations.
e_ident[EI_DATA]    ELFDATA2MSB  For all big-endian implementations.
e_ident[EI_DATA]    ELFDATA2LSB  For all little-endian implementations.
```

The ELF header's e_flags member holds bit flags associated with the file. The 64-bit PowerPC processor family defines the following flags.

E_flags defining the ABI level:

0 For ELF object files of an unspecified nature.

1 For the Power ELF V1 ABI using function descriptors. This ABI is currently only used for big-endian PowerPC implementations.

2 For the OpenPOWER ELF V2 ABI using the facilities described here and including function pointers to directly reference functions.

The ABI version to be used for the ELF header file is specified with the .abiversion pseudo-op:

```
.abiversion 2
```

Processor identification resides in the ELF header's e_machine member, and must have the value EM_PPC64, defined as the value 21.

3.2 Special Sections

Table 3-1 lists the sections that are used in the Power Architecture to hold program and control information. It also shows their types and attributes.

Table 3-1. Special Sections (Page 1 of 2)

Section Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.toc	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.plt[1]	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE

1. The type of the OpenPOWER ABI .plt section is SHT_NOBITS, not SHT_PROGBITS as on most other processors.

Table 3-1. Special Sections (Page 2 of 2)

Section Name	Type	Attributes
.sbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.bss1	SHT_NOBITS	SHF_ALLOC + SHF_WRITE

1. The type of the OpenPOWER ABI .plt section is SHT_NOBITS, not SHT_PROGBITS as on most other processors.

Suggested uses of these special sections follow:

- The .got section may hold the Global Offset Table (GOT). This section is not normally present in a relocatable object file because it is linker generated. The linker must ensure that .got is aligned to an 8-byte boundary. In an executable or shared library, it may contain part or all of the TOC. For more information, see *Section 2.3 Coding Examples* on page 57 and *Section 4.2.3 Global Offset Table* on page 125.
- The .toc section may hold the initialized TOC. The .toc section must be aligned to an 8-byte boundary. Address elements within .toc must be aligned to 8-byte boundaries to support linker optimization of the .toc section. In a relocatable object file, .toc may contain addresses of objects and functions; in this respect it may be thought of as a compiler-managed GOT. It may also contain other constants or variables; in this respect it is like .sdata. In an executable or shared library, it may contain part or the entirety of the TOC. For more information, see *Section 3.3 TOC* on page 86, *Section 2.3 Coding Examples* on page 57, and *Section 4.2.3 Global Offset Table* on page 125.
- The .plt section may hold the procedure linkage table. This section is not normally present in a relocatable object file because it is linker generated. Each entry within the .plt section is an 8-byte address. The linker must ensure that .plt is aligned to an 8-byte boundary. For more information, see *Section 4.2.5 Procedure Linkage Table* on page 127.
- The .sdata section may hold initialized small-sized data. For more information, see *Section 3.4.2 Use of the Small Data Area* on page 89.
- The .sbss section may hold uninitialized small-sized data.
- The .data1 section may hold initialized medium-sized data.
- The .bss1 section may hold uninitialized medium-sized data.

Tools that support this ABI are not required to use these sections. However, if a tool uses these sections, it must assign the types and attributes specified in *Table 3-1*. Tools are not required to use the sections precisely as suggested. Relocation information and the code that refers to it define the actual use of a section.

3.3 TOC

The TOC is part of the data segment of an executable program.

This section describes a common layout of the TOC in an executable file or shared object. Particular tools are not required to follow the layout specified here.

The TOC region commonly includes data items within the .got, .toc, .sdata, and .sbss sections. In the medium code model, they can be addressed with 32-bit signed offsets from the TOC pointer register. The TOC pointer register typically points to the beginning of the .got section + 0x8000, which permits a 2 GB TOC with the medium and large code models. The .got section is typically created by the link editor based on @got relocations. The .toc section is typically included from relocatable object files referenced during the link phase.

The TOC may straddle the boundary between initialized and uninitialized data in the data segment. The common order of sections in the data segment, some of which may be empty, follows:

```
.rodata
.data
.data1
.got
.toc
.sdata
.sbss
.plt
.bss1
.bss
```

The medium code model is expected to provide a sufficiently large TOC to provide all data addressing needs of a module with a single TOC.

Compilers may generate two-instruction medium code model references (or, if selected, short displacement one-instruction references) for all data items that are in the TOC for the object file being compiled. Such references are relative to the TOC pointer register, r2. (The linker may optimize two-instruction forms to one instruction forms, replacing a first instruction of the two instruction form with a nop and rewriting the second instruction. Consequently, the TOC pointer must be live during the first and second instruction of a two-instruction reference.)

Modules Containing Multiple TOCs

The link editor may create multiple TOCs. In such a case, the constituent .got, .toc, .sdata, and .sbss sections are conceptually repeated as necessary, with each TOC typically using a TOC pointer value of its base plus 0x8000. Any constituent section of type SHT_NOBITS in any TOC but the last is converted to type SHT_PROGBITS filled with zeros.

When multiple TOCs are present, linking must take care to save, initialize, and restore TOC pointers within a single module when calling from one function to a second function using a different TOC pointer value. Many of the same issues associated with a cross-module call apply also to calls within a module but using different TOC pointers.

3.4 Symbol Table

3.4.1 Symbol Values

An executable file that contains a symbol reference that is to be resolved dynamically by an associated shared object will have a symbol table entry for that symbol. This entry will identify the symbol as undefined by setting the st_shndx member to SHN_UNDEF.

The OpenPOWER ABI uses the three most-significant bits in the symbol st_other field to specify the number of instructions between a function's global entry point and local entry point. The global entry point is used when it is necessary to set up the TOC pointer (r2) for the function. The local entry point is used when r2 is known to already be valid for the function. A value of zero in these bits asserts that the function does not use r2.

The st_other values have the following meanings:

0 The local and global entry points are the same, and the function has a single entry point with no requirement on r12 or r2. On return, r2 will contain the same value as at entry.

This value should be used for functions that do not require the use of a TOC register to access external data. In particular, functions that do not access data through the TOC pointer can use a common entry point for the local and global entry points.

Note: If the function is not a leaf function, it must call subroutines using the R_PPC64_REL24_NOTOC relocation to indicate that the TOC register is not initialized. In turn, this may lead to more expensive procedure linkage table (PLT) stub code than would be necessary if a TOC register were initialized.

1 The local and global entry points are the same, and r2 should be treated as caller-saved for local and global callers.

2 The local entry point is at one instruction past the global entry point.

When called at the global entry point, r12 must be set to the function entry address. r2 will be set to the TOC base that this function needs, so it must be preserved and restored by the caller.

When called at the local entry point, r12 is not used and r2 must already point to the TOC base that this function needs, and it will be preserved.

3 The local entry point is at two instructions past the global entry point.

When called at the global entry point, r12 must be set to the function entry address. r2 will be set to the TOC base that this function needs, so it must be preserved and restored by the caller.

When called at the local entry point, r12 is not used and r2 must already point to the TOC base that this function needs, and it will be preserved.

4 The local entry point is at four instructions past the global entry point.

When called at the global entry point, r12 must be set to the function entry address. r2 will be set to the TOC base that this function needs, so it must be preserved and restored by the caller.

When called at the local entry point, r12 is not used and r2 must already point to the TOC base that this function needs, and it will be preserved.

5 The local entry point is at eight instructions past the global entry point.

When called at the global entry point, r12 must be set to the function entry address. r2 will be set to the TOC base that this function needs, so it must be preserved and restored by the caller.

When called at the local entry point, r12 is not used and r2 must already point to the TOC base that this function needs, and it will be preserved.

6 The local entry point is at 16 instructions past the global entry point.

When called at the global entry point, r12 must be set to the function entry address. r2 will be set to the TOC base that this function needs, so it must be preserved and restored by the caller.

When called at the local entry point, r12 is not used and r2 must already point to the TOC base that this function needs, and it will be preserved.

7 Reserved

The local-entry-point handling field of st_other is generated with the .localentry pseudoop:

```
    .globl my_func
    .type   my_func, @function
my_func:
    addis r2, r12 @ha(.TOC.-my_func)
    addi r2, r2 @l(.TOC.-my_func)
    .localentry my_func, .-my_func
    ... ; function definition
    blr
```

Functions called via symbols with an st_other value of 0 may be called without a valid TOC pointer in r2. Symbols of functions that require a local entry with a valid TOC pointer should generate a symbol with an st_other field value of 2 - 6 and both local and global entry points, even if the global entry point will not be used. (In such a case, the instructions of the global entry setup sequence may optionally be initialized with TRAP instructions.)

3.4.2 Use of the Small Data Area

For a data item in the .sdata or .sbss sections, a compiler may generate short-form one-instruction references. In an executable file or shared library, such a reference is relative to the address of the TOC base symbol (which can be obtained from r2 if a TOC pointer is initialized. A compiler that generates code using the small data area should provide an option to select the maximum size of objects placed in the small data area, and a means of disabling any use of the small data area. When generating code for ELF shared libraries, the small data area should not be used for default-visibility global objects. This is to satisfy ELF shared-library symbol interposition rules. That is, an ordinary global symbol in a shared library may be overridden by a symbol of the same name defined in the executable or another shared library. Supporting interposition when using TOC-pointer relative addressing would require text relocations.

3.5 Relocation Types

The relocation entries in a relocatable file are used by the link editor to transform the contents of that file into an executable file or a shared object file. The application and result of a relocation are similar for both. Several relocatable files may be combined into one output file. The link editor merges the content of the files, sets the value of all function symbols, and performs relocations.

The 64-bit OpenPOWER Architecture uses Elf64_Rela relocation entries exclusively. A relocation entry may operate upon a halfword, word, or doubleword. The r_offset member of the relocation entry designates the first byte of the address affected by the relocation. The subfield of r_offset affected by a relocation is implicit in the definition of the applied relocation type. The r_addend member of the relocation entry serves as the relocation addend, which is described in *Section 3.5.1* for each relocation type.

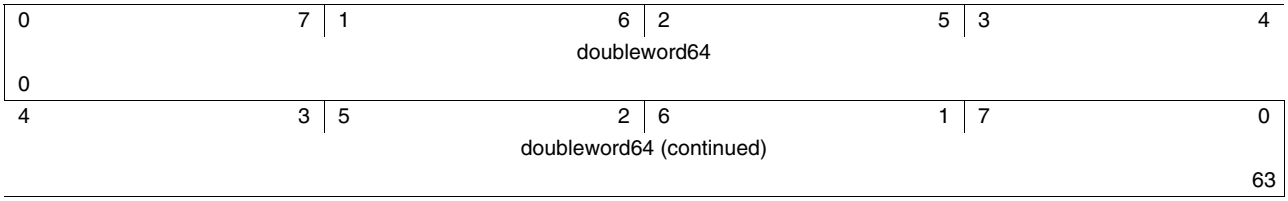
A relocation type defines a set of instructions and calculations necessary to alter the subfield data of a particular relocation field.

3.5.1 Relocation Fields

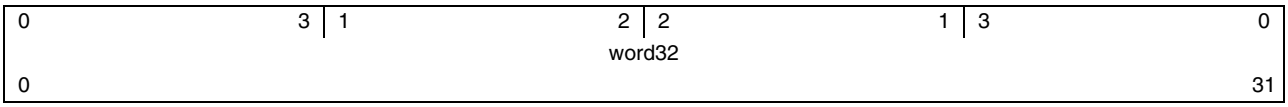
The following relocation fields identify a subfield of an address affected by a relocation.

Bit numbers are shown at the bottom of the boxes. Byte numbers are shown in the top of the boxes; big-endian byte numbers are displayed in the upper left corners and little-endian in the upper right corners. The byte order specified in a relocatable file's ELF header applies to all the elements of a relocation entry, the relocation field definitions, and relocation type calculations.

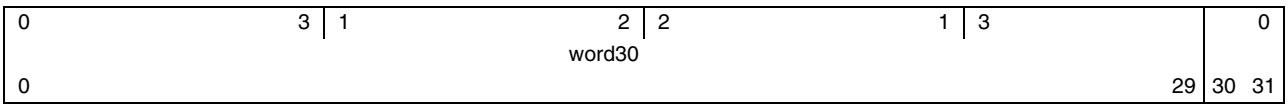
doubleword64 specifies a 64-bit field occupying 8 bytes, the alignment of which is 8 bytes unless otherwise specified.



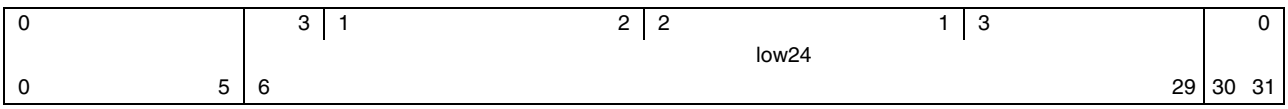
word32 specifies a 32-bit field taking up 4 bytes and maintaining 4-byte alignment unless otherwise indicated.



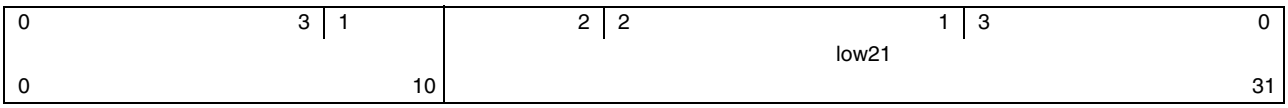
word30 specifies a 30-bit field taking up bits 0 - 29 of a word and maintaining 4-byte alignment unless otherwise indicated.



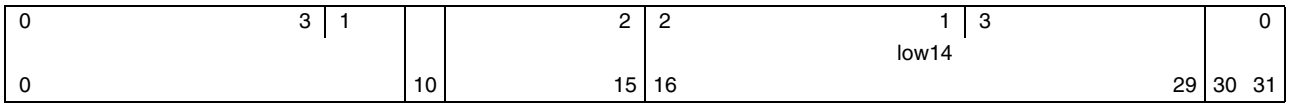
low24 specifies a 24-bit field taking up bits 6 - 29 of a word and maintaining 4-byte alignment. The other bits remain unchanged. A call or unconditional branch instruction is an example of this field.



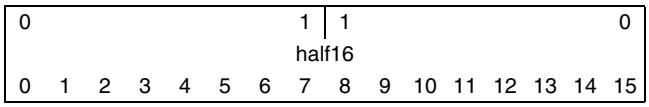
low21 specifies a 21-bit field occupying the least-significant bits of a word with 4-byte alignment.



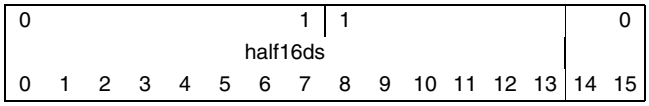
low14 specifies a 14-bit field taking up bits 16 - 29 and possibly bit 10 (the branch prediction bit) of a word and maintaining 4-byte alignment. The other bits remain unchanged. A conditional branch instruction is an example usage.



half16 specifies a 16-bit field taking up two bytes and maintaining 2-byte alignment. The immediate field of an Add Immediate instruction is an example of this field.



half16ds is similar to half16, but really just 14 bits because the two least-significant bits must be zero and are not really part of the field. (Used by, for example, the ldu instruction.) In addition to the use of this relocation field with the DS forms, half16ds relocations are also used in conjunction with DQ forms. In those instances, the linker and assembler collaborate to create valid DQ forms. They raise an error if the specified offset does not meet the constraints of a valid DQ instruction form displacement.



3.5.2 Relocation Notations

The following notations are used in the relocation table.

A	Represents the addend used to compute the value of the relocatable field.
B	Represents the base address at which a shared object file has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different. See Program Header in the System V ABI for more information about the base address.
G	Represents the offset from .TOC at which the address of the relocation entry's symbol resides during execution. This implies the creation of a .got section. For more information, see *Section 2.3 Coding Examples* on page 57 and *Section 4.2.3 Global Offset Table* on page 125. Reference in a calculation to the value G implicitly creates a GOT entry for the indicated symbol.
L	Represents the section offset or address of the procedure linkage table entry for the symbol. This implies the creation of a .plt section if one does not already exist. It also implies the creation of a procedure linkage table (PLT) entry for resolving the symbol. For an unresolved symbol, the PLT entry points to a PLT resolver stub. For a resolved symbol, a procedure linkage table entry holds the final effective address of a dynamically resolved symbol (see *Section 4.2.5 Procedure Linkage Table* on page 127).
M	Similar to G, except that the address that is stored may be the address of the procedure linkage table entry for the symbol.

P	Represents the place (section offset or address) of the storage unit being relocated (computed using r_offset).
R	Represents the offset of the symbol within the section in which the symbol is defined (its section-relative address).
S	Represents the value of the symbol whose index resides in the relocation entry.
+	Denotes 32-bit modulus addition.
-	Denotes 32-bit modulus subtraction.
>>	Denotes arithmetic right-shifting.
#lo(value)	Denotes the least-significant 16 bits of the indicated value. That is: #lo(x) = (x & 0xffff).
#hi(value)	Denotes bits 16 - 63 of the indicated value. That is: #hi(x) = x >> 16
#ha(value)	Denotes the high adjusted value: bits 16 - 63 of the indicated value, compensating for #lo() being treated as a signed number. That is: #ha(x) = (x + 0x8000) >> 16
TP	The value of the thread pointer in general-purpose register r13.
TLS_TP_OFFSET	The constant value 0x7000, representing the offset (in bytes) of the location that the thread pointer is initialized to point to, relative to the start of the thread local storage for the first initially available module.
TCB_LENGTH	The constant value 0x8, representing the length of the thread control block (TCB) in bytes.
tcb	Represents the base address of the TCB. `tcb = (tp - (TLS_TP_OFFSET + TCB_LENGTH))`
dtv	Represents the base address of the dynamic thread vector (DTV). `dtv = tcb[0]`
dtpmod	Represents the load module index of the load module that contains the definition of the symbol being relocated and is used to index the DTV.
dtprel	Represents the offset of the symbol being relocated relative to the value of dtv[dtpmod]. `dtv[dtpmod] + dtprel = (S + A)`
tprel	Represents the offset of the symbol being relocated relative to the TP. `tp + tprel = (S + A)`

tlsgd — Allocates two contiguous entries in the GOT to hold a tls_index structure, with values dtpmod and dtprel, and computes the offset from .TOC of the first entry.

If n is the offset computed:

```
GOT[n] = dtpmod

GOT[n + 1] = dtprel
```

The call to __tls_get_addr () happens as:

```
__tls_get_addr ((tls_index *) &GOT[n])
```

tlsld — Allocates two contiguous entries in the GOT to hold a tls_index structure, with values dtpmod and zero, and computes the offset from .TOC of the first entry.

If n is the offset computed:

```
GOT[n] = dtpmod

GOT[n + 1] = 0
```

The call to __tls_get_addr () happens as:

```
__tls_get_addr ((tls_index *) &GOT[n])
```

tprelg — Allocates an entry in the GOT with value tprel, and computes the offset from .TOC of the entry.

If n is the offset computed:

```
GOT[n] = tprel
```

The value of tprel is loaded into a register from the location (`GOT` + n) to be used in an r2 form instruction.

Note: Relocations not using the #ha(), #hi(), and #lo() modifiers (those flagged with an asterisk(*) will trigger a relocation failure if the value computed does not fit in the field specified.

3.5.3 Relocation Types Table

The following rules apply to the relocation types defined in *Table 3-2* on page 94:

- For relocation types in which the names contain 14 or 16, the upper 49 bits of the value computed before shifting must both be the same. For relocation types in which the names contain 24, the upper 39 bits of the value computed before shifting must all be the same. For relocation types in which the names contain 14 or 24, the low 2 bits of the value computed before shifting must all be zero.
- The relocation types whose Field column entry contains an asterisk (*) are subject to failure if the value computed does not fit in the allocated bits.
- Relocations that refer to half16ds (56 - 66, 87 - 88, 91 - 92, 95 - 96, and 101 - 102) are to be used to direct the linker to look at the underlying instruction and treat the field as a DS or DQ field. ABI-compliant tools should give an error for attempts to relocate an address to a value that is not divisible by 4.

Table 3-2. Relocation Table (Page 1 of 4)

Relocation Name	Value	Field	Expression
R_PPC64_NONE	0	none	none
R_PPC64_ADDR32	1	word32*	S + A
R_PPC64_ADDR24	2	low24*	(S + A) >> 2
R_PPC64_ADDR16	3	half16*	S + A
R_PPC64_ADDR16_LO	4	half16	#lo(S + A)
R_PPC64_ADDR16_HI	5	half16*	#hi(S + A)
R_PPC64_ADDR16_HA	6	half16*	#ha(S + A)
R_PPC64_ADDR14	7	low14*	(S + A) >> 2
R_PPC64_REL24	10	low24*	(S + A - P) >> 2
R_PPC64_REL14	11	low14*	(S + A - P) >> 2
R_PPC64_GOT16	14	half16*	G
R_PPC64_GOT16_LO	15	half16	#lo(G)
R_PPC64_GOT16_HI	16	half16*	#hi(G)
R_PPC64_GOT16_HA	17	half16*	#ha(G)
R_PPC64_COPY	19	varies	See *Section 3.5.4 Relocation Descriptions* on page 97.
R_PPC64_GLOB_DAT	20	doubleword64	S + A
R_PPC64_JMP_SLOT	21	doubleword64	See *Section 3.5.4 Relocation Descriptions* on page 97.
R_PPC64_RELATIVE	22	doubleword64	B + A
R_PPC64_UADDR32	24	word32*	S + A
R_PPC64_UADDR16	25	half16*	S + A
R_PPC64_REL32	26	word32*	S + A - P
R_PPC64_PLT32	27	word32*	L
R_PPC64_PLTREL32	28	word32*	L - P
R_PPC64_PLT16_LO	29	half16	#lo(L)
R_PPC64_PLT16_HI	30	half16*	#hi(L)
R_PPC64_PLT16_HA	31	half16*	#ha(L)
R_PPC64_SECTOFF	33	half16*	R + A
R_PPC64_SECTOFF_LO	34	half16	#lo(R + A)
R_PPC64_SECTOFF_HI	35	half16*	#hi(R + A)
R_PPC64_SECTOFF_HA	36	half16*	#ha(R + A)
R_PPC64_ADDR30	37	word30	(S + A - P) >> 2
R_PPC64_ADDR64	38	doubleword64	S + A
R_PPC64_ADDR16_HIGHER	39	half16	#higher(S + A)

Note: Relocation values 8, 9, 12, 13, 18, 23, 32, and 247 are not used. This is to maintain a correspondence to the relocation values used by the 32-bit PowerPC ELF ABI.

Table 3-2. Relocation Table (Page 2 of 4)

Relocation Name	Value	Field	Expression
R_PPC64_ADDR16_HIGHERA	40	half16	#highera(S + A)
R_PPC64_ADDR16_HIGHEST	41	half16	#highest(S + A)
R_PPC64_ADDR16_HIGHESTA	42	half16	#highesta(S + A)
R_PPC64_UADDR64	43	doubleword64	S + A
R_PPC64_REL64	44	doubleword64	S + A - P
R_PPC64_PLT64	45	doubleword64	L
R_PPC64_PLTREL64	46	doubleword64	L - P
R_PPC64_TOC16	47	half16*	S + A - .TOC.
R_PPC64_TOC16_LO	48	half16	#lo(S + A - .TOC.)
R_PPC64_TOC16_HI	49	half16*	#hi(S + A - .TOC.)
R_PPC64_TOC16_HA	50	half16*	#ha(S + A - .TOC.)
R_PPC64_TOC	51	doubleword64	.TOC.
R_PPC64_PLTGOT16	52	half16*	M
R_PPC64_PLTGOT16_LO	53	half16	#lo(M)
R_PPC64_PLTGOT16_HI	54	half16*	#hi(M)
R_PPC64_PLTGOT16_HA	55	half16*	#ha(M)
R_PPC64_ADDR16_DS	56	half16ds*	(S + A) >> 2
R_PPC64_ADDR16_LO_DS	57	half16ds	#lo(S + A) >> 2
R_PPC64_GOT16_DS	58	half16ds*	G >> 2
R_PPC64_GOT16_LO_DS	59	half16ds	#lo(G) >> 2
R_PPC64_PLT16_LO_DS	60	half16ds	#lo(L) >> 2
R_PPC64_SECTOFF_DS	61	half16ds*	(R + A) >> 2
R_PPC64_SECTOFF_LO_DS	62	half16ds	#lo(R + A) >> 2
R_PPC64_TOC16_DS	63	half16ds*	(S + A - .TOC.) >> 2
R_PPC64_TOC16_LO_DS	64	half16ds	#lo(S + A - .TOC.) >> 2
R_PPC64_PLTGOT16_DS	65	half16ds*	M >> 2
R_PPC64_PLTGOT16_LO_DS	66	half16ds	#lo(M) >> 2
R_PPC64_TLS	67	none	none
R_PPC64_DTPMOD64	68	doubleword64	@dtpmod
R_PPC64_TPREL16	69	half16*	@tprel
R_PPC64_TPREL16_LO	70	half16	#lo(@tprel)
R_PPC64_TPREL16_HI	71	half16*	#hi(@tprel)
R_PPC64_TPREL16_HA	72	half16*	#ha(@tprel)
R_PPC64_TPREL64	73	doubleword64	@tprel
R_PPC64_DTPREL16	74	half16*	@dtprel

Note: Relocation values 8, 9, 12, 13, 18, 23, 32, and 247 are not used. This is to maintain a correspondence to the relocation values used by the 32-bit PowerPC ELF ABI.

Table 3-2. Relocation Table (Page 3 of 4)

Relocation Name	Value	Field	Expression
R_PPC64_DTPREL16_LO	75	half16	#lo(@dtprel)
R_PPC64_DTPREL16_HI	76	half16*	#hi(@dtprel)
R_PPC64_DTPREL16_HA	77	half16*	#ha(@dtprel)
R_PPC64_DTPREL64	78	doubleword64	@dtprel
R_PPC64_GOT_TLSGD16	79	half16*	@got@tlsgd
R_PPC64_GOT_TLSGD16_LO	80	half16	#lo(@got@tlsgd)
R_PPC64_GOT_TLSGD16_HI	81	half16*	#hi(@got@tlsgd)
R_PPC64_GOT_TLSGD16_HA	82	half16*	#ha(@got@tlsgd)
R_PPC64_GOT_TLSLD16	83	half16*	@got@tlsld
R_PPC64_GOT_TLSLD16_LO	84	half16	#lo(@got@tlsld)
R_PPC64_GOT_TLSLD16_HI	85	half16*	#hi(@got@tlsld)
R_PPC64_GOT_TLSLD16_HA	86	half16*	#ha(@got@tlsld)
R_PPC64_GOT_TPREL16_DS	87	half16ds*	@got@tprel
R_PPC64_GOT_TPREL16_LO_DS	88	half16ds	#lo(@got@tprel)
R_PPC64_GOT_TPREL16_HI	89	half16*	#hi(@got@tprel)
R_PPC64_GOT_TPREL16_HA	90	half16*	#ha(@got@tprel)
R_PPC64_GOT_DTPREL16_DS	91	half16ds*	@got@dtprel
R_PPC64_GOT_DTPREL16_LO_DS	92	half16ds	#lo(@got@dtprel)
R_PPC64_GOT_DTPREL16_HI	93	half16*	#hi(@got@dtprel)
R_PPC64_GOT_DTPREL16_HA	94	half16*	#ha(@got@dtprel)
R_PPC64_TPREL16_DS	95	half16ds*	@tprel
R_PPC64_TPREL16_LO_DS	96	half16ds	#lo(@tprel)
R_PPC64_TPREL16_HIGHER	97	half16	#higher(@tprel)
R_PPC64_TPREL16_HIGHERA	98	half16	#highera(@tprel)
R_PPC64_TPREL16_HIGHEST	99	half16	#highest(@tprel)
R_PPC64_TPREL16_HIGHESTA	100	half16	#highesta(@tprel)
R_PPC64_DTPREL16_DS	101	half16ds*	@dtprel
R_PPC64_DTPREL16_LO_DS	102	half16ds	#lo(@dtprel)
R_PPC64_DTPREL16_HIGHER	103	half16	#higher(@dtprel)
R_PPC64_DTPREL16_HIGHERA	104	half16	#highera(@dtprel)
R_PPC64_DTPREL16_HIGHEST	105	half16	#highest(@dtprel)
R_PPC64_DTPREL16_HIGHESTA	106	half16	#highesta(@dtprel)
R_PPC64_TLSGD	107	none	none
R_PPC64_TLSLD	108	none	none
R_PPC64_TOCSAVE	109	none	none

Note: Relocation values 8, 9, 12, 13, 18, 23, 32, and 247 are not used. This is to maintain a correspondence to the relocation values used by the 32-bit PowerPC ELF ABI.

Table 3-2. Relocation Table (Page 4 of 4)

Relocation Name	Value	Field	Expression
R_PPC64_ADDR16_HIGH	110	half16	#high(S + A)
R_PPC64_ADDR16_HIGHA	111	half16	#higha(S + A)
R_PPC64_TPREL16_HIGH	112	half16	#high(@tprel)
R_PPC64_TPREL16_HIGHA	113	half16	#higha(@tprel)
R_PPC64_DTPREL16_HIGH	114	half16	#high(@dtprel)
R_PPC64_DTPREL16_HIGHA	115	half16	#higha(@dtprel)
R_PPC64_REL24_NOTOC	116	low24*	(S + A - P) >> 2
R_PPC64_ADDR64_LOCAL	117	doubleword64	S + A (See *Section 3.5.4 Relocation Descriptions* on page 97.)
R_PPC64_IRELATIVE	248	doubleword64	See *Section 3.5.4 Relocation Descriptions* on page 97.
R_PPC64_REL16	249	half16*	S + A - P
R_PPC64_REL16_LO	250	half16	#lo(S + A - P)
R_PPC64_REL16_HI	251	half16*	#hi(S + A - P)
R_PPC64_REL16_HA	252	half16*	#ha(S + A - P)
R_PPC64_GNU_VTINHERIT	253		
R_PPC64_GNU_VTENTRY	254		

Note: Relocation values 8, 9, 12, 13, 18, 23, 32, and 247 are not used. This is to maintain a correspondence to the relocation values used by the 32-bit PowerPC ELF ABI.

3.5.4 Relocation Descriptions

The following list describes relocations that can require special handling or description.

R_PPC64_GOT16*

These relocation types are similar to the corresponding R_PPC64_ADDR16* types. However, they refer to the address of the symbol's GOT entry and instruct the link editor to build a GOT.

R_PPC64_PLTGOT16*

These relocation types are similar to the corresponding R_PPC64_GOT16* types. However, if the link editor *cannot* determine the actual value of the symbol, the GOT entry may contain the address of an entry in the procedure linkage table. The link editor creates that entry in the procedure linkage table and stores that address in the GOT entry. This permits lazy resolution of function symbols at run time. If the link editor *can* determine the value of the symbol, it stores that value in the corresponding GOT entry. The link editor may generate an R_PPC64_GLOB_DAT relocation as usual.

R_PPC64_PLTREL32, R_PPC64_PLTREL64

These relocations indicate that reference to a symbol should be resolved through a call to the symbol's procedure linkage table entry. Additionally, it instructs the link editor to build a procedure linkage table for the executable or shared object if one is not created.

R_PPC64_COPY

This relocation type is created by the link editor for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current relocatable file and in a shared object file. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

R_PPC64_GLOB_DAT

This relocation type allows determination of the correspondence between symbols and GOT entries. It is similar to R_PPC64_ADDR64. However, it sets a GOT entry to the address of the specified symbol.

R_PPC64_JMP_SLOT

This relocation type is created by the link editor for dynamic linking. Its offset member gives the location of a procedure linkage table (PLT) entry. The dynamic linker modifies the PLT entry to transfer control to the designated symbol's address (see *Section 4.2.5 Procedure Linkage Table* on page 127).

R_PPC64_RELATIVE

This relocation type is created by the link editor for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The corresponding virtual address is computed by the dynamic linker. It adds the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_PPC64_IRELATIVE

The link editor creates this relocation type for dynamic linking. Its addend member specifies the global entry-point location of a resolver function returning a function pointer. It is used to implement the STT_GNU_IFUNC framework. The resolver is called, and the returned pointer copied into the location specified by the relocation offset member.

R_PPC64_TLS, R_PPC64_TLSGD, R_PPC64_TLSLD

Used as markers on thread local storage (TLS) code sequences, these relocations tie the entire sequence with a particular TLS symbol. For more information, see *Section 3.7 Thread Local Storage ABI* on page 102.

R_PPC64_TOCSAVE

This relocation type indicates a position where a TOC save may be inserted in the function to avoid a TOC save as part of the PLT stub code. A nop can be emitted by a compiler in a function's prologue code. A link editor can change it to a TOC pointer save instruction. This marker relocation is placed on the prologue nop and on nops after bl instructions, with the symbol plus addend pointing to the prologue nop. If the link editor uses the prologue to save r2, it may omit r2 saves in the PLT call stub code emitted for calls marked by R_PPC64_TOCSAVE.

R_PPC64_UADDR*

These relocation types are the same as the corresponding R_PPC64_ADDR* types, except that the datum to be relocated is allowed to be unaligned.

R_PPC64_ADDR64_LOCAL

When a separate local entry point exists, this relocation type is used to initialize a memory location with the address of that local entry point.

R_PPC64_REL24_NOTOC

This relocation type is used to specify a function call where the TOC pointer is not initialized. It is similar to R_PPC64_REL24 in that it specifies a symbol to be resolved. However, if the symbol is resolved by inserting a call to a PLT stub code, the PLT stub code must not rely on the presence of a valid TOC base address in TOC register r2 to reference the PLT function table.

3.5.5 Assembler Syntax

The offset from .TOC in the GOT where the value of the symbol is stored is given by the assembly syntax symbol@got. The value of symbol is the address of the variable named symbol.

For example,

```
addis r3, r2,x@got@ha
ld r3,x@got@l(r3)
```

Although the Power ISA only defines 16-bit displacements, many TOCs (and hence a GOT) are larger then 64 KB but fit within 2 GB, which can be addressed with 32-bit offsets from r2. Therefore, this ABI defines a simple syntax for 32-bit offsets to the GOT.

The syntaxes SYMBOL@got@ha, SYMBOL@got@h, and SYMBOL@got@l refer to the high adjusted, high, and low parts of the GOT offset. (For an explanation of the meaning of “high adjusted,” see*Section 3.5 Relocation Types* on page 89). SYMBOL@got@ha corresponds to bits 32 - 63 of the offset within the global offset table with adjustment for the sign extension of the low-order offset bits. SYMBOL@got@l corresponds to the 16 low-order bits of the offset within the global offset table.

The syntax SYMBOL@toc refers to the value (SYMBOL - .TOC.), where .TOC. represents the TOC base for the current object file. This provides the address of the variable whose name is SYMBOL as an offset from the TOC base.

As with the GOT, the syntaxes SYMBOL@toc@ha, SYMBOL@toc@h, and SYMBOL@toc@l refer to the high adjusted, high, and low parts of the TOC offset.

The syntax SYMBOL@got@plt may be used to refer to the offset in the TOC of a procedure linkage table entry stored in the global offset table. The corresponding syntaxes SYMBOL@got@plt@ha, SYMBOL@got@plt@h, and SYMBOL@got@plt@l are also defined.

Note: If X is a variable stored in the TOC, then X@got is the offset within the TOC of a doubleword whose value is X@toc.

The special symbol .TOC. is used to represent the TOC base for the current object file.

The following code might appear in a PIC code setup sequence to compute the distance from a function entry point to the TOC base:

```
addis 2,12,.TOC.-func@ha
addi 2,2,.TOC.-func@l
```

The syntax `SYMBOL@localentry` refers to the value of the local entry point associated with a function symbol. It can be used to initialize a memory word with the address of the local entry point as follows:

```
.quad func@localentry
```

3.6 Assembler- and Linker-Mediated Executable Optimization

To optimize object code, the assembler and linker may rewrite object code to implement the function call and return conventions and access to global and thread-local data. It is the responsibility of compilers and programmers to generate assembly programs and objects that conform to the requirements as indicated in this section.

3.6.1 Function Call

The static linker must modify a nop instruction after a bl function call to restore the TOC pointer in r2 from 24(r1) when an external symbol that may use the TOC may be called, as in *Section 2.3.6 Function Calls* on page 72. Object files must contain a nop slot after a bl instruction to an external symbol.

3.6.2 Reference Optimization

References to the GOT may be optimized by rewriting indirect reference code to replace the reference by an address computation. This transformation is only performed by the linker when the symbol is known to be local to the module.

3.6.3 Displacement Optimization for TOC Pointer Relative Accesses

Assemblers and linkers *may* optimize TOC reference code that consists of two instructions with equivalent code when offset@ha is 0.

TOC reference code:

```
addis rt, r2, offset@ha
lwz rt, offset@l(rt)
```

Equivalent code:

```
NOP
lwz rt, offset(r2)
```

Compilers and programmers *must* ensure that r2 is live at the actual data access point associated with extended displacement addressing.

3.6.3.1 TOC Pointer Usage

To enable linker-based optimizations when global data is accessed, the TOC pointer needs to be available for dereference at the point of all uses of values derived from the TOC pointer in conjunction with the @l operator. This property is used by the linker to optimize TOC pointer accesses. In addition, all reaching definitions for a TOC-pointer-derived access must compute the same definition.

In some implementations, non-ABI-compliant code may be processed by providing additional linker options; for example, linker options disabling linker optimization. However, this behavior in support of non-ABI-compliant code is not guaranteed to be portable and supported in all systems.

Compliant example

```
        addis r4, r2, mysym@toc@ha
        b target


        ...


        addis   r4, r2, mysym@toc@ha
target:
        addi    r4, r4, mysym@toc@l
        ...
```

Non-compliant example

```
        li  r4, 0  ; #d1
        b target

        ...

        addis   r4, r2, mysym@toc@ha ; #d2
target:
        addi    r4, r4, mysym@toc@l ; incompatible definitions #d1 and #d2 reach this
        ...
```

3.6.3.2 Table Jump Sequences

Some linkers may rewrite jump table sequences, as described in *Section 2.3.7 Branching* on page 75. For example, linkers may rewrite address references created using GOT-indirect loads and bl+4 sequences to use TOC-relative address computation.

3.6.4 Fusion

Code generation in compilers, linkers, and by programmers shall use a destructive sequence of two sequential instructions consisting of first an addis followed by a second instruction using a D form instruction to create or load from a 32-bit offset from a register to enable hardware fusion whenever possible:

```
        addis      r4, r3, upper
        <lbz,lhz,lwz,ld> r4, lower(r4)

        addis      r4, r3, upper
        addi       r4, r4, lower
```

It is encouraged that assemblers provide pseudo-ops to facilitate such code generation with a single assembler mnemonic.

3.6.5 Thread-Local Linker Optimizations

Additional code rewriting is performed by the linker in conjunction with the use of thread-local storage described in *Section 3.7.4 TLS Link Editor Optimizations* on page 108.

3.7 Thread Local Storage ABI

The *ELF Handling for Thread-Local Storage* document is the authoritative TLS ABI specification that defines the context in which information in the TLS section of this Power Architecture 64-bit ELF V2 ABI must be viewed. For information about how to access this document, see *Section 1.1 Reference Documentation* on page 17. To maintain congruence with that document, in this section the term module refers to an executable or shared object since both are treated similarly.

3.7.1 TLS Background

Most C/C++ implementations support (as an extension to earlier versions of the language) the keyword __thread to be used as a storage-class specifier in variable declarations and definitions of data objects with thread storage duration. (The 2011 ISO C Standard uses _Thread_local as the keyword, while the 2011 ISO C++ Standard uses thread_local.) A variable declared in this manner is automatically allocated local to each thread. Its lifetime is defined to be the entire execution of the thread. Any initialization value is assigned once before thread startup.

3.7.2 TLS Runtime Handling

A thread-local variable is completely identified by the module in which it is defined, along with the offset of the variable relative to the start of the TLS block for the module. A module is referenced by its index (an integer starting with 1, which is assigned by the run-time environment) into the dynamic thread vector (DTV). The offset of the variable is kept in the st_value field of the TLS variable's symbol table entry.

The TLS data structures follow variant I of the ELF TLS ABI. For the 64-bit PowerPC Architecture, the specific organization of the data structures is as follows.

The thread control block (TCB) consists of the DTV, which is an 8-byte pointer. An extended TCB may have additional implementation-specific fields; these fields are located *before* the DTV pointer because the addresses are computed as negative offsets from the TCB address. The fields must never be rearranged for any reason.

The current glibc extended TCB is:

```
typedef struct
{

  /* Reservation for dynamic system optimizer ABI.  */
  uintptr_t dso_slot2;
  uintptr_t dso_slot1;

  /* Reservation for tar register (ISA 2.07).  */
  uintptr_t tar_save;
```

```
  /* GCC split stack support.  */
  void *__private_ss;

  /* Reservation for the event-based branching ABI.  */
  uintptr_t ebb_handler;
  uintptr_t ebb_ctx_pointer;
  uintptr_t ebb_reserved1;
  uintptr_t ebb_reserved2;
  uintptr_t pointer_guard;

/* Reservation for stack guard */
  uintptr_t stack_guard;

/* DTV pointer */
  dtv_t *dtv;
} tcbhead_t;
```

Modules that will not be unloaded will be present at startup time; the TLS blocks for these are created consecutively and immediately follow the TCB. The offset of the TLS block of an initially available module from the TCB remains fixed after program start.

The tlsoffset(m) values for a module with index m, where m ranges 1 - M, M being the total number of modules, are computed as follows:

```
tlsoffset(1) = round(16, align(1))
tlsoffset(m + 1) = round(tlsoffset(m) + tlssize(m), align(m + 1))
```

- The function round() returns its first argument rounded up to the next multiple of its second argument:

  ```
  round(x, y) = y × ceiling(x / y)
  ```

- The function ceiling() returns the smallest integer greater than or equal to its argument, where n is an integer satisfying: $n - 1 < x \leq n$:

  ```
  ceiling(x) = n
  ```

In the case of dynamic shared objects (DSO), TLS blocks are allocated on an as-needed basis, with the details of allocation abstracted away by the __tls_get_addr() function, which is used to retrieve the address of any TLS variable.

The prototype for the __tls_get_addr() function, is defined as follows.

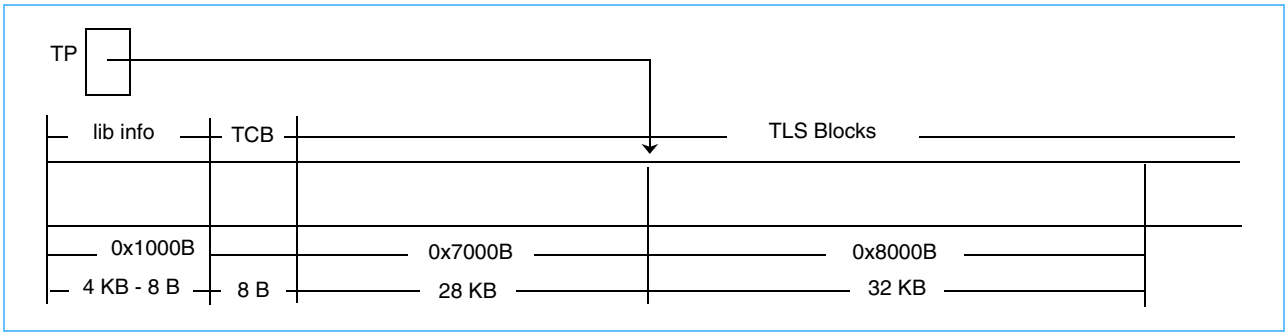
```
typedef struct
{
  unsigned long int ti_module;
  unsigned long int ti_offset;
} tls_index;

extern void *__tls_get_addr (tls_index *ti);
```

The thread pointer (TP) is held in r13 and is used to access the TCB. The TP is initialized to point 0x7000 bytes past the end of the TCB. The TP offset allows for efficient addressing of the TCB and up to 4 KB - 8 B of other thread library information (placed before the TCB).

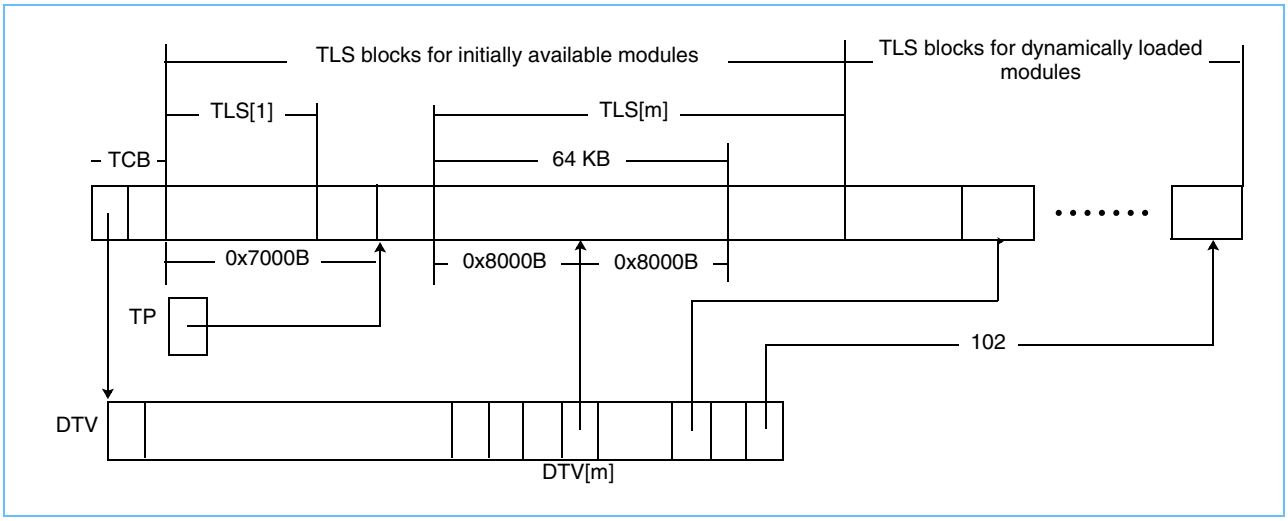
Figure 3-1 shows the region of memory before and after the TCB that can be efficiently addressed by the TP.

Figure 3-1. Thread Pointer Addressable Memory



Each DTV pointer points 0x8000 bytes past the start of each TLS block. (For implementation reasons, the actual value stored in the DTV may point to the start of a TLS block. However, values returned by accessor functions will be offset by 0x8000 bytes.) This offset allows the first 64 KB of each block to be addressed from a DTV pointer using fewer machine instructions.

Figure 3-2. TLS Block Diagram



TLS[m] denotes the TLS block for the module with index m. DTV[m] denotes the DTV pointer for the module with index m.

3.7.3 TLS Access Models

TLS data access is categorized into the following models:

- General Dynamic TLS Model
- Local Dynamic TLS Model
- Initial Exec TLS Model
- Local Exec TLS Model

Examples for each access model are provided in the following TLS Model subsections.

3.7.3.1 General Dynamic TLS Model

Note: This specification provides examples based on the medium code model, which is the default for the ELF V2 ABI.

Given the following code fragment, to determine the address of a thread-local variable x, the __tls_get_addr() function is called with one parameter. That parameter is a pointer to a data object of type tls_index.

```
extern __thread unsigned int x;
&x;
```

Table 3-3. General Dynamic Initial Relocations

Code Sequence	Relocation	Symbol
addis r3, r2, x@got@tlsgd@ha	R_PPC64_GOT_TLSGD16_HA	x
addi r3, r3, x@got@tlsgd@l	R_PPC64_GOT_TLSGD16_LO	x
bl __tls_get_addr(x@tlsgd)	R_PPC64_TLSGD	x
	R_PPC64_REL24	__tls_get_addr
nop		

Table 3-4. General Dynamic GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_DTPMOD64	x
GOT[n+1]	R_PPC64_DTPREL64	x

The relocation specifier @got@tlsgd causes the link editor to create a data object of type tls_index in the GOT. The address of this data object is loaded into the first argument register with the addis and addi instruction, and a standard function call is made. Notice that the bl instruction has two relocations: the R_PPC64_TLSGD tying it to the argument setup instructions and the R_PPC64_REL24 specifying the call destination.

3.7.3.2 Local Dynamic TLS Model

For the Local Dynamic TLS Model, three different relocation sequences may be used, depending on the size of the thread storage block offset to the variable. For the following code sequence, a different relocation sequence is used for each variable.

```
static __thread unsigned int x1;
static __thread unsigned int x2;
static __thread unsigned int x3;
&x1;
&x2;
&x3;
```

Table 3-5. Local Dynamic Initial Relocations

Code Sequence	Relocation	Symbol
addis r3, r2, x1@got@tlsld@ha	R_PPC64_GOT_TLSLD16_HA	x1
addi r3, r3, x1@got@tlsld@l	R_PPC64_GOT_TLSLD16_LO	x1
bl __tls_get_addr(x1@tlsld)	R_PPC64_TLSLD	x1
	R_PPC64_REL24	__tls_get_addr
nop		
...		
addi r9, r3, x1@dtprel	R_PPC64_DTPREL16	x1
...		
addis r9, r3, x2@dtprel@ha	R_PPC64_DTPREL16_HA	x2
addi r9, r9, x2@dtprel@l	R_PPC64_DTPREL16_LO	x2
...		
addis r9, r2, x3@got@dtprel@ha	R_PPC64_GOT_DTPREL16_HA	x3
ld r9, x3@got@dtprel@l(r9)	R_PPC64_GOT_DTPREL16_LO_DS	x3
add r9, r9, r3		

Table 3-6. Local Dynamic GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_DTPMOD64	x1
GOT[n+1]	0	
GOT[m]	R_PPC64_DTPREL64	x3

The relocation specifier @got@tlsld in the first instruction causes the link editor to generate a tls_index data object in the GOT with a fixed 0 offset. The following code assumes that x1 is in the first 64 KB of the thread storage block. The x2 symbol is not within the first 64 KB but is within the first 2 GB, and x3 is outside the 2 GB area. To load the values of x1, x2, and x3 instead of their addresses, replace the latter part of *Table 3-5* with the following code sequence.

Table 3-7. Local Dynamic Relocations with Values Loaded

Code Sequence	Relocation	Symbol
...		
lwz r0, x1@dtprel(r3)	R_PPC64_DTPREL16	x1
...		
addis r9, r3, x2@dtprel@ha	R_PPC64_DTPREL16_HA	x2
lwz r0, x2@dtprel@l(r9)	R_PPC64_DTPREL16_LO	x2
...		

Table 3-7. Local Dynamic Relocations with Values Loaded

Code Sequence	Relocation	Symbol
addis r9, r2, x3@got@dtprel@ha	R_PPC64_GOT_DTPREL16_HA	x3
ld r9, x3@got@dtprel@l(r9)	R_PPC64_GOT_DTPREL16_LO_DS	x3
lwzx r0, r3, r9		

3.7.3.3 Initial Exec TLS Model

Given the following code fragment, the relocation sequence in *Table 3-8* is used for the Initial Exec TLS Model:

```
extern __thread unsigned int x;
&x;
```

Table 3-8. Initial Exec Initial Relocations

Code Sequence	Relocation	Symbol
addis r9, r2, x@got@tprel@ha	R_PPC64_GOT_TPREL16_HA	x
ld r9, x@got@tprel@l(r9)	R_PPC64_GOT_TPREL16_LO_DS	x
add r9, r9, x@tls	R_PPC64_TLS	x

Table 3-9. Initial Exec GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_TPREL64	x

The relocation specifier @got@tprel in the first instruction causes the link editor to generate a GOT entry with a relocation that the dynamic linker will replace with the offset for x relative to the thread pointer. The relocation specifier x@tls tells the assembler to use an r13 form of the instruction. That is, add r9,r9,r13 in this case, and tag the instruction with a relocation that indicates it belongs to a TLS sequence. This relocation specifier can be used later by the link editor when optimizing TLS code.

To read the contents of the variable instead of calculating its address, the add r9, r9, x@tls instruction might be replaced with lwzx r0, r9, x@tls.

3.7.3.4 Local Exec TLS Model

Given the following code fragment, three different relocation sequences may be used, depending on the size of the offset to the variable. The sequence in *Table 3-10* handles offsets within 60 KB relative to the end of the TCB (where r13 points 28 KB past the end of the TCB, which is immediately before the first TLS block). The sequence in *Table 3-11* handles offsets past 60 KB and less than 2 GB + 28 KB relative to the end of the TCB. The third sequence is identical to the Initial Exec sequence shown in *Table 3-8*.

```
static __thread unsigned int x;
&x;
```

Figure 3-3 on page 108 illustrates which sequence is used.

Figure 3-3. Local Exec TLS Model Sequences

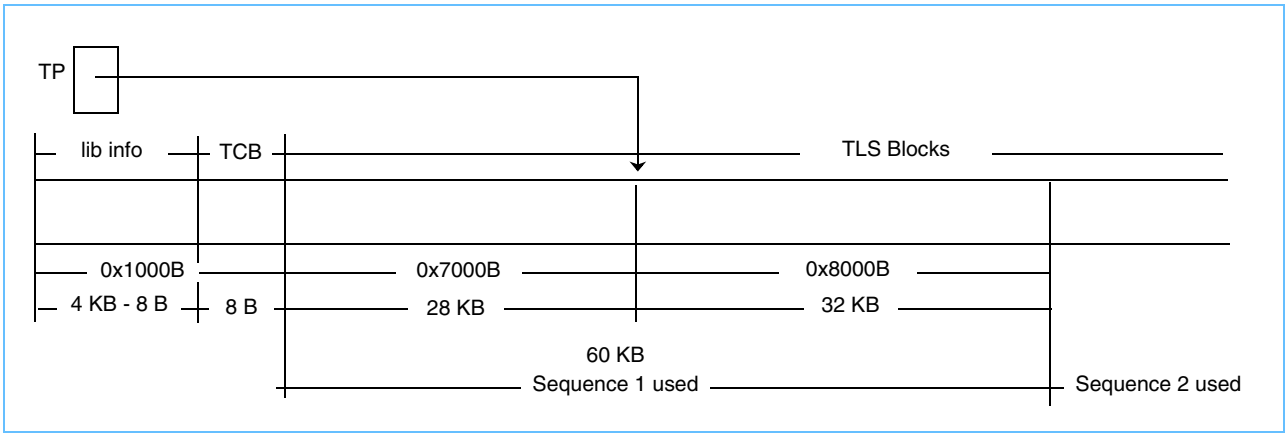


Table 3-10. Local Exec Initial Relocations (Sequence 1)

Code Sequence	Relocation	Symbol
addi r9, r13, x1@tprel	R_PPC_TPREL16	x

Table 3-11. Local Exec Initial Relocations (Sequence 2)

Code Sequence	Relocation	Symbol
addis r9, r13, x2@tprel@ha	R_PPC64_TPREL16_HA	x
addi r9, r9, x2@tprel@l	R_PPC64_TPREL16_LO	x

3.7.4 TLS Link Editor Optimizations

In some cases, the link editor may be able to optimize TLS code sequences, provided the compiler emits code sequences as described.

The following TLS link editor transformations are provided as optimizations to convert between specific TLS access models:

- General Dynamic to Initial Exec
- General Dynamic to Local Exec
- Local Dynamic to Local Exec
- Initial Exec to Local Exec

3.7.4.1 General Dynamic to Initial Exec

Table 3-12. General-Dynamic-to-Initial-Exec Initial Relocations

Code Sequence	Relocation	Symbol
addis r3, r2, x@got@tlsgd@ha	R_PPC64_GOT_TLSGD16_HA	x
addi r3, r3, x@got@tlsgd@l	R_PPC64_GOT_TLSGD16_LO	x
bl __tls_get_addr(x@tlsgd)	R_PPC64_TLSGD	x
	R_PPC64_REL24	__tls_get_addr
nop		

Table 3-13. General-Dynamic-to-Initial-Exec GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_DTPMOD64	x
GOT[n+1]	R_PPC64_DTPREL64	x

The preceding code and global offset table entries are replaced by the following code and global offset table entries.

Table 3-14. General-Dynamic-to-Initial-Exec Replacement Initial Relocations

Code Sequence	Relocation	Symbol
addis r3, r2, x@got@tprel@ha	R_PPC64_GOT_TPREL16_HA	x
ld r3, x@got@tprel@l(r3)	R_PPC64_GOT_TPREL16_LO_DS	x
nop		
add r3, r3, r13		

Table 3-15. General-Dynamic-to-Initial-Exec Replacement GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_TPREL64	x

3.7.4.2 General Dynamic to Local Exec

Table 3-16. General-Dynamic-to-Local-Exec Initial Relocations

Code Sequence	Relocation	Symbol
addis r3, r2, x@got@tlsgd@ha	R_PPC64_GOT_TLSGD16_HA	x
addi r3, r3, x@got@tlsgd@l	R_PPC64_GOT_TLSGD16_LO	x
bl __tls_get_addr(x@tlsgd)	R_PPC64_TLSGD	x
	R_PPC64_REL24	__tls_get_addr
nop		

Table 3-17. General-Dynamic-to-Local-Exec GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_DTPMOD64	x
GOT[n+1]	R_PPC64_DTPREL64	x

The preceding code and global offset table entries are replaced by the following code, which makes no reference to GOT entries. The GOT entries in *Table 3-17* can be removed from the GOT by the linker when performing this code transformation.[1]

Table 3-18. General-Dynamic-to-Local-Exec Replacement Initial Relocations

Code Sequence	Relocation	Symbol
nop		
addis r3, r13, x@tprel@ha	R_PPC64_TPREL16_HA	x
nop		
addi r3, r3, x@tprel@l	R_PPC64_TPREL16_LO	x

3.7.4.3 Local Dynamic to Local Exec

Under this TLS linker optimization, the function call is replaced with an equivalent code sequence. However, as shown in the following code examples, the dtprel sequences are left unchanged.

Table 3-19. Local-Dynamic-to-Local-Exec Initial Relocations (Page 1 of 2)

Code Sequence	Relocation	Symbol
addis r3, r2, x1@got@tlsld@ha	R_PPC64_GOT_TLSLD16_HA	x1
addi r3, r3, x1@got@tlsld@l	R_PPC64_GOT_TLSLD16_LO	x1
bl __tls_get_addr(x1@tlsld)	R_PPC64_TLSLD	x1
	R_PPC64_REL24	__tls_get_addr
nop		
...		
addi r9, r3, x1@dtprel	R_PPC64_DTPREL16	x1
...		
addis r9, r3, x2@dtprel@ha	R_PPC64_DTPREL16_HA	x2
addi r9, r9, x2@dtprel@l	R_PPC64_DTPREL16_LO	x2
...		

1. To further optimize the code in *Table 3-17*, a linker may reschedule the sequence to exploit fusion by generating a sequence that may be fused by Power processors:

```
nop
addis r3, r13, x@tprel@ha
addi r3, r3, x@tprel@l
nop
```

Table 3-19. Local-Dynamic-to-Local-Exec Initial Relocations (Page 2 of 2)

Code Sequence	Relocation	Symbol
addis r9, r2, x3@got@dtprel@ha	R_PPC64_GOT_DTPREL16_HA	x3
ld r9, x3@got@dtprel@l(r9)	R_PPC64_GOT_DTPREL16_LO_DS	x3
add r9, r9, r3		

Table 3-20. Local-Dynamic-to-Local-Exec GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_DTPMOD64	x1
GOT[n+1]		
...		
GOT[m]	R_PPC64_DTPREL64	x3

The preceding code and global offset table entries are replaced by the following code and global offset table entries.

Table 3-21. Local-Dynamic-to-Local-Exec Replacement Initial Relocations

Code Sequence	Relocation	Symbol
nop		
addis r3, r13, L@tprel@ha	R_PPC64_TPREL16_HA	link editor generated local symbol
nop		
addi r3, r3, L@tprel@l	R_PPC64_TPREL16_LO	link editor generated local symbol[1]
..		
addi r9, r3, x1@dtprel	R_PPC64_DTPREL16	x1
..		
addis r9, r3, x2@dtprel@ha	R_PPC64_DTPREL16_HA	x2
addi r9, r9, x2@dtprel@l	R_PPC64_DTPREL16_LO	x2
...		
addis r9, r2, x3@got@dtprel@ha	R_PPC64_GOT_DTPREL16_HA	x3
ld r9, x3@got@dtprel@l(r9)	R_PPC64_GOT_DTPREL16_LO_DS	x3
add r9, r9, r3		

1. The linker may prefer to schedule the addis and addi to be adjacent to take advantage of fusion as a microarchitecture optimization opportunity.

The GOT[n] and GOT[n+1] entries can be removed by the linker after the code transformation as shown in *Table 3-22*.

Table 3-22. Local-Dynamic-to-Local-Exec Replacement GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[m]	R_PPC64_DTPREL64	x3

The local symbol generated by the link editor points to the start of the thread storage block plus 0x7000 bytes. In practice, a section symbol with a suitable offset will be used.

3.7.4.4 Initial Exec to Local Exec

This transformation is only performed by the linker when the symbol is within 2 GB + 28 KB of the thread pointer.

Table 3-23. Initial-Exec-to-Local-Exec Initial Relocations

Code Sequence	Relocation	Symbol
addis r9, r2, x@got@tprel@ha	R_PPC64_GOT_TPREL16_HA	x
ld r9, x@got@tprel@l(r9)	R_PPC64_GOT_TPREL16_LO_DS	x
add r9, r9, x@tls	R_PPC64_TLS	x

Table 3-24. Initial-Exec-to-Local-Exec GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_TPREL64	x

The preceding code and global offset table entries are replaced by the following code and global offset table entries.

Table 3-25. Initial-Exec-to-Local-Exec Replacement Initial Relocations

Code Sequence	Relocation	Symbol
nop		
addis r9, r13, x@tprel@ha	R_PPC64_TPREL16_HA	x
addi r9, r9, x@tprel@l	R_PPC64_TPREL16_LO	x

Other sizes and types of thread-local variables may use any of the X-form indexed load or store instructions.

Table 3-26 shows how to access the contents of a variable using the X-form indexed load and store instructions.

Table 3-26. Initial-Exec-to-Local-Exec X-form Initial Relocations

Code Sequence	Relocation	Symbol
addis r9, r2, x@got@tprel@ha	R_PPC64_GOT_TPREL16_HA	x
ld r9, x@got@tprel@l(r9)	R_PPC64_GOT_TPREL16_LO_DS	x
lbzx r10, r9, x@tls	R_PPC64_TLS	x
addi r10, r10, 1		
stbx r10, r9, x@tls	R_PPC64_TLS	x

Table 3-27. Initial-Exec-to-Local-Exec X-form GOT Entry Relocations

Code Sequence	Relocation	Symbol
GOT[n]	R_PPC64_TPREL64	x

The preceding code and global offset table entries are replaced by the following code and global offset table entries.

Table 3-28. Initial-Exec-to-Local-Exec X-form Replacement Initial Relocations

Code Sequence	Relocation	Symbol
nop		
addis r9, r13, x@tprel@ha	R_PPC64_TPREL16_HA	x
lbz r10, x@tprel@l(r9)	R_PPC64_TPREL16_LO	x
addi r10, r10, 1		
stb r10, x@tprel@l(r9)	R_PPC_64TPREL16_LO	x

3.7.5 ELF TLS Definitions

The result of performing a relocation for a TLS symbol is the module ID and its offset within the TLS block. These are then stored in the GOT. Later, they are obtained by the dynamic linker at run-time and passed to __tls_get_addr(), which returns the address for the variable for the current thread.

For more information, see *Section 3.5 Relocation Types* on page 89. For TLS relocations, see *Table 3-2* on page 94.

TLS Relocation Descriptions

The following marker relocations tie together instructions in TLS code sequences. They allow the link editor to reliably optimize TLS code. R_PPC64_TLSGD and R_PPC64_TLSLD shall be emitted immediately before their associated __tls_get_addr call relocation.

R_PPC64_TLS

R_PPC64_TLSGD

R_PPC64_TLSLD

3.8 System Support Functions and Extensions

3.8.1 Back Chain

Systems must provide a back chain by default, and they must include compilers allocating a back chain and system libraries allocating a back chain. Alternate libraries may be supplied in addition to, and beyond, but never instead of those providing a back chain. Code generating and using a back chain shall be the default for compilers, linkers, and library selection.

3.8.2 Nested Functions

Nested functions that access their ancestors' stack frames are entered with r11 initialized to an environment pointer. The environment pointer is typically a copy of the stack pointer for the most recent instance of the nested function's parent's stack frame. When a function pointer to a nested function referencing its outer context is created, an implementation may create a trampoline to load the present environment pointer to r11, followed by an unconditional branch to the function code of the nested function contained in the text segment.

When a trampoline is used, a pointer to a nested function is represented by the code address of the trampoline.

In some environments, the trampoline code may be created by allocating memory on the data stack, making at least pages containing trampolines executable. In other environments, executable pages may be prohibited in the stack area for security reasons.

Alternate implementations, such as creating code stacks for allocating nested function trampolines, may be used. In garbage-collected environments, yet other ways for managing trampolines are available.

3.8.3 Traceback Tables

To support debuggers and exception handlers, the 64-bit *OpenPOWER ELF V2 ABI* defines the use of descriptive debug and unwind information that enables flexible debugging and unwinding of optimized code (such as, for example, DWARF).

To support legacy tooling, the *OpenPOWER ELF V2 ABI* also specifies the use of a traceback table that may provide additional information about functions.

Section 3.8.3.1 Traceback Table Fields on page 114 describes a minimal set of fields that may, optionally, specify information about a function. Additional fields may be present in a traceback table in accordance with commonly used PowerPC traceback conventions in other environments, but they are not specified in the current ABI definition.

3.8.3.1 Traceback Table Fields

The mandatory fields of a traceback table follow:

version — Eight-bit field. This defines the type code for the table. The only currently defined value is zero.

lang — Eight-bit field. This defines the source language for the compiler that generated the code to which this traceback table applies. The default values are as follows:

C	0
FORTRAN	1
Pascal	2
Ada	3
PL/1	4
Basic	5
LISP	6
COBOL	7
Modula2	8
C++	9
RPG	10
PL.8, PLIX	11
Assembly	12
Java	13
Objective C	14

The codes 0xf - 0xfa are reserved. The codes 0xfb - 0xff are reserved for IBM.

globalink — One-bit field. This field is set to 1 if this routine is a special routine used to support the linkage convention: a linkage function including a procedure linkage table function, pointer glue code, a trampoline, or other compiler- or linker-generated functions that stack traceback functions should skip, other than is_eprol functions. For more information, see *Section 2.3.6 Function Calls* on page 72. These routines have an unusual register usage and stack format.

is_eprol — One-bit field. This field is set to 1 if this routine is an out-of-line prologue or epilogue function, including a register save or restore function. Stack traceback functions should skip these. For more information, see *Section 2.3.2 Function Prologue and Epilogue* on page 60. These routines have an unusual register usage and stack format.

has_tboff — One-bit field. This field is set to 1 if the offset of the traceback table from the start of the function is stored in the tb_offset field.

int_proc — One-bit field. This field is set to 1 if this function is a stackless leaf function that does not have a separate stack frame.

has_ctl — One-bit field. This field is set to 1 if ctl_info is provided.

tocless — One-bit field. This field is set to 1 if this function does not have a TOC. For example, a stackless leaf assembly language routine with no references to external objects.

fp_present — One-bit field. This field is set to 1 if the function uses floating-point processor instructions.

log_abort — One-bit field. Reserved.

int_handl — One-bit field. Reserved.

name_present — One-bit field. This field is set to 1 if the name for the procedure is present following the traceback field, as determined by the name_len and name fields.

uses_alloca	One-bit field. This field is set to 1 if the procedure performs dynamic stack allocation. To address their local variables, these procedures require a different register to hold the stack pointer value. This register may be chosen by the compiler, and must be indicated by setting the value of the alloc_reg field.
cl_dis_inv	Three-bit field. Reserved.
saves_cr	One-bit field. This field indicates whether the CR fields are saved in the CR save word. If traceback tables are used in place of DWARF unwind information, at least all volatile CR fields must be saved in the CR save word.
saves_lr	One-bit field. This field is set to 1 if the function saves the LR in the LR save doubleword.
stores_bc	One-bit field. This field is set to 1 if the function saves the back chain (the SP of its caller) in the stack frame header.
fixup	One-bit field. This field is set to 1 if the link editor replaced the original instruction by a branch instruction to a special fix-up instruction sequence.
fp_saved	Six-bit field. This field is set to the number of nonvolatile floating-point registers that the function saves. When traceback unwind and debug information is used, the last register saved is always f31. Therefore, for example, a value of 2 in this field indicates that f30 and f31 are saved.
has_vec_info	One-bit field. This field is set to 1 if the procedure saves nonvolatile vector registers in the vector register save area, specifies the number of vector parameters, or uses VMX instructions.
spare4	One-bit field. Reserved.
gpr_saved	Six-bit field. This field is set to the number of nonvolatile general registers that the function saves. As with fp_saved, when traceback unwind and debug information is used, the last register saved is always r31.
fixedparms	Eight-bit field. This field is set to the number of fixed-point parameters.
floatparms	Seven-bit field. This field is set to the number of floating-point parameters.
parmsonstk	One-bit field. This field is set to 1 if all of the parameters are placed in the parameter save area.

4. Program Loading and Dynamic Linking

4.1 Program Loading

A number of criteria constrain the mapping of an executable file or shared object file to virtual memory segments. During mapping, the operating system may use delayed physical reads to improve performance, which necessitates that file offsets and virtual addresses are congruent, modulo the page size.

Page size must be less than or equal to the operating system implemented congruency. This ABI defines 64 KB congruency as the minimum allowable. To maintain interoperability between operating system implementations, 64 KB congruency is recommended.

Note: There is historical precedence for 64 KB congruency in that there is synergy with the Power Architecture instruction set whereby low and high adjusted relocations can be easily performed using addi or addis instructions.

The value of the p_align member of the program header struct must be 0x10000 or a larger power of 2. If a larger congruency size is used for large pages, p_align should match the congruency value. The size of each segment is defined to be a positive, integral power of two, but no less than 64 KB.

The following program header information illustrates an application that is mapped with a base address of 0x10000000:

Table 4-1. Program Header Example

Header Member	Text Segment	Data Segment
p_type	PT_LOAD	PT_LOAD
p_offset	0x000000	0x000af0
p_vaddr	0x10000000	0x10010af0
p_paddr	0x10000000	0x10010af0
p_filesz	0x00af0	0x00124
p_memsz	0x00af0	0x00128
p_flags	R-E	RW-
p_align	0x10000	0x10000

Note: For the PT_LOAD entry describing the data segment, the p_memsz may be greater than the p_filesz. The difference is the size of the .bss section. On implementations that use virtual memory file mapping, only the portion of the file between the .data p_offset (rounded down to the nearest page) to p_offset + p_filesz (rounded up to the next page size) is included. If the distance between p_offset + p_filesz and p_offset + p_memsz crosses a page boundary, then additional memory must be allocated out of anonymous memory to include data through p_vaddr + p_memsz.

Table 4-1 on page 117 demonstrates a typical mapping of file to memory segments.

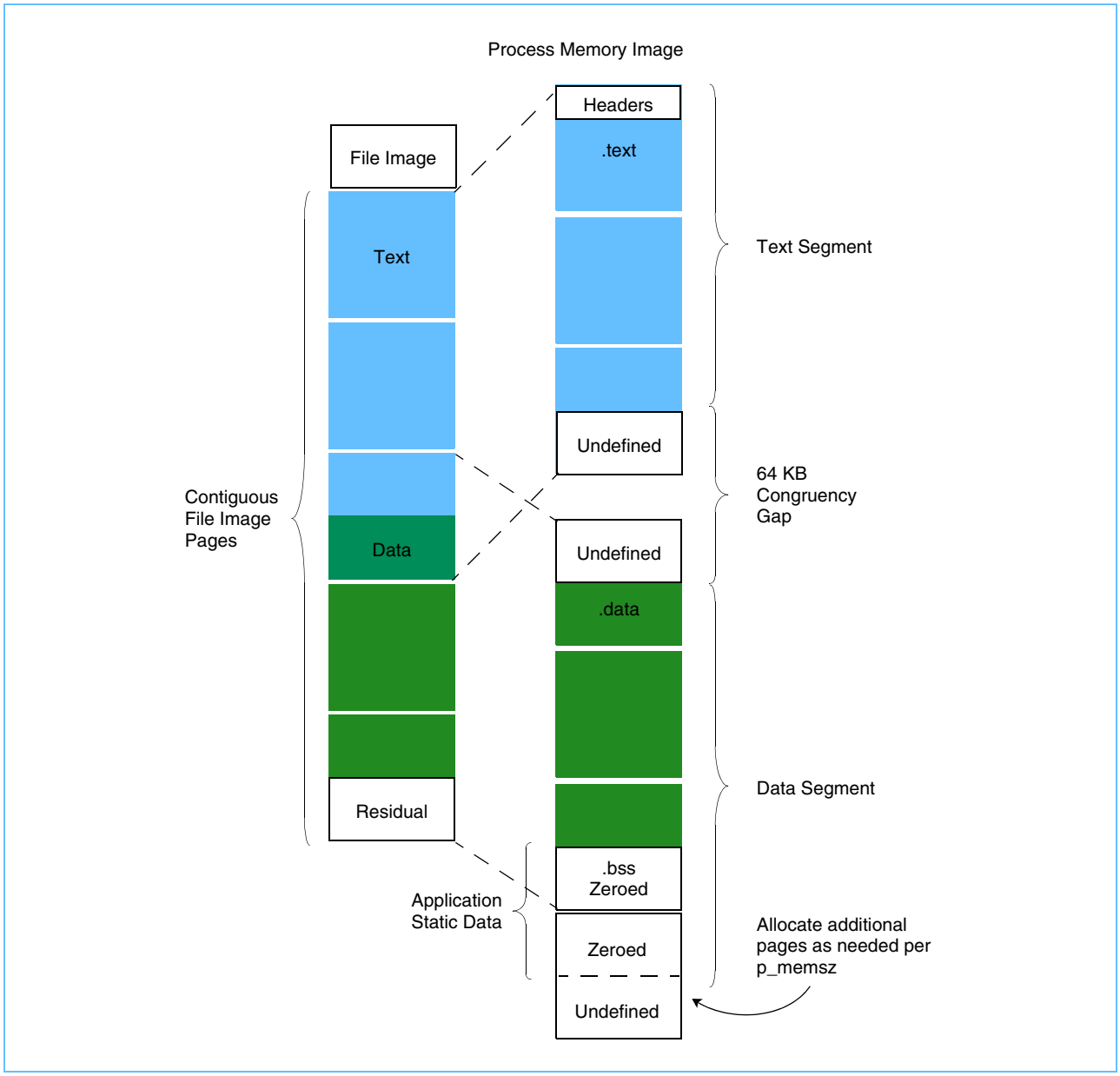
Table 4-2. Memory Segment Mappings

File	Section	Virtual Address
0x0	header	0x10000000
0x100	.text	0x10000100
0xaf0	.data	0x10010af0
0xc14	.bss	0x10010c14
0xc18	.dataend	0x10010c18

Operating systems typically enforce memory permission on a per-page granularity. This ABI maintains that the memory permissions are consistent across each memory segment when a file image is mapped to a process memory segment. The text segment and data segment require differing memory permissions. To maintain congruency of file offset to virtual address modulo the page size, the system maps the file region holding the overlapped text and data twice at different virtual addresses for each segment (see *Figure 4-1* on page 119).

To increase the security attributes of this ABI, the text and certain sections of the data segment (such as the .rodata section) may be protected as readonly after the pages are mapped and relocations are resolved. See *Section 4.2.5.2 Immediate Binding* on page 127 for more information.

Figure 4-1. File Image to Process Memory Image Mapping



As a result of this mapping, there can be up to four pages of impure text or data in the virtual memory segments for the application as described in the following list:

1. ELF header information, program headers, and other information will precede the .text section and reside at the beginning of the text segment.
2. The last memory page of the text segment can contain a copy of the partial, first file-image data page as an artifact of page faulting the last file-image text page from the file image to the text segment while maintaining the required offsets as shown in *Figure 4-1*.

3. Likewise, the first memory page of the data segment may contain a copy of the partial, last file-image text page as an artifact of page faulting the first file-image data page from the file image to the data segment while maintaining the required offsets.
4. The last faulted data-segment memory page may contain residual data from the last file-image data page that is not part of the actual file image. The system is required to zero this residual memory after that page is mapped to the data segment. If the application requires static data, the remainder of this page is used for that purpose. If the static data requirements exceed the remnant left in the last faulted memory page, additional pages shall be mapped from anonymous memory and zeroed.

Note: The handling of the contents of the first three pages is undefined by this ABI. They are unused by the executable program once started.

4.1.1 Addressing Models

When mapping an executable file or shared object file to memory, the system can use the following addressing models. Each application is allocated its own virtual address space.

- Traditionally, executable files are mapped to virtual memory using an absolute addressing model, where the mapping of the sections to segments uses the section p_vaddr specified by the ELF header directly as an absolute address.
- The position-independent code (PIC) addressing model allows the file image text of an executable file or shared object file to be loaded into the virtual address space of a process at an arbitrary starting address chosen by the kernel loader or program interpreter (dynamic linker).

Notes:

- Shared objects need to use the PIC addressing model so that all references to global variables go through the Global Offset Table.
- Position-independent executables should use the PIC addressing model.

4.1.2 Process Initialization

To provide a standard environment for application programs, the exec system call creates an initial program machine state. That state includes the use of registers, the layout of the stack frame, and argument passing. For example, a C program might typically issue the following declaration to begin executing at the local entry point of a function named main:

```
extern int main (int argc, char *argv[], char *envp[]), void *auxv[]);
int main(int argc, char *argv[], char *envp[], ElfW(auxv_t) *auxvec)
```

where:

argc is a nonnegative argument count.

argv is an array of argument strings. It is terminated by a NULL pointer, argv[argc] == 0.

envp is an array of environment strings. It is also terminated by a NULL pointer.

auxv is an array of structures that contain the auxiliary vector. It is terminated by a structure entry with an a_type of AT_NULL. For more information, see *Section 4.1.2.3 Auxiliary Vector* on page 122.

This section explains how to implement the call to main or to the entry point.

4.1.2.1 Registers

The contents of most registers are *not* specified when a process is first entered from an exec system call. A program should not expect the operating system to set all registers to 0. If a register other than those listed in *Table 4-3* must have a specific value, the program must set it to that value during process initialization.

The contents of the following registers *are* specified:

Table 4-3. Registers Specified during Process Initialization

Register	Description
r1	The initial stack pointer, aligned to a quadword boundary.
r2	Undefined.
r3	Contains argc, the nonnegative argument count.
r4	Contains argv, a pointer to the array of argument pointers in the stack. The array is immediately followed by a NULL pointer. If there are no arguments, r4 points to a NULL pointer.
r5	Contains envp, a pointer to the array of environment pointers in the stack. The array is immediately followed by a NULL pointer. If no environment exists, r5 points to a NULL pointer.
r6	Contains a pointer to the auxiliary vector. The auxiliary vector shall have at least one member, a terminating entry with an a_type of AT_NULL (see *Section 4.1.2.3 Auxiliary Vector* on page 122).
r7	Contains a termination function pointer. If r7 contains a nonzero value, the value represents a function pointer that the application should register with atexit. If r7 contains zero, no action is required.
r12	Contains the address of the global entry point of the first function being invoked, which represents the start address of the executable specified in the exec call.
FPSCR	Contains 0, specifying “round to nearest” mode for both binary and decimal rounding modes, IEEE Mode, and the disabling of floating-point exceptions.
VSCR	Vector Status and Control Register. Contains 0, specifying vector Java/IEEE mode and that no saturation has occurred.

The run-time that gets control from _start is responsible for:

- Creating the first stack frame
- Initializing the first stack frame's back chain pointer to NULL
- Allocating and initializing TLS storage
- Initializing the thread control block (TCB) and dynamic thread vector (DTV)
- Initializing any __thread variables
- Setting R13 for the initial process thread.

This initialization must be completed before any library initialization codes are run and before control is transferred to the main program (main()).

4.1.2.2 Process Stack

Although every process has a stack, no fixed stack address is defined by the system. In addition, a program's stack address can change from one system to another. It can even change from one process invocation to another. Thus, the process initialization code must use the stack address in general-purpose register r1. Data in the stack segment at addresses below the stack pointer contain undefined values.

4.1.2.3 Auxiliary Vector

The argument and environment vectors transmit information from one application program to another. However, the auxiliary vector conveys information from the operating system to the program. This vector is an array of structures, defined as follows:

```
typedef struct
{
  long    a_type;
  union
    {
      long  a_val;
      void  *a_ptr;
      void  (*a_fcn)();
    } a_un;
} auxv_t;
```

Name	Value	a_un field
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr
AT_PLATFORM	15	a_ptr
AT_HWCAP	16	a_val
AT_DCACHEBSIZE	19	a_val
AT_ICACHEBSIZE	20	a_val
AT_UCACHEBSIZE	21	a_val
AT_BASE_PLATFORM	24	a_ptr
AT_HWCAP2	26	a_val

AT_NULL

The auxiliary vector has no fixed length; instead an entry of this type denotes the end of the vector. The corresponding value of a_un is undefined.

AT_IGNORE

This type indicates that the entry has no meaning. The corresponding value of a_un is undefined.

AT_EXECFD

As Chapter 5 in the *System V ABI* describes, exec may pass control to an interpreter program. When this happens, the system places either an entry of type AT_EXECFD or one of type AT_PHDR in the auxiliary vector. The entry for type AT_EXECFD uses the a_val member to contain a file descriptor open to read the application program's object file.

AT_PHDR

Under some conditions, the system creates the memory image of the application program before passing control to an interpreter program. When this happens, the a_ptr member of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image. If the AT_PHDR entry is present, entries of types AT_PHENT, AT_PHNUM, and AT_ENTRY must also be present. See the Program Header section in Chapter 5 of the *System V ABI* for more information about the program header table.

AT_PHENT

The a_val member of this entry holds the size, in bytes, of one entry in the program header table to which the AT_PHDR entry points.

AT_PHNUM

The a_val member of this entry holds the number of entries in the program header table to which the AT_PHDR entry points.

AT_PAGESZ

If present, this entry's a_val member gives the system page size in bytes. The same information is also available through the sysconf system call.

AT_BASE

The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See the Program Header section in Chapter 5 of the *System V ABI* for more information about the base address.

AT_FLAGS

If present, the a_val member of this entry holds 1-bit flags. Bits with undefined semantics are set to zero. Other auxiliary vector types are reserved. No flags are currently defined for AT_FLAGS on the 64-bit OpenPOWER ABI Architecture.

AT_ENTRY

The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.

AT_DCACHEBSIZE

The a_val member of this entry gives the data cache block size for processors on the system on which this program is running. If the processors have unified caches, AT_DCACHEBSIZE is the same as AT_UCACHEBSIZE.

AT_ICACHEBSIZE

The a_val member of this entry gives the instruction cache block size for processors on the system on which this program is running. If the processors have unified caches, AT_ICACHEBSIZE is the same as AT_UCACHEBSIZE.

AT_UCACHEBSIZE

The a_val member of this entry is zero if the processors on the system on which this program is running do not have a unified instruction and data cache. Otherwise, it gives the cache block size.

AT_PLATFORM

The a_ptr member is the address of the platform name string. For virtualized systems, this may be different (that is, an older platform) than the physical machine running this environment.

AT_BASE_PLATFORM

The a_ptr member is the address of the platform name string for the physical machine. For virtualized systems, this will be the platform name of the real hardware.

AT_HWCAP

The a_val member of this entry is a bit map of hardware capabilities. Some bit mask values include:

```
PPC_FEATURE_32              0x80000000 /* Always set for powerpc64 */
PPC_FEATURE_64              0x40000000 /* Always set for powerpc64 */
PPC_FEATURE_HAS_ALTIVEC     0x10000000
PPC_FEATURE_HAS_FPU         0x08000000
PPC_FEATURE_HAS_MMU         0x04000000
PPC_FEATURE_UNIFIED_CACHE   0x01000000
PPC_FEATURE_NO_TB           0x00100000 /* 601/403gx have no timebase */
PPC_FEATURE_POWER4          0x00080000 /* POWER4 ISA 2.00 */
PPC_FEATURE_POWER5          0x00040000 /* POWER5 ISA 2.02 */
PPC_FEATURE_POWER5_PLUS     0x00020000 /* POWER5+ ISA 2.03 */
PPC_FEATURE_CELL_BE         0x00010000 /* CELL Broadband Engine */
PPC_FEATURE_BOOKE           0x00008000 /* ISA Category Embedded */
PPC_FEATURE_SMT             0x00004000 /* Simultaneous Multi-Threading */
PPC_FEATURE_ICACHE_SNOOP    0x00002000
PPC_FEATURE_ARCH_2_05       0x00001000 /* ISA 2.05 */
PPC_FEATURE_PA6T            0x00000800 /* PA Semi 6T Core */
PPC_FEATURE_HAS_DFP         0x00000400 /* Decimal FP Unit */
PPC_FEATURE_POWER6_EXT      0x00000200 /* P6 + mffgpr/mftgpr */
PPC_FEATURE_ARCH_2_06       0x00000100 /* ISA 2.06 */
PPC_FEATURE_HAS_VSX         0x00000080 /* P7 Vector Extension.  */
PPC_FEATURE_PSERIES_PERFMON_COMPAT  0x00000040
PPC_FEATURE_TRUE_LE         0x00000002
PPC_FEATURE_PPC_LE          0x00000001
```

AT_HWCAP2

The a_val member of this entry is a bit map of hardware capabilities. Some bit mask values include:

```
PPC_FEATURE2_ARCH_2_07      0x80000000 /* ISA 2.07 */
PPC_FEATURE2_HAS_HTM        0x40000000 /* Hardware Transactional Memory */
PPC_FEATURE2_HAS_DSCR       0x20000000 /* Data Stream Control Register */
PPC_FEATURE2_HAS_EBB        0x10000000 /* Event Base Branching */
PPC_FEATURE2_HAS_ISEL       0x08000000 /* Integer Select */
PPC_FEATURE2_HAS_TAR        0x04000000 /* Target Address Register */
PPC_FEATURE2_HAS_VCRYPTO    0x02000000
```

When a process starts to execute, its stack holds the arguments, environment, and auxiliary vector received from the exec call. The system makes no guarantees about the relative arrangement of argument strings, environment strings, and the auxiliary information, which appear in no defined or predictable order. Further, the system may allocate memory after the null auxiliary vector entry and before the beginning of the information block.

4.2 Dynamic Linking

4.2.1 Program Interpreter

For dynamic linking, the standard program interpreter is /lib/ld64.so.2. It may be located in different places on different distributions.

4.2.2 Dynamic Section

The dynamic section provides information used by the dynamic linker to manage dynamically loaded shared objects, including relocation, initialization, and termination when loaded or unloaded, resolving dependencies on other shared objects, resolving references to symbols in the shared object, and supporting debugging. The following dynamic tags are relevant to this processor-specific ABI:

DT_PLTGOT

The d_ptr member of this dynamic tag holds the address of the first byte of the Procedure Linkage Table (PLT).

DT_JMPREL

The d_ptr member of this dynamic tag points to the first byte of the table of relocation entries, which have a one-to-one correspondence with PLT entries. Any executable or shared object with a PLT must have DT_JMPREL. A shared object containing only data will not have a PLT and thus will not have DT_JMPREL.

DT_PPC64_GLINK (DT_LOPROC + 0)

The d_ptr member of this dynamic tag points to 32 bytes before the .glink lazy link symbol resolver stubs that are described in *Section 4.2.5.3 Procedure Linkage Table* on page 128.

DT_PPC64_OPT (DT_LOPROC + 3)

The d_val member of this dynamic tag specifies whether various optimizations are possible. The low bit will be set to indicate that an optimized __tls_get_addr call stub is used. The next most significant bit will be set if multiple TOCs are present.

4.2.3 Global Offset Table

To support position-independent code, a Global Offset Table (GOT) shall be constructed by the link editor in the data segment when linking code that contains any of the various R_PPC64_GOT* relocations or when linking code that references the .TOC address. The GOT consists of an 8-byte header that contains the TOC base (the first TOC base when multiple TOCs are present), followed by an array of 8-byte addresses. The link editor shall emit dynamic relocations as appropriate for each entry in the GOT. At runtime, the dynamic linker

will apply these relocations after the addresses of all memory segments are known (and thus the addresses of all symbols). At that point, the GOT may be considered to be an array of absolute addresses, but note that this ABI does not preclude the GOT containing nonaddress entries.

Absolute addresses are generated for all GOT relocations by the dynamic linker before giving control to any process image code. The dynamic linker is free to choose different memory segment addresses for the executable or shared objects in a different process image. After the initial mapping of the process image by the dynamic linker, memory segments reside at fixed addresses for the life of a process.

The symbol .TOC. may be used to access the GOT or in TOC-relative addressing to other data constructs, such as the procedure linkage table. The symbol may be offset by 0x8000 bytes, or another offset, from the start of the .got section. This offset allows the use of the full (64 KB) signed range of 16-bit displacement fields by using both positive and negative subscripts into the array of addresses, or a larger offset to afford addressing using references within ±2 GB with 32-bit displacements. The 32-bit displacements are constructed by using the addis instruction to provide a first high-order 16-bit portion of a 32-bit displacement in conjunction with an instruction to supply a low-order 16-bit portion of a 32-bit displacement.

In PIC code, the TOC pointer r2 points to the TOC base, enabling easy reference. For static nonrelocatable modules, the GOT address is fixed and can be directly used by code making reference to the stub.

All functions except leaf routines must load the value of the TOC base into the TOC register r2.

4.2.4 Function Addresses

The following requirements concern function addresses.

When referencing a function address, consider the following requirements:

- Intraobject executable or shared object function address references may be resolved by the dynamic linker to the absolute virtual address of the symbol.
- Function address references from within the executable file to a function defined in a shared object file are resolved by the link editor to the .text section address of the PLT call stub for that function within the executable file.
- In a static module, when a function pointer reference is made to a function provided by a dynamically loaded shared module, the function may be resolved to the address of a PLT stub. If this resolution is made, all function pointer references must be made through the same PLT stub in the static module to ensure correct intraobject comparisons for function addresses.
- A function address of a nested function *may* also be resolved to the address of a trampoline used to call it.

When comparing function addresses, consider the following requirements:

- The address of a function shall compare to the same value in executables and shared objects.
- For intraobject comparisons of function addresses within the executable or shared object, the link editor may directly compare the absolute virtual addresses.
- For a function address comparison where an executable references a function defined in a shared object, the link editor will place the address of a .text section PLT call stub for that function in the corresponding dynamic symbol table entry's st_value field (see *Section 3.4.1 Symbol Values* on page 87).
- When the dynamic linker loads shared objects associated with an executable and resolves any GOT entry relocations into absolute addresses, it will search the dynamic symbol table of the executable for each symbol that needs to be resolved.

- If it finds the symbol and the st_value of the symbol table entry is nonzero, it shall use the address indicated in the st_value entry as the symbol's address. If the dynamic linker does not find the symbol in the executable's dynamic symbol table or the entry's st_value member is zero, the dynamic linker may consider the symbol as undefined in the executable file.

4.2.5 Procedure Linkage Table

When the link editor builds an executable file or shared object file, it does not know the absolute address of undefined function calls. Therefore, it cannot generate code to directly transfer execution to another shared object or executable. For each execution transfer to an undefined function call in the file image, the link editor places a relocation against an entry in the Procedure Linkage Table (PLT) of the executable or shared object that corresponds to that function call.

Additionally, for all nonstatic functions with standard (nonhidden) visibility in a shared object, the link editor invokes the function through the PLT, even if the shared object defines the function. The same is not true for executables.

The link editor knows the number of functions invoked through the PLT, and it reserves space for an appropriately sized .plt section. The .plt section is located in the section following the .got. It consists of an array of addresses and is initialized by the module loader. There will also be an array of R_PPC_JMP_SLOT relocations in .rela.plt, with a one-to-one correspondence between elements of each array. Each R_PPC_JMP_SLOT reloc will have r_offset pointing at the .plt word it relocates.

A unique PLT is constructed by the static linker for each static module (that is, the main executable) and each dynamic shared object. The PLT is located in the data segment of the process image at object load time by the dynamic linker using the information about the .plt section stored in the file image. The individual PLT entries are populated by the dynamic linker using one of the following binding methods. Execution can then be redirected to a dependent shared object or executable.

4.2.5.1 Lazy Binding

The lazy binding method is the default. It delays the resolution of a PLT entry to an absolute address until the function call is made the first time. The benefit of this method is that the application does not pay the resolution cost until the first time it needs to call the function, if at all.

To implement lazy binding, the dynamic loader points each PLT entry to a lazy resolution stub at load time. After the function call is made the first time, this lazy resolution stub gets control, resolves the symbol, and updates the PLT entry to hold the final value to be used for future calls.

4.2.5.2 Immediate Binding

The immediate binding method resolves the absolute addresses of all PLT entries in the executable and dependent shared objects at load time, before passing execution control to the application. The environment variable LD_BIND_NOW may be set to a nonnull value to signal the dynamic linker that immediate binding is requested at load time, before control is given to the application.

For some performance-sensitive situations, it may be better to pay the resolution cost to populate the PLT entries up front rather than during execution.

4.2.5.3 Procedure Linkage Table

For every call site that needs to use the PLT, the link editor constructs a call stub in the .text section and resolves the call site to use that call stub. The call stub transfers control to the address indicated in the PLT entry. These call stubs need not be adjacent to one another or unique. They can be scattered throughout the text segment so that they can be reached with a branch and link instruction.

Depending on relocation information at the call site, the stub provides one of the following properties:

1. The caller has set up r2 to hold the TOC pointer and expects the PLT call stub to save that value to the TOC save stack slot. This is the default.
2. The caller has set up r2 to hold the TOC pointer and has already saved that value to the TOC save stack slot itself. This is indicated by the presence of a R_PPC64_TOCSAVE relocation on the nop following the call.
3. The caller has not set up r2 to hold the TOC pointer. This is indicated by use of a R_PPC64_REL24_NOTOC relocation (instead of R_PPC64_REL24) on the call instruction.

In any scenario, the PLT call stub must transfer control to the function whose address is provided in the associated PLT entry. This address is treated as a global entry point for ABI purposes. This means that the PLT call stub loads the address into r12 before transferring control.

Although the details of the call stub implementation are left to the link editor, some examples are provided. In those examples, func@plt is used to denote the address of the PLT entry for func; func@plt@toc denotes the offset of that address relative to the TOC pointer; and the @ha and @l variants denote the high-adjusted and low parts of these values as usual. Because the link editor synthesizes the PLT call stubs directly, it can determine all these values as immediate constants. The assembler is not required to support those notations.

A possible implementation for case 1 looks as follows (if func@plt@toc is less than 32 KB, the call stub may be simplified to omit the addis):

```
    std   r2,24(r1)
    addis r12,r2,func@plt@toc@ha
    ld    r12,func@plt@toc@l(r12)
    mtctr r12
    bctr
```

For case 2, the same implementation as for case 1 may be used, except that the first instruction “std r2,24(r1)” is omitted:

```
    addis r12,r2,func@plt@toc@ha
    ld    r12,func@plt@toc@l(r12)
    mtctr r12
    bctr
```

A possible implementation for case 3 looks as follows:

```
    mflr  r0
    bcl   20,31,1f
1:  mflr  r2
    mtlr  r0
    addis r2,r2,(.TOC.-1b)@ha
    addi  r2,r2,(.TOC.-1b)@l
    addis r12,r2,func@plt@toc@ha
```

```
    ld    r12,func@plt@toc@l(r12)
    mtctr r12
    bctr
```

When generating non-PIC code for the small or medium code model, a simpler variant may alternatively be used for cases 2 or 3:

```
    lis   r12,func@plt@ha
    ld    r12,func@plt@l(r12)
    mtctr r12
    bctr
```

To support lazy binding, the link editor also provides a set of symbol resolver stubs, one for each PLT entry. Each resolver stub consists of a single instruction, which is usually a branch to a common resolver entry point or a nop. The resolver stubs are placed in the .glink section, which is merged into the .text section of the final executable or dynamic object. The address of the resolver stubs is communicated to the dynamic loader through the DT_PPC64_GLINK dynamic section entry. The address of the symbol resolver stub associated with PLT entry N is determined by adding 4xN + 32 to the d_ptr field of the DT_PPC64_GLINK entry. When using lazy binding, the dynamic linker initializes each PLT entry at load time to that address.

The resolver stubs provided by the link editor must call into the main resolver routine provided by the dynamic linker. This resolver routine must be called with r0 set to the index of the PLT entry to be resolved, r11 set to the identifier of the current dynamic object, and r12 set to the resolver entry point address (as usual when calling a global entry point). The resolver entry point address and the dynamic object identifier are installed at load time by the dynamic linker into the two doublewords immediately preceding the array of PLT entries, allowing the resolver stubs to retrieve these values from there. These two doublewords are considered part of the .plt section; the DT_PLTGOT dynamic section entry points to the first of those words.

Beyond the above requirements, the implementation of the .glink resolver stubs is up to the link editor. The following shows an example implementation:

```
    # ABI note: At entry to the resolver stub:
    # - r12 holds the address of the res_N stub for the target routine
    # - all argument registers hold arguments for the target routine
PLTresolve:
    # Determine addressability. This sequence works for both PIC
    # and non-PIC code and does not rely on presence of the TOC pointer.
    mflr    r0
    bcl     20,31,1f
1:  mflr    r11
    mtlr    r0
    # Compute .plt section index from entry point address in r12
    # .plt section index is placed into r0 as argument to the resolver
    sub     r0,r12,r11
    subi    r0,r0,res_0-1b
    srdi    r0,r0,2
    # Load address of the first byte of the PLT
    ld      r12,PLToffset-1b(r11)
    add     r11,r12,r11
    # Load resolver address and DSO identifier from the
    # first two doublewords of the PLT
    ld      r12,0(r11)
```

```
    ld      r11,8(r11)
    # Branch to resolver
    mtctr   r12
    bctr
    # ABI note: At entry to the resolver:
    # - r12 holds the resolver address
    # - r11 holds the DSO identifier
    # - r0 holds the PLT index of the target routine
    # - all argument registers hold arguments for the target routine

    # Constant pool holding offset to the PLT
    # Note that there is no actual symbol PLT; the link editor
    # synthesizes this value when creating the .glink section
PLToffset:
    .quad PLT-.

    # A table of branches, one for each PLT entry
    # The idea is that the PLT call stub loads r12 with these
    # addresses, so (r12 - res_0) gives the PLT index × 4.

res_0:b PLTresolve
res_1:b PLTresolve
    ...
```

After resolution, the value of a PLT entry in the PLT is the address of the function's global entry point, unless the resolver can determine that a module-local call occurs with a shared TOC value wherein the TOC is shared between the caller and the callee.

5. Libraries

5.1 Library Requirements

This ABI does not specify any additional interfaces for general-purpose libraries. However, certain processor-specific support routines are defined to ensure portability between ABI-conforming implementations.

Such processor-specific support definitions concern vector and floating-point alignment, register save and restore routines, variable argument list layout, and a limited set of data definitions.

5.1.1 C Library Conformance with Generic ABI

5.1.1.1 Malloc Routine Return Pointer Alignment

The malloc() routine must always return a pointer with the alignment of the largest alignment needed for loads and stores of the built-in data types. This is currently 16 bytes.

5.1.1.2 Library Handling of Limited-Access Bits in Registers

Requirements for the handling of limited-access bits in certain registers by standard library functions are defined in *Section 2.2.1.2 Limited-Access Bits* on page 38.

5.1.2 Save and Restore Routines

All of the save and restore routines described in *Section 2.3.3 Register Save and Restore Functions* on page 63 are required. These routines use unusual calling conventions due to their special purpose. Parameters to these functions are described in *Section 2.3.3.1 GPR Save and Restore Functions* on page 63, *Section 2.3.3.2 FPR Save and Restore Functions* on page 65, and *Section 2.3.3.3 Vector Save and Restore Functions* on page 67.

The symbols for these functions shall be hidden and locally resolved within each module. The symbols so created shall not be exported.

These functions can either be provided in a utility library that is linked by the linker to each module, or the functions can be synthesized by the linker as necessary to resolve symbols.

5.1.3 Types Defined in Standard Header

The type va_list shall be defined as follows:

```
typedef void * va_list;
```

The following integer types are defined in headers required to be provided by freestanding implementations, or have their limits defined in such headers, and shall have the following definitions.

- typedef long long ptrdiff_t;
- typedef unsigned int size_t;
- typedef long wchar_t;
- typedef int sig_atomic_t;
- typedef unsigned int wint_t;

- typedef signed char int8_t;
- typedef short int16_t;
- typedef int int32_t;
- typedef long long int64_t;
- typedef unsigned char uint8_t;
- typedef unsigned short uint16_t;
- typedef unsigned int uint32_t;
- typedef unsigned long long uint64_t;
- typedef signed char int_least8_t;
- typedef short int_least16_t;
- typedef int int_least32_t;
- typedef long long int_least64_t;
- typedef unsigned char uint_least8_t;
- typedef unsigned short uint_least16_t;
- typedef unsigned int uint_least32_t;
- typedef unsigned long long uint_least64_t;
- typedef signed char int_fast8_t;
- typedef int int_fast16_t;
- typedef int int_fast32_t;
- typedef long long int_fast64_t;
- typedef unsigned char uint_fast8_t;
- typedef unsigned int uint_fast16_t;
- typedef unsigned int uint_fast32_t;
- typedef unsigned long long uint_fast64_t;
- typedef long long intptr_t;
- typedef unsigned long long uintptr_t;
- typedef long long intmax_t;
- typedef unsigned long long uintmax_t;

5.2 POWER ISA-Specific API and ABI Extensions

The Data Stream Control Register (DSCR) affects how the processor handles data streams that are detected by the hardware and defined by the software. For more information, see “Data Stream Control Overview, ABI, and API” at the following link:

https://github.com/paflib/paflib/wiki/Data-Stream-Control-Overview,-ABI,-and-API

The event-based branching facility generates exceptions when certain criteria are met. For more information, see the “Event Based Branching Overview, ABI, and API” section at the following link:

https://github.com/paflib/paflib/wiki/Event-Based-Branching----Overview,-ABI,-and-API

6. Vector Programming Interfaces

To ensure portability of applications optimized to exploit the SIMD functions of Power ISA processors, the ELF V2 ABI defines a set of functions and data types for SIMD programming. ELF V2-compliant compilers will provide suitable support for these functions, preferably as built-in functions that translate to one or more Power ISA instructions.

Compilers are encouraged, but not required, to provide built-in functions to access individual instructions in the IBM POWER® instruction set architecture. In most cases, each such built-in function should provide direct access to the underlying instruction.

However, to ease porting between little-endian (LE) and big-endian (BE) POWER systems, and between POWER and other platforms, it is preferable that some built-in functions provide the same semantics on both LE and BE POWER systems, even if this means that the built-in functions are implemented with different instruction sequences for LE and BE. To achieve this, vector built-in functions provide a set of functions derived from the set of hardware functions provided by the Power vector SIMD instructions. Unlike traditional “hardware intrinsic” built-in functions, no fixed mapping exists between these built-in functions and the generated hardware instruction sequence. Rather, the compiler is free to generate optimized instruction sequences that implement the semantics of the program specified by the programmer using these built-in functions.

This is primarily applicable to the vector facility of the POWER ISA, also known as Power SIMD, consisting of the VMX (or Altivec) and VSX instructions. This set of instructions operates on groups of 2, 4, 8, or 16 vector elements at a time in 128-bit registers. On a big-endian POWER platform, vector elements are loaded from memory into a register so that the 0th element occupies the high-order bits of the register, and the (N-1)th element occupies the low-order bits of the register. This is referred to as big-endian element order. On a little-endian POWER platform, vector elements are loaded from memory such that the 0th element occupies the low-order bits of the register, and the (N-1)th element occupies the high-order bits. This is referred to as little-endian element order.

6.1 Vector Data Types

Languages provide support for the data types in *Table 2-7* on page 22 to represent vector data types stored in vector registers.

For the C and C++ programming languages (and related/derived languages), these data types may be accessed based on the type names listed in *Table 2-7* on page 22 when Power ISA SIMD language extensions are enabled using either the vector or __vector keywords.

For the FORTRAN language, *Table 6-1* gives a correspondence of FORTRAN and C/C++ language types.

Table 6-1. FORTRAN Vector Data Types (Page 1 of 2)

XL Fortran Vector Type	XL C/C++ Vector Type
VECTOR(INTEGER(1))	vector signed char
VECTOR(INTEGER(2))	vector signed short
VECTOR(INTEGER(4))	vector signed int
VECTOR(INTEGER(8))	vector signed long long, vector signed long
VECTOR(UNSIGNED(1))	vector unsigned char
Note: If FORTRAN equivalents for vector bool types are supported, constructing them as VECTOR(LOGICAL(n)) is recommended.	

Table 6-1. FORTRAN Vector Data Types (Page 2 of 2)

XL Fortran Vector Type	XL C/C++ Vector Type
VECTOR(UNSIGNED(2))	vector unsigned short
VECTOR(UNSIGNED(4))	vector unsigned int
VECTOR(UNSIGNED(8))	vector unsigned long long, vector unsigned long
VECTOR(REAL(4))	vector float
VECTOR(REAL(8))	vector double
VECTOR(PIXEL)	vector pixel
Note: If FORTRAN equivalents for vector bool types are supported, constructing them as VECTOR(LOGICAL(n)) is recommended.	

The assignment operator = is the preferred way to assign values from one vector data type to another vector data type in accordance with the C and C++ programming languages. Type casts and pointer references may be used as appropriate:

```
vector char vca;
vector char vcb;
vector int via;
int a[4];
void *vp;

via = *(vector int *) &a[0];
vca = (vector char) via;
vcb = vca;
vca = *(vector char *)vp;
*(vector char *)&a[0] = vca;
```

The assignment operator always performs a byte-by-byte data copy in accordance with the application's endian mode.

6.2 Vector Layout and Element Numbering

Vector data types consist of a homogeneous sequence of elements of the base data type specified in the vector data type. Individual elements of a vector can be addressed by a vector element number. Element numbers can be established either by counting from the “left” of a register and assigning the left-most element the element number 0, or from the “right” of the register and assigning the right-most element the element number 0.

In big-endian environments, establishing element counts from the left makes the element stored at the lowest memory address the lowest numbered element. Thus, when vectors and arrays of a given base data type are overlayed, vector element 0 corresponds to array element 0, vector element 1 corresponds to array element 1, and so forth.

In little-endian environments, establishing element counts from the right makes the element stored at the lowest memory address the lowest numbered element. Thus, when vectors and arrays of a given base data type are overlaid, vector element 0 will correspond to array element 0, vector element 1 will correspond to array element 1, and so forth.

Consequently, the vector numbering schemes can be described as big-endian and little-endian vector layouts and vector element numberings. (The term “endian” comes from the endian debates presented in *Gulliver's Travels* by Jonathan Swift.)

For internal consistency, in the ELF V2 ABI, the default vector layout and vector element ordering in big-endian environments shall be big-endian, and the default vector layout and vector element ordering in little-endian environments shall be little-endian.

This element numbering shall also be used by the [] accessor method to vector elements provided as an extension of the C/C++ languages by some compilers, as well as for other language extensions or library constructs that directly or indirectly refer to elements by their element number.

6.3 Vector Built-in Functions

The Power language environments provide a well-known set of built-in functions for the Power SIMD instructions (including both Altivec/VMX and VSX). A full description of these built-in functions is beyond the scope of this ABI document. Most built-in functions are polymorphic, operating on a variety of vector types (vectors of signed characters, vectors of unsigned halfwords, and so forth).

Some of the Power SIMD (VMX/Altivec and/or VSX) hardware instructions refer, implicitly or explicitly, to vector element numbers. For example, the vspltb instruction has as one of its inputs an index into a vector. The element at that index position is to be replicated in every element of the output vector. For another example, the vmuleuh instruction operates on the even-numbered elements of its input vectors. The hardware instructions define these element numbers using big-endian element order, even when the machine is running in little-endian mode. Thus, a built-in function that maps directly to the underlying hardware instruction, regardless of the target endianness, has the potential to confuse programmers on little-endian platforms.

It is more useful to define built-in functions that map to these instructions to use natural element order. That is, the explicit or implicit element numbers specified by such built-in functions should be interpreted using big-endian element order on a big-endian platform, and using little-endian element order on a little-endian platform.

This ABI defines the following built-in functions to use natural element order. The Implementation Notes column suggests possible ways to implement little-endian (LE) versions of the built-in functions, although designers of a compiler are free to use other methods to implement the specified semantics as they see fit.

Table 6-2. Endian-Sensitive Data Types (Page 1 of 2)

Built-In Function	Corresponding POWER Instructions	Implementation Notes
vec_extract	None	vec_extract (v, 3) is equivalent to v[3].
vec_insert	None	vec_insert (v, 3, x) is equivalent to v[3] = x.
vec_mergee	vmrgew	Swap inputs and use vmrgow for LE. Phased in.[a]
vec_mergeh	vmrghb, vmrghh, vmrghw	Swap inputs and use vmrglb, and so on, for LE.
vec_mergel	vmrglb, vmrglh, vmrglw	Swap inputs and use vmrghb, and so on, for LE.
vec_mergeo	vmrgow	Swap inputs and use vmrgew for LE. Phased in.[a]
vec_mule	vmuleub, vmulesb, vmuleuh, vmulesh	Replace with vmuloub, and so on, for LE.
vec_mulo	vmuloub, vmulosb, vmulouh, vmulosh	Replace with vmuleub, and so on, for LE.
vec_pack	vpkuhum, vpkuwum	Swap input arguments for LE.

Table 6-2. Endian-Sensitive Data Types (Page 2 of 2)

Built-In Function	Corresponding POWER Instructions	Implementation Notes
vec_packpx	vpkpx	Swap input arguments for LE.
vec_packs	vpkuhus, vpkshss, vpkuwus, vpkswss	Swap input arguments for LE.
vec_packsu	vpkuhus, vpkshus, vpkuwus, vpkswus	Swap input arguments for LE.
vec_perm	vperm	For LE, swap input arguments and complement the selection vector.
vec_splat	vspltb, vsplth, vspltw	Subtract the element number from N-1 for LE.
vec_sum2s	vsum2sws	For LE, swap elements 0 and 1, and elements 2 and 3, of the second input argument; then swap elements 0 and 1, and elements 2 and 3, of the result vector.
vec_sums	vsumsws	For LE, use element 3 in little-endian order from the second input vector, and place the result in element 3 in little-endian order of the result vector.
vec_unpackh	vupkhsb, vupkhpx, vupkhsh	Use vupklsb, and so on, for LE.
vec_unpackl	vupklsb, vupklpx, vupklsh	Use vupkhsb, and so on, for LE.

a. This optional function is being phased in, and it may not be available on all implementations.

Reminder: The assignment operator = is the preferred way to assign values from one vector data type to another vector data type in accordance with the C and C++ programming languages.

Extended Data Movement Functions

The built-in functions in *Table 6-3* map to Altivec/VMX load and store instructions and provide access to the “auto-aligning” memory instructions of the Altivec ISA where low-order address bits are discarded before affecting a memory accesses. These instructions access load and store data in accordance with the program's current endian mode, and do not need to be adapted by the compiler to reflect little-endian operating during code generation:

Table 6-3. Altivec Memory Access Built-In Functions

Built-in Function	Corresponding POWER Instructions	Implementation Notes
vec_ld	lvx	Hardware works as a function of endian mode.
vec_lde	lvebx, lvehx, lvewx	Hardware works as a function of endian mode.
vec_ldl	lvxl	Hardware works as a function of endian mode.
vec_st	stvx	Hardware works as a function of endian mode.
vec_ste	stvebx, stvehx, stvewx	Hardware works as a function of endian mode.
vec_stl	stvxl	Hardware works as a function of endian mode.

Previous versions of the Altivex built-in functions defined intrinsics to access the Altivec instructions lvsl and lvsr, which may be used in conjunction with vec_vperm and Altivec load and store instructions for unaligned access. These instructions are deprecated, and will not interoperate with little-endian data layout in conjunction with the little-endian vector layout described in this document. Consequently, the built-in pseudo sequences published in previous VMX documents are not applicable to this little-endian environment. It is recommended (but not required) that compilers issue a warning when these functions are used in little-endian environments. It is recommended that programmers use the assignment operator = or the vector vec_xl and vec_xst vector built-in functions to access unaligned data streams.

The set of extended mnemonics in *Table 6-4* on page 137 may be provided by some compilers and are not required by the Power SIMD programming interfaces. In particular, the assignment operator = will have the same effect of copying values between vector data types and provides a preferable method to assign values while giving the compiler more freedom to optimize data allocation. The only use for these functions is to support some coding patterns enabling big-endian vector layout code sequences in both big-endian and little-endian environments. Memory access built-in functions that specify a vector element format (that is, the w4 and d2 forms) are deprecated. They will be phased out in future versions of this specification because vec_xl and vec_xst provide overloaded layout-specific memory access based on the specified vector data type.

Table 6-4. Optional Built-In Memory Access Functions

Built-in Function	Corresponding POWER Instructions	Little-Endian Implementation Notes
vec_xl	lxvd2x	lxvd2x ; xxpermdi
vec_xlw4[1]	lxvw4x	lxvd2x ; xxpermdi
vec_xld2[1]	lxvd2x	lxvd2x ; xxpermdi
vec_xst	stxvd2x	xxpermdi ; stxvd2x
vec_xstw4[1]	stxvw4x	xxpermdi ; stxvd2x
vec_xstd2[1]	stxvd2x	xxpermdi ; stxvd2x

1. Deprecated. The use of vector data type assignment and overloaded vec_xl and vec_xst vector built-in functions are preferred forms for assigning vector operations. Similarly, the use of __builtin_lxvd2x, __builtin_lxvw4x, __builtin_stxvd2x, __builtin_stxvw4x, available in some compilers, is discouraged.

The two optional built-in vector functions in *Table 6-5* on page 137 can be used to load and store vectors with a big-endian element ordering (that is, bytes from low to high memory will be loaded from left to right into a vector char variable), independent of the -qaltivec=be or -maltivec=be setting. For more information, see *Section 6.3.1 Big-Endian Vector Layout in Little-Endian Environments* on page 137.

Table 6-5. Optional Fixed Data Layout Built-In Vector Functions

Built-in Function	Corresponding POWER Instructions	Little-Endian Implementation Notes
vec_xl_be	lxvd2x	Use lxvd2x for vector long long; vector long, vector double. Use lxvd2x followed by reversal of elements within each doubleword for all other data types.
vec_xst_be	stxvd2x	Use stxvd2x for vector long long; vector long, vector double. Use stxvd2x following a reversal of elements within each doubleword for all other data types.

6.3.1 Big-Endian Vector Layout in Little-Endian Environments

Because the vector layout and element numbering cannot be represented in source code in an endian-neutral manner, code originating from big-endian platforms may need to be compiled on little-endian platforms, or vice versa. To simplify such application porting, some compilers may provide an additional bridge mode to enable a simplified porting for some applications.

It should be noted that such support only works for homogeneous data being loaded into vector registers (that is, no unions or structs containing elements of different sizes) and when those vectors are loaded from and stored to memory with element-size-specific built-in vector memory functions of *Table 6-6* and *Table 6-7*. That is because, in this mode, data within each element needs to be adjusted for little-endian data representation while providing a big-endian layout and numbering of vector elements within a vector.

Note: Because of the internal contradiction of big-endian vector layouts and little-endian data, such an environment will have intrinsic limitations for the type of functionality that may be offered. However, it may provide a useful bridge in the porting of code using vector built-ins between environments having different data layout models.

Compiler designers may implement additional built-in functions or other mechanisms that use big-endian element ordering in little-endian mode. For example, the GCC and XL compilers define the options -maltivec=be and -qaltivec=be, respectively, to allow programmers to specify that the above built-ins will generate big-endian hardware instructions directly for the corresponding big-endian sequences in little-endian mode. To ensure consistent element operation, in this mode, the lvx instructions and related instructions are changed to maintain a big-endian data layout in registers by adding appropriate permute sequences as shown in *Table 6-6*.

Table 6-6. Altivec Built-In Vector Memory Access Functions (BE Layout In LE Mode)

Built-In Function	Corresponding POWER Instructions	BE Vector Layout in Little-Endian Mode Implementation Notes
vec_ld	lvx	Reverse elements with a vperm after load for LE based on vector base type.
vec_lde	lvebx, lvehx, lvewx	Reverse elements with a vperm after load for LE based on vector base type.
vec_ldl	lvxl	Reverse elements with a vperm after load for LE based on vector base type.
vec_st	stvx	Reverse elements with a vperm before store for LE based on vector base type.
vec_ste	stvebx, stvehx, stvewx	Reverse elements with a vperm before store for LE based on vector base type.
vec_stl	stvxl	Reverse elements with a vperm before store for LE based on vector base type.

Access to memory instructions handling potentially unaligned accesses may be accomplished using instructions (or instruction sequences) that perform little-endian load of the underlying vector data type while maintaining big-endian element ordering. See *Table 6-7*.

Table 6-7. Optional Built-In Memory Access Functions (BE Layout In LE Mode) (Page 1 of 2)

Built-In Function	Corresponding POWER Instructions	BE Vector Layout in Little-Endian Mode Implementation Notes
vec_xl	lxvd2x	lxvd2x - for vector long long; vector long, vector double. lxvw4x - for vector int; vector float. lxvd2x followed by reversal of elements within each doubleword – for all other data types.
vec_xlw[1]	lxvw4x	lxvw4x.
vec_xld2[1]	lxvd2x	lxvd2x.

1. Deprecated. The use of vector data type assignment and overloaded vec_xl and vec_xst vector built-in functions are preferred forms for assigning vector operations. Similarly, the use of __builtin_lxvd2x, __builtin_lxvw4x, __builtin_stxvd2x, __builtin_stxvw4x, available in some compilers, is discouraged.

Table 6-7. Optional Built-In Memory Access Functions (BE Layout In LE Mode) (Page 2 of 2)

Built-In Function	Corresponding POWER Instructions	BE Vector Layout in Little-Endian Mode Implementation Notes
vec_xst	stxvd2x	stxvd2x - for vector long long; vector long, vector double. stxvw4x - for vector int; vector float. stxvd2x following a reversal of elements within each doubleword – for all other data types.
vec_xstw4[1]	stxvw4x	stxvw4x.
vec_xstd2[1]	stxvd2x	stxvd2x.

1. Deprecated. The use of vector data type assignment and overloaded vec_xl and vec_xst vector built-in functions are preferred forms for assigning vector operations. Similarly, the use of __builtin_lxvd2x, __builtin_lxvw4x, __builtin_stxvd2x, __builtin_stxvw4x, available in some compilers, is discouraged.

Note: The use of -maltivec=be or -qaltivec=be in little-endian mode disables the transformations described in *Table 6-2* on page 135.

The operation of the assignment operator is never changed by a setting such as -qaltivec=be or -maltivec=be.

6.4 Library Interfaces

6.4.1 printf and scanf of Vector Data Types

Support for vector variable input and output *may* be provided as an extension to the following POSIX library functions for the new vector conversion format strings:

- scanf
- fscanf
- sscanf
- wsscanf
- printf
- fprintf
- sprintf
- snprintf
- wsprintf
- vprintf
- vfprintf
- vsprintf
- vwsprintf

(One sample implementation for such an extended specification is libvecprintf.)

The new size formatters are as follows:

- vl or lv consumes one argument and modifies an existing integer conversion, resulting in vector signed int, vector unsigned int, or vector bool for output conversions or vector signed int * or vector unsigned int * for input conversions. The data is then treated as a series of four 4-byte components, with the subsequent conversion format applied to each.
- vh or hv consumes one argument and modifies an existing short integer conversion, resulting in vector signed short or vector unsigned short for output conversions or vector signed short * or vector unsigned

short * for input conversions. The data is treated as a series of eight 2-byte components, with the subsequent conversion format applied to each.

- v consumes one argument and modifies a 1-byte integer, 1-byte character, or 4-byte floating-point conversion. If the conversion is a floating-point conversion, the result is vector float for output conversion or vector float * for input conversion. The data is treated as a series of four 4-byte floating-point components with the subsequent conversion format applied to each. If the conversion is an integer or character conversion, the result is either vector signed char, vector unsigned char, or vector bool char for output conversion, or vector signed char * or vector unsigned char * for input conversions. The data is treated as a series of sixteen 1-byte components, with the subsequent conversion format applied to each.
- vv consumes one argument and modifies an 8-byte floating-point conversion. If the conversion is a floating-point conversion, the result is vector double for output conversion or vector double * for input conversion. The data is treated as a series of two 8-byte floating-point components with the subsequent conversion format applied to each. Integer and byte conversions are not defined for the vv modifier.

Note: As new vector types are defined, new format codes should be defined to support scanf and printf of those types.

Any conversion format that can be applied to the singular form of a vector-data type can be used with a vector form. The %d, %x, %X, %u, %i, and %o integer conversions can be applied with the %lv, %vl, %hv, %vh, and %v vector-length qualifiers. The %c character conversion can be applied with the %v vector length qualifier. The %a, %A, %e, %E, %f, %F, %g, and %G float conversions can be applied with the %v vector length qualifier.

For input conversions, an optional separator character can be specified excluding white space preceding the separator. If no separator is specified, the default separator is a space including white space characters preceding the separator, unless the conversion is c. Then, the default conversion is null.

For output conversions, an optional separator character can be specified immediately preceding the vector size conversion. If no separator is specified, the default separator is a space unless the conversion is c. Then, the default separator is null.

Appendix A. Predefined Functions for Vector Programming

So that programmers can access the vector facilities provided by the Power ISA, ABI-compliant environments should provide the following vector functions and predicates. While functions are specified in this document in conjunction with C/C++ language syntax, it is recommended that other environments follow the proposed vector built-in naming and function set, based on the vector types provided by the respective language.

If signed or unsigned is omitted, the signedness of the vector type is the default signedness of the base type. The default varies depending on the operating system, so a portable program should always specify the signedness.

Arguments that are documented as const int require literal integral values within the vector built-in invocation.

In addition to the hardware-specific vector built-in functions, implementations are expected to provide the following interfaces:

Built-In Function	Implementation Notes
vec_extract	vec_extract (v, 3) is equivalent to v[3].
vec_insert	vec_insert (v, 3, x) is equivalent to v[3] = x.

Environments may provide the optional built-in vector functions listed in *Table A-1* to adjust for endian behavior by reversing the order of elements (reve) and bytes within elements (revb).

Table A-1 Optional Built-In Functions

vector signed char vec_revb(vector signed char);
vector unsigned char vec_revb(vector unsigned char);
vector signed short vec_revb (vector signed short);
vector unsigned short vec_revb (vector unsigned short);
vector signed int vec_revb (vector signed int);
vector unsigned int vec_revb (vector unsigned int);
vector signed long long vec_revb (vector signed long long);
vector unsigned long long vec_revb (vector unsigned long long);
vector signed char vec_reve(vector signed char);
vector unsigned char vec_reve(vector unsigned char);
vector signed short vec_reve (vector signed short);
vector unsigned short vec_reve (vector unsigned short);
vector signed int vec_reve (vector signed int);
vector unsigned int vec_reve (vector unsigned int);
vector signed long long vec_reve (vector signed long long);
vector unsigned long long vec_reve (vector unsigned long long);
vector signed char vec_reve (vector signed float);
vector double vec_reve (vector double);

A.1 Vector Built-In Functions

Table A-2. Vector Built-In Function (with Prototype) (Page 1 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_ABS(ARG1)	Purpose: Returns a vector containing the absolute values of the contents of the given vector. Result value: The value of each element of the result is the absolute value of the corresponding element of ARG1. For integer vectors, the arithmetic is modular.
	vector signed char vec_abs (vector signed char);
	vector signed int vec_abs (vector signed int);
	vector signed long long vec_abs (vector signed long long);
	vector signed short vec_abs (vector signed short);
	vector double vec_abs (vector double);
	vector float vec_abs (vector float);
VEC_ABSS(ARG1)	Purpose: Returns a vector containing the saturated absolute values of the contents of the given vector. Result value: The value of each element of the result is the saturated absolute value of the corresponding element of ARG1.
	vector signed char vec_abss (vector signed char);
	vector signed int vec_abss (vector signed int);
	vector signed short vec_abss (vector signed short);
VEC_ADD(ARG1, ARG2)	Purpose: Returns a vector containing the sums of each set of corresponding elements of the given vectors. **Note:** For vector signed long long, vector signed long vectors, this function emulates the operation. Result value: The value of each element of the result is the sum of the corresponding elements of ARG1 and ARG2. For integer vectors and unsigned vectors, the arithmetic is modular.
	vector signed char vec_add (vector bool char, vector signed char);
	vector signed char vec_add (vector signed char, vector bool char);
	vector signed char vec_add (vector signed char, vector signed char);
	vector unsigned char vec_add (vector bool char, vector unsigned char);
	vector unsigned char vec_add (vector unsigned char, vector bool char);
	vector unsigned char vec_add (vector unsigned char, vector unsigned char);
	vector signed int vec_add (vector bool int, vector signed int);
	vector signed int vec_add (vector signed int, vector bool int);
	vector signed int vec_add (vector signed int, vector signed int);
	vector unsigned int vec_add (vector bool int, vector unsigned int);
	vector unsigned int vec_add (vector unsigned int, vector bool int);
	vector unsigned int vec_add (vector unsigned int, vector unsigned int);
	vector signed long long vec_add (vector signed long long, vector signed long long);
	vector unsigned long long vec_add (vector unsigned long long, vector unsigned long long);

Table A-2. Vector Built-In Function (with Prototype) (Page 2 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector signed short vec_add (vector bool short, vector signed short);
	vector signed short vec_add (vector signed short, vector bool short);
	vector signed short vec_add (vector signed short, vector signed short);
	vector unsigned short vec_add (vector bool short, vector unsigned short);
	vector unsigned short vec_add (vector unsigned short, vector bool short);
	vector unsigned short vec_add (vector unsigned short, vector unsigned short);
	vector double vec_add (vector double, vector double);
	vector float vec_add (vector float, vector float);
VEC_ADDC(ARG1, ARG2)	Purpose: Returns a vector containing the carry produced by adding each set of corresponding elements of the given vectors. Result value: The value of each element of the result is the carry produced by adding the corresponding elements of ARG1 and ARG2 (1 if there is a carry, 0 otherwise).
	vector unsigned int vec_addc (vector unsigned int, vector unsigned int);
VEC_ADDS(ARG1, ARG2)	Purpose: Returns a vector containing the saturated sums of each set of corresponding elements of the given vectors. Result value: The value of each element of the result is the saturated sum of the corresponding elements of ARG1 and ARG2.
	vector signed char vec_adds (vector bool char, vector signed char);
	vector signed char vec_adds (vector signed char, vector bool char);
	vector signed char vec_adds (vector signed char, vector signed char);
	vector unsigned char vec_adds (vector bool char, vector unsigned char);
	vector unsigned char vec_adds (vector unsigned char, vector bool char);
	vector unsigned char vec_adds (vector unsigned char, vector unsigned char);
	vector signed int vec_adds (vector bool int, vector signed int);
	vector signed int vec_adds (vector signed int, vector bool int);
	vector signed int vec_adds (vector signed int, vector signed int);
	vector unsigned int vec_adds (vector bool int, vector unsigned int);
	vector unsigned int vec_adds (vector unsigned int, vector bool int);
	vector unsigned int vec_adds (vector unsigned int, vector unsigned int);
	vector signed short vec_adds (vector bool short, vector signed short);
	vector signed short vec_adds (vector signed short, vector bool short);
	vector signed short vec_adds (vector signed short, vector signed short);
	vector unsigned short vec_adds (vector bool short, vector unsigned short);
	vector unsigned short vec_adds (vector unsigned short, vector bool short);
	vector unsigned short vec_adds (vector unsigned short, vector unsigned short);

Table A-2. Vector Built-In Function (with Prototype) (Page 3 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_AND(ARG1, ARG2)	Purpose: Performs a bitwise AND of the given vectors. Result value: The result is the bitwise AND of ARG1 and ARG2.
	vector bool char vec_and (vector bool char, vector bool char);
	vector signed char vec_and (vector bool char, vector signed char);
	vector signed char vec_and (vector signed char, vector bool char);
	vector signed char vec_and (vector signed char, vector signed char);
	vector unsigned char vec_and (vector bool char, vector unsigned char);
	vector unsigned char vec_and (vector unsigned char, vector bool char);
	vector unsigned char vec_and (vector unsigned char, vector unsigned char);
	vector bool int vec_and (vector bool int, vector bool int);
	vector signed int vec_and (vector bool int, vector signed int);
	vector signed int vec_and (vector signed int, vector bool int);
	vector signed int vec_and (vector signed int, vector signed int);
	vector unsigned int vec_and (vector bool int, vector unsigned int);
	vector unsigned int vec_and (vector unsigned int, vector bool int);
	vector unsigned int vec_and (vector unsigned int, vector unsigned int);
	vector bool short vec_and (vector bool short, vector bool short);
	vector signed short vec_and (vector bool short, vector signed short);
	vector signed short vec_and (vector signed short, vector bool short);
	vector signed short vec_and (vector signed short, vector signed short);
Phased in.[a]	vector signed long long vec_and (vector signed long long, vector signed long long)
Phased in.[a]	vector signed long long vec_and (vector signed long long, vector bool long long)
Phased in.[a]	vector signed long long vec_and (vector bool long long, vector signed long long)
	vector unsigned short vec_and (vector bool short, vector unsigned short);
	vector unsigned short vec_and (vector unsigned short, vector bool short);
	vector unsigned short vec_and (vector unsigned short, vector unsigned short);
Phased in.[a]	vector unsigned long long vec_and (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_and (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_and (vector bool long long, vector unsigned long long)
	vector double vec_and (vector bool long long, vector double);
	vector double vec_and (vector double, vector bool long long);
	vector double vec_and (vector double, vector double);
	vector float vec_and (vector bool int, vector float);
	vector float vec_and (vector float, vector bool int);
	vector float vec_and (vector float, vector float);

Table A-2. Vector Built-In Function (with Prototype) (Page 4 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_ANDC(ARG1, ARG2)	Purpose: Performs a bitwise AND of the first argument and the bitwise complement of the second argument. Result value: The result is the bitwise AND of ARG1 with the bitwise complement of ARG2.
	vector bool char vec_andc (vector bool char, vector bool char);
	vector signed char vec_andc (vector bool char, vector signed char);
	vector signed char vec_andc (vector signed char, vector bool char);
	vector signed char vec_andc (vector signed char, vector signed char);
	vector unsigned char vec_andc (vector bool char, vector unsigned char);
	vector unsigned char vec_andc (vector unsigned char, vector bool char);
	vector unsigned char vec_andc (vector unsigned char, vector unsigned char);
	vector bool int vec_andc (vector bool int, vector bool int);
	vector signed int vec_andc (vector bool int, vector signed int);
	vector signed int vec_andc (vector signed int, vector bool int);
	vector signed int vec_andc (vector signed int, vector signed int);
Phased in.[a]	vector signed long long vec_andc (vector signed long long, vector signed long long)
Phased in.[a]	vector signed long long vec_andc (vector signed long long, vector bool long long)
Phased in.[a]	vector signed long long vec_andc (vector bool long long, vector signed long long)
	vector unsigned int vec_andc (vector bool int, vector unsigned int);
	vector unsigned int vec_andc (vector unsigned int, vector bool int);
	vector unsigned int vec_andc (vector unsigned int, vector unsigned int);
Phased in.[a]	vector unsigned long long vec_andc (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_andc (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_andc (vector bool long long, vector unsigned long long)
	vector bool short vec_andc (vector bool short, vector bool short);
	vector signed short vec_andc (vector bool short, vector signed short);
	vector signed short vec_andc (vector signed short, vector bool short);
	vector signed short vec_andc (vector signed short, vector signed short);
	vector unsigned short vec_andc (vector bool short, vector unsigned short);
	vector unsigned short vec_andc (vector unsigned short, vector bool short);
	vector unsigned short vec_andc (vector unsigned short, vector unsigned short);
	vector double vec_andc (vector bool long long, vector double);
	vector double vec_andc (vector double, vector bool long long);
	vector double vec_andc (vector double, vector double);
	vector float vec_andc (vector bool int, vector float);
	vector float vec_andc (vector float, vector bool int);
	vector float vec_andc (vector float, vector float);

Table A-2. Vector Built-In Function (with Prototype) (Page 5 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_AVG(ARG1, ARG2)	Purpose: Returns a vector containing the average of each set of corresponding elements of the given vectors. Result value: The value of each element of the result is the average of the values of the corresponding elements of ARG1 and ARG2.
	vector signed char vec_avg (vector signed char, vector signed char);
	vector unsigned char vec_avg (vector unsigned char, vector unsigned char);
	vector signed int vec_avg (vector signed int, vector signed int);
	vector unsigned int vec_avg (vector unsigned int, vector unsigned int);
	vector signed short vec_avg (vector signed short, vector signed short);
	vector unsigned short vec_avg (vector unsigned short, vector unsigned short);
VEC_CEIL(ARG1)	Purpose: Returns a vector containing the smallest representable floating-point integral values greater than or equal to the values of the corresponding elements of the given vector. **Note:** VEC_CEIL is another name for VEC_ROUNDP. Result value: Each element of the result contains the smallest representable floating-point integral value greater than or equal to the value of the corresponding element of ARG1.
	vector double vec_ceil (vector double);
	vector float vec_ceil (vector float);
VEC_CMPB(ARG1, ARG2)	Purpose: Performs a bounds comparison of each set of corresponding elements of the given vectors. Result value: Each element of the result has the value 0 if the value of the corresponding element of ARG1 is less than or equal to the value of the corresponding element of ARG2 and greater than or equal to the negative of the value of the corresponding element of ARG2. Otherwise: • If an element of ARG2 is greater than or equal to zero, then the value of the corresponding element of the result is 0 if the absolute value of the corresponding element of ARG1 is equal to the value of the corresponding element of ARG2, negative if it is greater than the value of the corresponding element of ARG2, and positive if it is less than the value of the corresponding element of ARG2. • If an element of ARG2 is less than zero, then the value of the element of the result is positive if the value of the corresponding element of ARG1 is less than or equal to the value of the element of ARG2. Otherwise. it is negative.
	vector signed int vec_cmpb (vector float, vector float);
VEC_CMPEQ(ARG1, ARG2)	Purpose: Returns a vector containing the results of comparing each set of corresponding elements of the given vectors for equality. Result value: For each element of the result, the value of each bit is 1 if the corresponding elements of ARG1 and ARG2 are equal. Otherwise, the value of each bit is 0.
	vector bool char vec_cmpeq (vector signed char, vector signed char);
	vector bool char vec_cmpeq (vector unsigned char, vector unsigned char);
	vector bool int vec_cmpeq (vector float, vector float);
	vector bool int vec_cmpeq (vector signed int, vector signed int);
	vector bool int vec_cmpeq (vector unsigned int, vector unsigned int);
	vector bool long vec_cmpeq (vector double, vector double);

Table A-2. Vector Built-In Function (with Prototype) (Page 6 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector bool short vec_cmpeq (vector signed short, vector signed short);
	vector bool short vec_cmpeq (vector unsigned short, vector unsigned short);
	vector bool long long vec_cmpeq (vector signed long long, vector signed long long)
	vector bool long long vec_cmpeq (vector unsigned long long, vector unsigned long long)
VEC_CMPGE(ARG1, ARG2)	Purpose: Returns a vector containing the results of a greater-than-or-equal-to comparison between each set of corresponding elements of the given vectors. Result value: For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is greater than or equal to the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.
	vector bool int vec_cmpge (vector float, vector float);
	vector bool long vec_cmpge (vector double, vector double);
	vector bool long long vec_cmpge (vector signed long long, vector signed long long)
	vector bool long long vec_cmpge (vector unsigned long long, vector unsigned long long)
VEC_CMPGT(ARG1, ARG2)	Purpose: Returns a vector containing the results of a greater-than comparison between each set of corresponding elements of the given vectors. Result value: For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is greater than the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.
	vector bool char vec_cmpgt (vector signed char, vector signed char);
	vector bool char vec_cmpgt (vector unsigned char, vector unsigned char);
	vector bool int vec_cmpgt (vector float, vector float);
	vector bool int vec_cmpgt (vector signed int, vector signed int);
	vector bool int vec_cmpgt (vector unsigned int, vector unsigned int);
	vector bool long vec_cmpgt (vector double, vector double);
	vector bool short vec_cmpgt (vector signed short, vector signed short);
	vector bool short vec_cmpgt (vector unsigned short, vector unsigned short);
	vector bool long long vec_cmpgt (vector signed long long, vector signed long long)
	vector bool long long vec_cmpgt (vector unsigned long long, vector unsigned long long)
VEC_CMPLE(ARG1, ARG2)	Purpose: Returns a vector containing the results of a less-than-or-equal-to comparison between each set of corresponding elements of the given vectors. Result value: For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is less than or equal to the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.
	vector bool int vec_cmple (vector float, vector float);
	vector bool long vec_cmple (vector double, vector double);
	vector bool long long vec_cmple (vector signed long long, vector signed long long)
	vector bool long long vec_cmple (vector unsigned long long, vector unsigned long long)

Table A-2. Vector Built-In Function (with Prototype) (Page 7 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_CMPLT(ARG1, ARG2)	Purpose: Returns a vector containing the results of a less-than comparison between each set of corresponding elements of the given vectors. Result value: For each element of the result, the value of each bit is 1 if the value of the corresponding element of ARG1 is less than the value of the corresponding element of ARG2. Otherwise, the value of each bit is 0.
	vector bool char vec_cmplt (vector signed char, vector signed char);
	vector bool char vec_cmplt (vector unsigned char, vector unsigned char);
	vector bool int vec_cmplt (vector float, vector float);
	vector bool int vec_cmplt (vector signed int, vector signed int);
	vector bool int vec_cmplt (vector unsigned int, vector unsigned int);
	vector bool long vec_cmplt (vector double, vector double);
	vector bool short vec_cmplt (vector signed short, vector signed short);
	vector bool short vec_cmplt (vector unsigned short, vector unsigned short);
	vector bool long long vec_cmplt (vector signed long long, vector signed long long)
	vector bool long long vec_cmplt (vector unsigned long long, vector unsigned long long)
VEC_CNTLZ(ARG1)	Purpose: Returns a vector containing the number of most-significant bits equal to 0 of each corresponding element of the given vector. Result value: The value of each element of the result is the sum of the corresponding single-precision floating-point elements of ARG1 and ARG2.
Phased in.[a]	vector signed char vec_cntlz (vector signed char);
Phased in.[a]	vector unsigned char vec_cntlz (vector unsigned char);
Phased in.[a]	vector unsigned int vec_cntlz (vector unsigned int);
Phased in.[a]	vector signed long long vec_cntlz (vector signed long long);
Phased in.[a]	vector unsigned long long vec_cntlz (vector unsigned long long);
Phased in.[a]	vector unsigned short vec_cntlz (vector unsigned short);
Phased in.[a]	vector int vec_cntlz (vector int);
Phased in.[a]	vector short vec_cntlz (vector short);
VEC_CPSGN(ARG1,ARG2)	Purpose: Returns a vector by copying the sign of the elements in vector ARG1 to the sign of the corresponding elements in vector ARG2. Result value: For each element of the result, copies the sign of the corresponding element in vector ARG1 to the sign of the corresponding element in vector ARG2.
Phased in.[a]	vector float vec_cpsgn (vector float, vector float);
Phased in.[a]	vector double vec_cpsgn(vector double, vector double);

Table A-2. Vector Built-In Function (with Prototype) (Page 8 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_CTF(ARG1, ARG2)	Purpose: Converts an integer vector into a vector float. **Note:** The second and fourth elements of the result vector are undefined when ARG1 is an vector signed long long, vector signed long vector. Result value: The value of each element of the result is the closest floating-point estimate of the value of the corresponding element of ARG1 divided by 2 to the power of ARG2.
	vector float vec_ctf (vector signed int, const int);
	vector float vec_ctf (vector unsigned int, const int);
Phased in.[a]	vector float vec_ctf (vector signed long long, const int)
Phased in.[a]	vector float vec_ctf (vector unsigned long long, const int)
VEC_CTS(ARG1, ARG2)	Purpose: Converts a real vector into a vector signed int. Result value: The value of each element of the result is the saturated value obtained by multiplying the corresponding element of ARG1 by 2 to the power of ARG2. **Note:** The second and fourth elements of the result vector are undefined when ARG1 is a vector double.
	vector signed int vec_cts (vector float, const int);
Phased in.[a]	vector signed int vec_cts (vector double, const int)
VEC_CTU(ARG1,ARG2)	Purpose: Converts a real vector into a vector unsigned int. Result value: The value of each element of the result is the saturated value obtained by multiplying the corresponding element of ARG1 by 2 to the power of ARG2. **Note:** The second and fourth elements of the result vector are undefined when ARG1 is a vector double.
	vector unsigned int vec_ctu (vector float, const int);
Phased in.[a]	vector unsigned int vec_ctu (vector double, const int)
VEC_DIV(ARG1, ARG2)	Purpose: Divides the elements in ARG1 by the corresponding elements in ARG2 and then assigns the result to corresponding elements in the result vector. This function emulates the operation on integer vectors. Result value: The value of each element of the result is obtained by dividing the corresponding element of ARG1 by the corresponding element of ARG2.
	vector double vec_div (vector double, vector double);
	vector float vec_div (vector float, vector float);
	vector unsigned char vec_div(vector unsigned char, vector unsigned char);
	vector signed char vec_div(vector signed char, vector signed char);
	vector unsigned short vec_div(vector unsigned short, vector unsigned short);
	vector signed short vec_div(vector signed short, vector signed short);
	vector unsigned int vec_div(vector unsigned int, vector unsigned int);
	vector signed int vec_div(vector signed int, vector signed int);
Phased in.[a]	vector unsigned long long vec_div(vector unsigned long long, vector unsigned long long);

Table A-2. Vector Built-In Function (with Prototype) (Page 9 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
Phased in.[a]	vector signed long long vec_div(vector signed long long, vector signed long long);
VEC_EQV(ARG1, ARG2)	Purpose: Performs a bitwise XNOR of the given vectors. Result value: The result is the bitwise XNOR of ARG1 and ARG2.
	vector signed char vec_eqv (vector bool signed char, vector signed char);
	vector signed char vec_eqv (vector signed char, vector bool signed char);
	vector signed char vec_eqv (vector signed char, vector signed char);
	vector unsigned char vec_eqv (vector bool unsigned char, vector unsigned char);
	vector unsigned char vec_eqv (vector unsigned char, vector bool unsigned char);
	vector unsigned char vec_eqv (vector unsigned char, vector unsigned char);
	vector unsigned int vec_eqv (vector bool unsigned int, vector unsigned int);
	vector unsigned int vec_eqv (vector unsigned int, vector bool unsigned int);
	vector unsigned int vec_eqv (vector unsigned int, vector unsigned int);
	vector signed long long vec_eqv (vector bool long long, vector signed long long);
	vector signed long long vec_eqv (vector signed long long, vector bool long long);
	vector signed long long vec_eqv (vector signed long long, vector signed long long);
	vector unsigned long long vec_eqv (vector bool long long, vector unsigned long long);
	vector unsigned long long vec_eqv (vector unsigned long long, vector bool long long);
	vector unsigned long long vec_eqv (vector unsigned long long, vector unsigned long long);
	vector unsigned short vec_eqv (vector bool unsigned short, vector unsigned short);
	vector unsigned short vec_eqv (vector unsigned short, vector bool unsigned short);
	vector unsigned short vec_eqv (vector unsigned short, vector unsigned short);
	vector int vec_eqv (vector bool int, vector int);
	vector int vec_eqv (vector int, vector bool int);
	vector int vec_eqv (vector int, vector int);
	vector short vec_eqv (vector bool short, vector short);
	vector short vec_eqv (vector short, vector bool short);
	vector short vec_eqv (vector short, vector short);
	vector float vec_eqv(vector float, vector float);
	vector float vec_eqv(vector bool int, vector float);
	vector float vec_eqv(vector float, vector bool int);
	vector double vec_eqv(vector double, vector double);
	vector double vec_eqv(vector bool long long, vector double);
	vector double vec_eqv(vector double, vector bool long long);

Table A-2. Vector Built-In Function (with Prototype) (Page 10 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_EXPTE(ARG1)	Purpose: Returns a vector containing estimates of 2 raised to the power of the corresponding elements of the given vector. Result value: Each element of the result contains the estimated value of 2 raised to the power of the corresponding element of ARG1.
	vector float vec_expte (vector float);
VEC_FLOOR(ARG1)	Purpose: Returns a vector containing the largest representable floating-point integral values less than or equal to the values of the corresponding elements of the given vector. **Note:** VEC_FLOOR is another name for VEC_ROUNDM. Result value: Each element of the result contains the largest representable floating-point integral value less than or equal to the value of the corresponding element of ARG1.
	vector double vec_floor (vector double);
	vector float vec_floor (vector float);
VEC_LOGE(ARG1)	Purpose: Returns a vector containing estimates of the base-2 logarithms of the corresponding elements of the given vector. Result value: Each element of the result contains the estimated value of the base-2 logarithm of the corresponding element of ARG1.
	vector float vec_loge (vector float);
VEC_MADD(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a fused multiply-add operation for each corresponding set of elements of the given vectors. Result value: The value of each element of the result is the product of the values of the corresponding elements of ARG1 and ARG2, added to the value of the corresponding element of ARG3, and then multiplied by -1.0.
	vector double vec_madd (vector double, vector double, vector double);
	vector float vec_madd (vector float, vector float, vector float);
VEC_MADDS(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a saturated multiply-high-and-add operation for each corresponding set of elements of the given vectors. Result value: For each element of the result, the value is produced in the following way: The values of the corresponding elements of ARG1 and ARG2 are multiplied. The value of the 17 most-significant bits of this product is then added, using 16-bit-saturated addition, to the value of the corresponding element of ARG3.
	vector signed short vec_madds (vector signed short, vector signed short, vector signed short);
VEC_MAX(ARG1, ARG2)	Purpose Returns a vector containing the maximum value from each set of corresponding elements of the given vectors. Result value The value of each element of the result is the maximum of the values of the corresponding elements of ARG1 and ARG2.
	vector signed char vec_max (vector bool char, vector signed char);
	vector signed char vec_max (vector signed char, vector bool char);

Table A-2. Vector Built-In Function (with Prototype) (Page 11 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector signed char vec_max (vector signed char, vector signed char);
	vector unsigned char vec_max (vector bool char, vector unsigned char);
	vector unsigned char vec_max (vector unsigned char, vector bool char);
	vector unsigned char vec_max (vector unsigned char, vector unsigned char);
	vector signed int vec_max (vector bool int, vector signed int);
	vector signed int vec_max (vector signed int, vector bool int);
	vector signed int vec_max (vector signed int, vector signed int);
	vector unsigned int vec_max (vector bool int, vector unsigned int);
	vector unsigned int vec_max (vector unsigned int, vector bool int);
	vector unsigned int vec_max (vector unsigned int, vector unsigned int);
	vector signed long long vec_max (vector signed long long, vector signed long long);
	vector unsigned long long vec_max (vector unsigned long long, vector unsigned long long);
	vector signed short vec_max (vector bool short, vector signed short);
	vector signed short vec_max (vector signed short, vector bool short);
	vector signed short vec_max (vector signed short, vector signed short);
	vector unsigned short vec_max (vector bool short, vector unsigned short);
	vector unsigned short vec_max (vector unsigned short, vector bool short);
	vector unsigned short vec_max (vector unsigned short, vector unsigned short);
	vector double vec_max (vector double, vector double);
	vector float vec_max (vector float, vector float);
VEC_MERGEE(ARG1, ARG2)	Purpose: Merges the even-numbered values from the two vectors. Result value: Assume that the N elements of each vector are numbered beginning with 0. The first N/2 elements of the result are taken, in order, from the even-numbered elements of ARG1. The last N/2 elements of the result are taken, in order, from the even-numbered elements of ARG2.
Phased in.[a]	vector bool int vec_mergee (vector bool int, vector bool int);
Phased in.[a]	vector signed int vec_mergee (vector signed int, vector signed int);
Phased in.[a]	vector unsigned int vec_mergee (vector unsigned int, vector unsigned int);
VEC_MERGEH(ARG1, ARG2)	Purpose: Merges the most-significant halves of two vectors. Result value: Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the most-significant 8 bytes of ARG1. The odd-numbered elements of the result are taken, in order, from the elements in the most-significant 8 bytes of ARG2.
	vector bool char vec_mergeh (vector bool char, vector bool char);
	vector signed char vec_mergeh (vector signed char, vector signed char);
	vector unsigned char vec_mergeh (vector unsigned char, vector unsigned char);
	vector bool int vec_mergeh (vector bool int, vector bool int);
	vector signed int vec_mergeh (vector signed int, vector signed int);

Table A-2. Vector Built-In Function (with Prototype) (Page 12 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector unsigned int vec_mergeh (vector unsigned int, vector unsigned int);
	vector bool short vec_mergeh (vector bool short, vector bool short);
	vector signed short vec_mergeh (vector signed short, vector signed short);
	vector unsigned short vec_mergeh (vector unsigned short, vector unsigned short);
	vector float vec_mergeh (vector float, vector float);
	vector pixel vec_mergeh (vector pixel, vector pixel);
	vector signed long long vec_mergeh (vector signed long long, vector signed long long)
	vector signed long long vec_mergeh (vector signed long long, vector bool long long)
	vector signed long long vec_mergeh (vector bool long long, vector signed long long)
Phased in.[a]	vector unsigned long long vec_mergeh (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_mergeh (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_mergeh (vector bool long long, vector unsigned long long)
	vector double vec_mergeh (vector bool long long, vector double);
	vector double vec_mergeh (vector double, vector bool long long);
	vector double vec_mergeh (vector double, vector double);
VEC_MERGEL(ARG1, ARG2)	Purpose: Merges the least-significant halves of two vectors. Result value: Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the least-significant 8 bytes of ARG1. The odd-numbered elements of the result are taken, in order, from the elements in the least-significant 8 bytes of ARG2.
	vector bool char vec_mergel (vector bool char, vector bool char);
	vector signed char vec_mergel (vector signed char, vector signed char);
	vector unsigned char vec_mergel (vector unsigned char, vector unsigned char);
	vector bool int vec_mergel (vector bool int, vector bool int);
	vector signed int vec_mergel (vector signed int, vector signed int);
	vector unsigned int vec_mergel (vector unsigned int, vector unsigned int);
	vector bool short vec_mergel (vector bool short, vector bool short);
	vector signed short vec_mergel (vector signed short, vector signed short);
	vector unsigned short vec_mergel (vector unsigned short, vector unsigned short);
	vector float vec_mergel (vector float, vector float);
	vector pixel vec_mergel (vector pixel, vector pixel);
	vector signed long long vec_mergel (vector signed long long, vector signed long long)
	vector signed long long vec_mergel (vector signed long long, vector bool long long)
	vector signed long long vec_mergel (vector bool long long, vector signed long long)
Phased in.[a]	vector unsigned long long vec_mergel (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_mergel (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_mergel (vector bool long long, vector unsigned long long)

Table A-2. Vector Built-In Function (with Prototype) (Page 13 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector double vec_mergel (vector bool long long, vector double);
	vector double vec_mergel (vector double, vector bool long long);
	vector double vec_mergel (vector double, vector double);
VEC_MERGEO(ARG1, ARG2)	Purpose: Merges the odd-numbered halves of two vectors. Result value: Assume that the N elements of each vector are numbered beginning with 0. The first N/2 elements of the result are taken, in order, from the odd-numbered elements of ARG1. The last N/2 elements of the result are taken, in order, from the odd-numbered elements of ARG2.
Phased in.[a]	vector bool int vec_mergeo (vector bool int, vector bool int);
Phased in.[a]	vector signed int vec_mergeo (vector signed int, vector signed int);
Phased in.[a]	vector unsigned int vec_mergeo (vector unsigned int, vector unsigned int);
VEC_MIN(ARG1, ARG2)	Purpose: Returns a vector containing the minimum value from each set of corresponding elements of the given vectors. Result value: The value of each element of the result is the minimum of the values of the corresponding elements of ARG1 and ARG2.
	vector signed char vec_min (vector bool char, vector signed char);
	vector signed char vec_min (vector signed char, vector bool char);
	vector signed char vec_min (vector signed char, vector signed char);
	vector unsigned char vec_min (vector bool char, vector unsigned char);
	vector unsigned char vec_min (vector unsigned char, vector bool char);
	vector unsigned char vec_min (vector unsigned char, vector unsigned char);
	vector signed int vec_min (vector bool int, vector signed int);
	vector signed int vec_min (vector signed int, vector bool int);
	vector signed int vec_min (vector signed int, vector signed int);
	vector unsigned int vec_min (vector bool int, vector unsigned int);
	vector unsigned int vec_min (vector unsigned int, vector bool int);
	vector unsigned int vec_min (vector unsigned int, vector unsigned int);
	vector signed long long vec_min (vector signed long long, vector signed long long);
	vector unsigned long long vec_min (vector unsigned long long, vector unsigned long long);
	vector signed short vec_min (vector bool short, vector signed short);
	vector signed short vec_min (vector signed short, vector bool short);
	vector signed short vec_min (vector signed short, vector signed short);
	vector unsigned short vec_min (vector bool short, vector unsigned short);
	vector unsigned short vec_min (vector unsigned short, vector bool short);
	vector unsigned short vec_min (vector unsigned short, vector unsigned short);
	vector double vec_min (vector double, vector double);
	vector float vec_min (vector float, vector float);

Table A-2. Vector Built-In Function (with Prototype) (Page 14 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_MLADD(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a saturated multiply-low-and-add operation for each corresponding set of elements of the given vectors. Result value: The value of each element of the result is the value of the least-significant 16 bits of the product of the values of the corresponding elements of ARG1 and ARG2, added to the value of the corresponding element of ARG3. The addition is performed using modular arithmetic.
	vector signed short vec_mladd (vector signed short, vector signed short, vector signed short);
	vector signed short vec_mladd (vector signed short, vector unsigned short, vector unsigned short);
	vector signed short vec_mladd (vector unsigned short, vector signed short, vector signed short);
	vector unsigned short vec_mladd (vector unsigned short, vector unsigned short, vector unsigned short);
VEC_MRADDS(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a saturated multiply-high-round-and-add operation for each corresponding set of elements of the given vectors. Result value: For each element of the result, the value is produced in the following way: The values of the corresponding elements of ARG1 and ARG2 are multiplied and rounded such that the 15 least-significant bits are 0. The value of the 17 most-significant bits of this rounded product is then added, using 16-bit-saturated addition, to the value of the corresponding element of ARG3.
	vector signed short vec_mradds (vector signed short, vector signed short, vector signed short);
VEC_MSUB(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a multiply-subtract operation using the given vectors. Result value: This function multiplies each element in ARG1 by the corresponding element in ARG2 and then subtracts the corresponding element in ARG3 from the result.
	vector double vec_msub (vector double, vector double, vector double);
	vector float vec_msub (vector float, vector float, vector float);
VEC_MSUM(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a multiply-sum operation using the given vectors. Result value: Assume that the elements of each vector are numbered beginning with 0. If ARG1 is a vector signed char or a vector unsigned char vector, then let m be 4. Otherwise, let m be 2. For each element n of the result vector, the value is obtained in the following way: For p = mn to mn+m-1, multiply element p of ARG1 by element p of ARG2. Add the sum of these products to element n of ARG3. All additions are performed using 32-bit modular arithmetic.
	vector signed int vec_msum (vector signed char, vector unsigned char, vector signed int);
	vector signed int vec_msum (vector signed short, vector signed short, vector signed int);
	vector unsigned int vec_msum (vector unsigned char, vector unsigned char, vector unsigned int);
	vector unsigned int vec_msum (vector unsigned short, vector unsigned short, vector unsigned int);
VEC_MSUMS(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a saturated multiply-sum operation using the given vectors. Result value: Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is obtained in the following way: For p = 2n to 2n+1, multiply element p of ARG1 by element p of ARG2. Add the sum of these products to element n of ARG3. All additions are performed using 32-bit saturated arithmetic.

Table A-2. Vector Built-In Function (with Prototype) (Page 15 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector signed int vec_msums (vector signed short, vector signed short, vector signed int);
	vector unsigned int vec_msums (vector unsigned short, vector unsigned short, vector unsigned int);
VEC_MUL(ARG1, ARG2)	Purpose: Returns a vector containing the results of performing a multiply operation using the given vectors. This function emulates the operation on integer vectors. Result value: This function multiplies corresponding elements in the given vectors and then assigns the result to corresponding elements in the result vector.
	vector double vec_mul (vector double, vector double);
	vector float vec_mul (vector float, vector float);
	vector unsigned char vec_mul(vector unsigned char, vector unsigned char);
	vector signed char vec_mul(vector signed char, vector signed char);
	vector unsigned short vec_mul(vector unsigned short, vector unsigned short);
	vector signed short vec_mul(vector signed short, vector signed short);
	vector unsigned int vec_mul(vector unsigned int, vector unsigned int);
	vector signed int vec_mul(vector signed int, vector signed int);
Phased in.[a]	vector unsigned long long vec_mul(vector unsigned long long, vector unsigned long long);
Phased in.[a]	vector signed long long vec_mul(vector signed long long, vector signed long long);
VEC_MULE(ARG1, ARG2)	Purpose: Returns a vector containing the results of multiplying every second set of the corresponding elements of the given vectors, beginning with the first element. Result value: Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element 2n of ARG1 and the value of element 2n of ARG2.
	vector signed int vec_mule (vector signed short, vector signed short);
	vector unsigned int vec_mule (vector unsigned short, vector unsigned short);
	vector signed short vec_mule (vector signed char, vector signed char);
	vector unsigned short vec_mule (vector unsigned char, vector unsigned char);
VEC_MULO(ARG1, ARG2)	Purpose: Returns a vector containing the results of multiplying every second set of corresponding elements of the given vectors, beginning with the second element. Result value: Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element 2n+1 of ARG1 and the value of element 2n+1 of ARG2.
	vector signed int vec_mulo (vector signed short, vector signed short);
	vector unsigned int vec_mulo (vector unsigned short, vector unsigned short);
	vector signed short vec_mulo (vector signed char, vector signed char);
	vector unsigned short vec_mulo (vector unsigned char, vector unsigned char);
VEC_NAND(ARG1, ARG2)	Purpose: Performs a bitwise NAND of the given vectors. Result value: The result is the bitwise NAND of ARG1 and ARG2.

Table A-2. Vector Built-In Function (with Prototype) (Page 16 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector signed char vec_nand (vector bool signed char, vector signed char);
	vector signed char vec_nand (vector signed char, vector bool signed char);
	vector signed char vec_nand (vector signed char, vector signed char);
	vector unsigned char vec_nand (vector bool unsigned char, vector unsigned char);
	vector unsigned char vec_nand (vector unsigned char, vector bool unsigned char);
	vector unsigned char vec_nand (vector unsigned char, vector unsigned char);
	vector unsigned int vec_nand (vector bool unsigned int, vector unsigned int);
	vector unsigned int vec_nand (vector unsigned int, vector bool unsigned int);
	vector unsigned int vec_nand (vector unsigned int, vector unsigned int);
	vector signed long long vec_nand (vector bool long long, vector signed long long);
	vector signed long long vec_nand (vector signed long long, vector bool long long);
	vector signed long long vec_nand (vector signed long long, vector signed long long);
	vector unsigned long long vec_nand (vector bool long long, vector unsigned long long);
	vector unsigned long long vec_nand (vector unsigned long long, vector bool long long);
	vector unsigned long long vec_nand (vector unsigned long long, vector unsigned long long);
	vector unsigned short vec_nand (vector bool unsigned short, vector unsigned short);
	vector unsigned short vec_nand (vector unsigned short, vector bool unsigned short);
	vector unsigned short vec_nand (vector unsigned short, vector unsigned short);
	vector int vec_nand (vector bool int, vector int);
	vector int vec_nand (vector int, vector bool int);
	vector int vec_nand (vector int, vector int);
	vector short vec_nand (vector bool short, vector short);
	vector short vec_nand (vector short, vector bool short);
	vector short vec_nand (vector short, vector short);
VEC_NEARBYINT(ARG1)	Purpose: Returns a vector containing the floating-point integral values nearest to the values of the corresponding elements of the given vector. Result value: Each element of the result contains the nearest representable floating-point integral value to the value of the corresponding element of ARG1. When an input element value is exactly between two integer values, the result value with the largest absolute value is selected.
	vector double vec_nearbyint (vector double);
	vector float vec_nearbyint (vector float);
VEC_NMADD(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a negative multiply-add operation on the given vectors. Result value: The value of each element of the result is the product of the corresponding elements of ARG1 and ARG2, added to the corresponding elements of ARG3, and then multiplied by -1.0.
	vector double vec_nmadd (vector double, vector double, vector double);
	vector float vec_nmadd (vector float, vector float, vector float);

Table A-2. Vector Built-In Function (with Prototype) (Page 17 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_NMSUB(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the results of performing a negative multiply-subtract operation on the given vectors. Result value: The value of each element of the result is the product of the corresponding elements of ARG1 and ARG2, subtracted from the corresponding element of ARG3, and then multiplied by 1.0.
	vector double vec_nmsub (vector double, vector double, vector double);
	vector float vec_nmsub (vector float, vector float, vector float);
VEC_NOR(ARG1, ARG2)	Purpose: Performs a bitwise NOR of the given vectors. Result value: The result is the bitwise NOR of ARG1 and ARG2.
	vector bool char vec_nor (vector bool char, vector bool char);
	vector signed char vec_nor (vector signed char, vector signed char);
	vector unsigned char vec_nor (vector unsigned char, vector unsigned char);
	vector bool int vec_nor (vector bool int, vector bool int);
	vector signed int vec_nor (vector signed int, vector signed int);
	vector unsigned int vec_nor (vector unsigned int, vector unsigned int);
	vector bool short vec_nor (vector bool short, vector bool short);
	vector signed short vec_nor (vector signed short, vector signed short);
	vector unsigned short vec_nor (vector unsigned short, vector unsigned short);
	vector double vec_nor (vector double, vector double);
	vector float vec_nor (vector float, vector float);
Phased in.[a]	vector signed long long vec_nor (vector signed long long, vector signed long long)
Phased in.[a]	vector signed long long vec_nor (vector signed long long, vector bool long long)
Phased in.[a]	vector signed long long vec_nor (vector bool long long, vector signed long long)
Phased in.[a]	vector unsigned long long vec_nor (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_nor (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_nor (vector bool long long, vector unsigned long long)
VEC_OR(ARG1, ARG2)	Purpose: Performs a bitwise OR of the given vectors. Result value: The result is the bitwise OR of ARG1 and ARG2.
	vector bool char vec_or (vector bool char, vector bool char);
	vector signed char vec_or (vector bool char, vector signed char);
	vector signed char vec_or (vector signed char, vector bool char);
	vector signed char vec_or (vector signed char, vector signed char);
	vector unsigned char vec_or (vector bool char, vector unsigned char);
	vector unsigned char vec_or (vector unsigned char, vector bool char);
	vector unsigned char vec_or (vector unsigned char, vector unsigned char);

Table A-2. Vector Built-In Function (with Prototype) (Page 18 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector bool int vec_or (vector bool int, vector bool int);
	vector signed int vec_or (vector bool int, vector signed int);
	vector signed int vec_or (vector signed int, vector bool int);
	vector signed int vec_or (vector signed int, vector signed int);
	vector unsigned int vec_or (vector bool int, vector unsigned int);
	vector unsigned int vec_or (vector unsigned int, vector bool int);
	vector unsigned int vec_or (vector unsigned int, vector unsigned int);
	vector bool short vec_or (vector bool short, vector bool short);
	vector signed short vec_or (vector bool short, vector signed short);
	vector signed short vec_or (vector signed short, vector bool short);
	vector signed short vec_or (vector signed short, vector signed short);
	vector unsigned short vec_or (vector bool short, vector unsigned short);
	vector unsigned short vec_or (vector unsigned short, vector bool short);
	vector unsigned short vec_or (vector unsigned short, vector unsigned short);
	vector double vec_or (vector bool long long, vector double);
	vector double vec_or (vector double, vector bool long long);
	vector double vec_or (vector double, vector double);
	vector float vec_or (vector bool int, vector float);
	vector float vec_or (vector float, vector bool int);
	vector float vec_or (vector float, vector float);
Phased in.[a]	vector signed long long vec_or (vector signed long long, vector signed long long)
Phased in.[a]	vector signed long long vec_or (vector signed long long, vector bool long long)
Phased in.[a]	vector signed long long vec_or (vector bool long long, vector signed long long)
Phased in.[a]	vector unsigned long long vec_or (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_or (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_or (vector bool long long, vector unsigned long long)
VEC_ORC(ARG1, ARG2)	Purpose: Performs a bitwise OR of the first vector with the negated second vector. Result value: The result is the bitwise OR of ARG1 and the bitwise negation of ARG2.
	vector signed char vec_orc (vector bool signed char, vector signed char);
	vector signed char vec_orc (vector signed char, vector bool signed char);
	vector signed char vec_orc (vector signed char, vector signed char);
	vector unsigned char vec_orc (vector bool unsigned char, vector unsigned char);
	vector unsigned char vec_orc (vector unsigned char, vector bool unsigned char);
	vector unsigned char vec_orc (vector unsigned char, vector unsigned char);
	vector unsigned int vec_orc (vector bool unsigned int, vector unsigned int);

Table A-2. Vector Built-In Function (with Prototype) (Page 19 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector unsigned int vec_orc (vector unsigned int, vector bool unsigned int);
	vector unsigned int vec_orc (vector unsigned int, vector unsigned int);
Phased in.[a]	vector signed long long vec_orc (vector bool long long, vector signed long long);
Phased in.[a]	vector signed long long vec_orc (vector signed long long, vector bool long long);
Phased in.[a]	vector signed long long vec_orc (vector signed long long, vector signed long long);
Phased in.[a]	vector unsigned long long vec_orc (vector bool long long, vector unsigned long long);
Phased in.[a]	vector unsigned long long vec_orc (vector unsigned long long, vector bool long long);
Phased in.[a]	vector unsigned long long vec_orc (vector unsigned long long, vector unsigned long long);
	vector unsigned short vec_orc (vector bool unsigned short, vector unsigned short);
	vector unsigned short vec_orc (vector unsigned short, vector bool unsigned short);
	vector unsigned short vec_orc (vector unsigned short, vector unsigned short);
	vector int vec_orc (vector bool int, vector int);
	vector int vec_orc (vector int, vector bool int);
	vector int vec_orc (vector int, vector int);
	vector short vec_orc (vector bool short, vector short);
	vector short vec_orc (vector short, vector bool short);
	vector short vec_orc (vector short, vector short);
VEC_PACK(ARG1, ARG2)	Purpose: Packs information from each element of two vectors into the result vector. Result value: The value of each element of the result vector is taken from the low-order half of the corresponding element of the result of concatenating ARG1 and ARG2.
	vector bool char vec_pack (vector bool short, vector bool short);
	vector signed char vec_pack (vector signed short, vector signed short);
	vector unsigned char vec_pack (vector unsigned short, vector unsigned short);
	vector bool int vec_pack (vector bool long long, vector bool long long);
	vector unsigned int vec_pack (vector unsigned long long, vector unsigned long long);
	vector bool short vec_pack (vector bool int, vector bool int);
	vector signed short vec_pack (vector signed int, vector signed int);
	vector unsigned short vec_pack (vector unsigned int, vector unsigned int);
	vector int vec_pack (vector signed long long, vector signed long long);
VEC_PACKPX(ARG1, ARG2)	Purpose: Packs information from each element of two vectors into the result vector. Result value: The value of each element of the result vector is taken from the corresponding element of the result of concatenating ARG1 and ARG2 in the following way: the least-significant bit of the high-order byte is stored into the first bit of the result element; the least-significant 5 bits of each of the remaining bytes are stored into the remaining portion of the result element.
	vector pixel vec_packpx (vector unsigned int, vector unsigned int);

Table A-2. Vector Built-In Function (with Prototype) (Page 20 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_PACKS(ARG1, ARG2)	Purpose: Packs information from each element of two vectors into the result vector, using saturated values. Result value: The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating ARG1 and ARG2.
	vector signed char vec_packs (vector signed short, vector signed short);
	vector unsigned char vec_packs (vector unsigned short, vector unsigned short);
	vector unsigned int vec_packs (vector unsigned long long, vector unsigned long long);
	vector signed short vec_packs (vector signed int, vector signed int);
	vector unsigned short vec_packs (vector unsigned int, vector unsigned int);
	vector int vec_packs (vector signed long long, vector signed long long);
VEC_PACKSU(ARG1, ARG2)	Purpose: Packs information from each element of two vectors into the result vector, using unsigned saturated values. Result value: The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating ARG1 and ARG2.
	vector unsigned char vec_packsu (vector signed short, vector signed short);
	vector unsigned char vec_packsu (vector unsigned short, vector unsigned short);
	vector unsigned int vec_packsu (vector signed long long, vector signed long long);
	vector unsigned short vec_packsu (vector signed int, vector signed int);
	vector unsigned short vec_packsu (vector unsigned int, vector unsigned int);
Phased in.[a]	vector unsigned int vec_packsu (vector unsigned long long, vector unsigned long long);
VEC_PERM(ARG1, ARG2, ARG3)	Purpose: Returns a vector that contains some elements of two vectors, in the order specified by a third vector. Result value: Each byte of the result is selected by using the least-significant 5 bits of the corresponding byte of ARG3 as an index into the concatenated bytes of ARG1 and ARG2.
	vector bool char vec_perm (vector bool char, vector bool char, vector unsigned char);
	vector signed char vec_perm (vector signed char, vector signed char, vector unsigned char);
	vector unsigned char vec_perm (vector unsigned char, vector unsigned char, vector unsigned char);
	vector bool int vec_perm (vector bool int, vector bool int, vector unsigned char);
	vector signed int vec_perm (vector signed int, vector signed int, vector unsigned char);
	vector unsigned int vec_perm (vector unsigned int, vector unsigned int, vector unsigned char);
	vector bool short vec_perm (vector bool short, vector bool short, vector unsigned char);
	vector signed short vec_perm (vector signed short, vector signed short, vector unsigned char);
	vector unsigned short vec_perm (vector unsigned short, vector unsigned short, vector unsigned char);
	vector double vec_perm (vector double, vector double, vector unsigned char);
	vector float vec_perm (vector float, vector float, vector unsigned char);
	vector pixel vec_perm (vector pixel, vector pixel, vector unsigned char);

Table A-2. Vector Built-In Function (with Prototype) (Page 21 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector signed long long vec_perm (vector signed long long, vector signed long long, vector unsigned char);
Phased in.[a]	vector unsigned long long vec_perm (vector unsigned long long, vector unsigned long long, vector unsigned char);
VEC_RE(ARG1)	Purpose: Returns a vector containing estimates of the reciprocals of the corresponding elements of the given vector. Result value: Each element of the result contains the estimated value of the reciprocal of the corresponding element of ARG1.
	vector float vec_re (vector float);
	vector double vec_re (vector double);
VEC_RECIPDIV(ARG1,ARG2)	Purpose: Returns a vector containing approximations of the division of the corresponding elements of ARG1 by the corresponding elements of ARG2. This implementation provides an implementation-dependent precision, which is commonly within 2 ulps for most of the numeric range expressible by the input operands. This built-in function does not correspond to a single IEEE operation and does not provide the overflow, underflow, and NaN propagation characteristics specified for IEEE division. (Precision may be a function of both the specified target processor model during compilation and the actual processor on which a program is executed.) Result value: Each element of the result vector contains a refined approximation of the division of the corresponding element of ARG1 by the corresponding element of ARG2.
	vector double vec_recipdiv (vector double, vector double);
	vector float vec_recipdiv (vector float, vector float);
VEC_RINT(ARG1)	Purpose: Returns a vector containing the floating-point integral values nearest to the values of the corresponding elements of the given vector. Result value: Each element of the result contains the nearest representable floating-point integral value to the value of the corresponding element of ARG1. When an input element value is exactly between two integer values, the result value is selected based on rounding mode specified by the Floating-Point Rounding Control field (RN) of the FPSCR register.
	vector double vec_rint (vector double);
VEC_RL(ARG1, ARG2)	Purpose: Rotates each element of a vector left by a given number of bits. Result value: Each element of the result is obtained by rotating the corresponding element of ARG1 left by the number of bits specified by the corresponding element of ARG2.
	vector signed char vec_rl (vector signed char, vector unsigned char);
	vector unsigned char vec_rl (vector unsigned char, vector unsigned char);
	vector signed int vec_rl (vector signed int, vector unsigned int);
	vector unsigned int vec_rl (vector unsigned int, vector unsigned int);
	vector signed long long vec_rl (vector signed long long, vector unsigned long long);
	vector signed long long vec_rl (vector unsigned long long, vector unsigned long long);
	vector signed short vec_rl (vector signed short, vector unsigned short);
	vector unsigned short vec_rl (vector unsigned short, vector unsigned short);

Table A-2. Vector Built-In Function (with Prototype) (Page 22 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_ROUND(ARG1)	Purpose: Returns a vector containing the rounded values of the corresponding elements of the given vector. Result value: Each element of the result contains the value of the corresponding element of ARG1, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding. **Note:** This function might not follow the strict operation definition of the resolution of a tie during a round if the -qstrict=nooperationprecision compiler option is specified.
	vector float vec_round (vector float);
Phased in.[a]	vector double vec_round (vector double);
VEC_RSQRT(ARG1)	Purpose: Returns a vector containing a refined approximation of the reciprocal square roots of the corresponding elements of the given vector. This function provides an implementation-dependent greater precision than VEC_RSQRTE. Result value: Each element of the result contains a refined approximation of the reciprocal square root of the corresponding element of ARG1.
	vector double vec_rsqrt (vector double);
	vector float vec_rsqrt (vector float);
VEC_RSQRTE(ARG1)	Purpose: Returns a vector containing estimates of the reciprocal square roots of the corresponding elements of the given vector. Result value: Each element of the result contains the estimated value of the reciprocal square root of the corresponding element of ARG1.
	vector double vec_rsqrte (vector double);
	vector float vec_rsqrte (vector float);
VEC_SEL(ARG1, ARG2, ARG3)	Purpose: Returns a vector containing the value of either ARG1 or ARG2 depending on the value of ARG3. Result value: Each bit of the result vector has the value of the corresponding bit of ARG1 if the corresponding bit of ARG3 is 0, or the value of the corresponding bit of ARG2 otherwise.
	vector bool char vec_sel (vector bool char, vector bool char, vector bool char);
	vector bool char vec_sel (vector bool char, vector bool char, vector unsigned char);
	vector signed char vec_sel (vector signed char, vector signed char, vector bool char);
	vector signed char vec_sel (vector signed char, vector signed char, vector unsigned char);
	vector unsigned char vec_sel (vector unsigned char, vector unsigned char, vector bool char);
	vector unsigned char vec_sel (vector unsigned char, vector unsigned char, vector unsigned char);
	vector bool int vec_sel (vector bool int, vector bool int, vector bool int);
	vector bool int vec_sel (vector bool int, vector bool int, vector unsigned int);
	vector signed int vec_sel (vector signed int, vector signed int, vector bool int);
	vector signed int vec_sel (vector signed int, vector signed int, vector unsigned int);
	vector unsigned int vec_sel (vector unsigned int, vector unsigned int, vector bool int);
	vector unsigned int vec_sel (vector unsigned int, vector unsigned int, vector unsigned int);
	vector bool short vec_sel (vector bool short, vector bool short, vector bool short);

Table A-2. Vector Built-In Function (with Prototype) (Page 23 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector bool short vec_sel (vector bool short, vector bool short, vector unsigned short);
	vector signed short vec_sel (vector signed short, vector signed short, vector bool short);
	vector signed short vec_sel (vector signed short, vector signed short, vector unsigned short);
	vector unsigned short vec_sel (vector unsigned short, vector unsigned short, vector bool short);
	vector unsigned short vec_sel (vector unsigned short, vector unsigned short, vector unsigned short);
	vector double vec_sel (vector double, vector double, vector bool long);
	vector double vec_sel (vector double, vector double, vector unsigned long);
	vector float vec_sel (vector float, vector float, vector bool int);
	vector float vec_sel (vector float, vector float, vector unsigned int);
	vector signed long long vec_sel (vector signed long long, vector signed long long)
	vector signed long long vec_sel (vector signed long long, vector bool long long)
	vector signed long long vec_sel (vector bool long long, vector signed long long)
Phased in.[a]	vector unsigned long long vec_sel (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_sel (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_sel (vector bool long long, vector unsigned long long)
VEC_SL(ARG1, ARG2)	Purpose: Performs a left shift for each element of a vector. Result value: Each element of the result vector is the result of left shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits that are shifted out are replaced by zeros.
	vector signed char vec_sl (vector signed char, vector unsigned char);
	vector unsigned char vec_sl (vector unsigned char, vector unsigned char);
	vector signed int vec_sl (vector signed int, vector unsigned int);
	vector unsigned int vec_sl (vector unsigned int, vector unsigned int);
	vector signed long long vec_sl (vector signed long long, vector unsigned long long);
	vector signed long long vec_sl (vector unsigned long long, vector unsigned long long);
	vector signed short vec_sl (vector signed short, vector unsigned short);
	vector unsigned short vec_sl (vector unsigned short, vector unsigned short);
VEC_SLD(ARG1, ARG2, ARG3)	Purpose: Left shifts two concatenated vectors by a given number of bytes. Result value: The result is the most-significant 16 bytes obtained by concatenating ARG1 and ARG2 and shifting left by the number of bytes specified by ARG3.
	vector bool char vec_sld (vector bool char, vector bool char, const int);
	vector signed char vec_sld (vector signed char, vector signed char, const int);
	vector unsigned char vec_sld (vector unsigned char, vector unsigned char, const int);
	vector bool int vec_sld (vector bool int, vector bool int, const int);
	vector signed int vec_sld (vector signed int, vector signed int, const int);
	vector unsigned int vec_sld (vector unsigned int, vector unsigned int, const int);

Table A-2. Vector Built-In Function (with Prototype) (Page 24 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector bool short vec_sld (vector bool short, vector bool short, const int);
	vector signed short vec_sld (vector signed short, vector signed short, const int);
	vector unsigned short vec_sld (vector unsigned short, vector unsigned short, const int);
	vector float vec_sld (vector float, vector float, const int);
	vector pixel vec_sld (vector pixel, vector pixel, const int);
VEC_SLL(ARG1, ARG2)	Purpose: Left shifts a vector by a given number of bits. Result value: The result is the contents of ARG1, shifted left by the number of bits specified by the three least-significant bits of ARG2. The bits that are shifted out are replaced by zeros.
	vector bool char vec_sll (vector bool char, vector unsigned char);
	vector bool char vec_sll (vector bool char, vector unsigned int);
	vector bool char vec_sll (vector bool char, vector unsigned short);
	vector signed char vec_sll (vector signed char, vector unsigned char);
	vector signed char vec_sll (vector signed char, vector unsigned int);
	vector signed char vec_sll (vector signed char, vector unsigned short);
	vector unsigned char vec_sll (vector unsigned char, vector unsigned char);
	vector unsigned char vec_sll (vector unsigned char, vector unsigned int);
	vector unsigned char vec_sll (vector unsigned char, vector unsigned short);
	vector bool int vec_sll (vector bool int, vector unsigned char);
	vector bool int vec_sll (vector bool int, vector unsigned int);
	vector bool int vec_sll (vector bool int, vector unsigned short);
	vector signed int vec_sll (vector signed int, vector unsigned char);
	vector signed int vec_sll (vector signed int, vector unsigned int);
	vector signed int vec_sll (vector signed int, vector unsigned short);
	vector unsigned int vec_sll (vector unsigned int, vector unsigned char);
	vector unsigned int vec_sll (vector unsigned int, vector unsigned int);
	vector unsigned int vec_sll (vector unsigned int, vector unsigned short);
	vector bool short vec_sll (vector bool short, vector unsigned char);
	vector bool short vec_sll (vector bool short, vector unsigned int);
	vector bool short vec_sll (vector bool short, vector unsigned short);
	vector signed short vec_sll (vector signed short, vector unsigned char);
	vector signed short vec_sll (vector signed short, vector unsigned int);
	vector signed short vec_sll (vector signed short, vector unsigned short);
	vector unsigned short vec_sll (vector unsigned short, vector unsigned char);
	vector unsigned short vec_sll (vector unsigned short, vector unsigned int);
	vector unsigned short vec_sll (vector unsigned short, vector unsigned short);
	vector pixel vec_sll (vector pixel, vector unsigned char);

Table A-2. Vector Built-In Function (with Prototype) (Page 25 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector pixel vec_sll (vector pixel, vector unsigned int);
	vector pixel vec_sll (vector pixel, vector unsigned short);
VEC_SLO(ARG1, ARG2)	Purpose: Left shifts a vector by a given number of bytes (octets). Result value: The result is the contents of ARG1, shifted left by the number of bytes specified by bits 121 - 124 of ARG2. The bits that are shifted out are replaced by zeros.
	vector signed char vec_slo (vector signed char, vector signed char);
	vector signed char vec_slo (vector signed char, vector unsigned char);
	vector unsigned char vec_slo (vector unsigned char, vector signed char);
	vector unsigned char vec_slo (vector unsigned char, vector unsigned char);
	vector signed int vec_slo (vector signed int, vector signed char);
	vector signed int vec_slo (vector signed int, vector unsigned char);
	vector unsigned int vec_slo (vector unsigned int, vector signed char);
	vector unsigned int vec_slo (vector unsigned int, vector unsigned char);
	vector signed short vec_slo (vector signed short, vector signed char);
	vector signed short vec_slo (vector signed short, vector unsigned char);
	vector unsigned short vec_slo (vector unsigned short, vector signed char);
	vector unsigned short vec_slo (vector unsigned short, vector unsigned char);
	vector float vec_slo (vector float, vector signed char);
	vector float vec_slo (vector float, vector unsigned char);
	vector pixel vec_slo (vector pixel, vector signed char);
	vector pixel vec_slo (vector pixel, vector unsigned char);
VEC_SPLAT(ARG1, ARG2)	Purpose: Returns a vector that has all of its elements set to a given value. Result value: The value of each element of the result is the value of the element of ARG1 specified by ARG2.
	vector bool char vec_splat (vector bool char, const int);
	vector signed char vec_splat (vector signed char, const int);
	vector unsigned char vec_splat (vector unsigned char, const int);
	vector bool int vec_splat (vector bool int, const int);
	vector signed int vec_splat (vector signed int, const int);
	vector unsigned int vec_splat (vector unsigned int, const int);
Phased in.[a]	vector bool long vec_splat (vector bool long, const int);
Phased in.[a]	vector signed long vec_splat (vector signed long, const int);
Phased in.[a]	vector unsigned long vec_splat (vector unsigned long, const int);
	vector bool short vec_splat (vector bool short, const int);
	vector signed short vec_splat (vector signed short, const int);
	vector unsigned short vec_splat (vector unsigned short, const int);

Table A-2. Vector Built-In Function (with Prototype) (Page 26 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector float vec_splat (vector float, const int);
Phased in.[a]	vector double vec_splat (vector double, const int);
	vector pixel vec_splat (vector pixel, const int);
Phased in.[a]	vector signed long long vec_splat (vector_signed long long, const int)
Phased in.[a]	vector unsigned long long vec_splat (vector unsigned long long, const int);
Phased in.[a]	vector double vec_splat (vec_double, const int);
VEC_SPLAT_S16(ARG1)	Purpose: Returns a vector with all elements equal to the given value. Result value: Each element of the result has the value of ARG1.
	vector signed short vec_splat_s16 (const int);
VEC_SPLAT_S32(ARG1)	Purpose: Returns a vector with all elements equal to the given value. Result value: Each element of the result has the value of ARG1.
	vector signed int vec_splat_s32 (const int);
VEC_SPLAT_S8(ARG1)	Purpose: Returns a vector with all elements equal to the given value. Result value: The bit pattern of ARG1 is interpreted as a signed value. Each element of the result is given this value.
	vector signed char vec_splat_s8 (const int);
VEC_SPLAT_U16(ARG1)	Purpose: Returns a vector with all elements equal to the given value. Result value: The bit pattern of ARG1 is interpreted as an unsigned value. Each element of the result is given this value.
	vector unsigned short vec_splat_u16 (const int);
VEC_SPLAT_U32(ARG1)	Purpose: Returns a vector with all elements equal to the given value. Result value: The bit pattern of ARG1 is interpreted as an unsigned value. Each element of the result is given this value.
	vector unsigned int vec_splat_u32 (const int);
VEC_SPLAT_U8(ARG1)	Purpose: Returns a vector with all elements equal to the given value. Result value: The bit pattern of ARG1 is interpreted as an unsigned value. Each element of the result is given this value.
	vector unsigned char vec_splat_u8 (const int);
VEC_SQRT(ARG1)	Purpose: Returns a vector containing the square root of each element in the given vector. Result value: Each element of the result vector is the square root of the corresponding element of ARG1.
	vector double vec_sqrt (vector double);

Table A-2. Vector Built-In Function (with Prototype) (Page 27 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector float vec_sqrt (vector float);
VEC_SR(ARG1,ARG2)	Purpose: Performs an algebraic right shift for each element of a vector. Result value: Each element of the result vector is the result of logically right shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits that are shifted out are replaced by zeros.
	vector signed char vec_sr (vector signed char, vector unsigned char);
	vector unsigned char vec_sr (vector unsigned char, vector unsigned char);
	vector signed int vec_sr (vector signed int, vector unsigned int);
	vector unsigned int vec_sr (vector unsigned int, vector unsigned int);
	vector signed long long vec_sr (vector signed long long, vector unsigned long long);
	vector unsigned long long vec_sr (vector unsigned long long, vector unsigned long long);
	vector signed short vec_sr (vector signed short, vector unsigned short);
	vector unsigned short vec_sr (vector unsigned short, vector unsigned short);
VEC_SRA(ARG1, ARG2)	Purpose: Performs an algebraic right shift for each element of a vector. Result value: Each element of the result vector is the result of algebraically right shifting the corresponding element of ARG1 by the number of bits specified by the value of the corresponding element of ARG2, modulo the number of bits in the element. The bits that are shifted out are replaced by copies of the most-significant bit of the element of ARG1.
	vector signed char vec_sra (vector signed char, vector unsigned char);
	vector unsigned char vec_sra (vector unsigned char, vector unsigned char);
	vector signed int vec_sra (vector signed int, vector unsigned int);
	vector unsigned int vec_sra (vector unsigned int, vector unsigned int);
	vector signed long long vec_sra (vector signed long long, vector unsigned long long);
	vector unsigned long long vec_sra (vector unsigned long long, vector unsigned long long);
	vector signed short vec_sra (vector signed short, vector unsigned short);
	vector unsigned short vec_sra (vector unsigned short, vector unsigned short);
VEC_SRL(ARG1, ARG2)	Purpose: Right shifts a vector by a given number of bits. Result value: The result is the contents of ARG1, shifted right by the number of bits specified by the 3 least-significant bits of ARG2. The bits that are shifted out are replaced by zeros.
	vector bool char vec_srl (vector bool char, vector unsigned char);
	vector bool char vec_srl (vector bool char, vector unsigned int);
	vector bool char vec_srl (vector bool char, vector unsigned short);
	vector signed char vec_srl (vector signed char, vector unsigned char);
	vector signed char vec_srl (vector signed char, vector unsigned int);
	vector signed char vec_srl (vector signed char, vector unsigned short);
	vector unsigned char vec_srl (vector unsigned char, vector unsigned char);

Table A-2. Vector Built-In Function (with Prototype) (Page 28 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector unsigned char vec_srl (vector unsigned char, vector unsigned int);
	vector unsigned char vec_srl (vector unsigned char, vector unsigned short);
	vector bool int vec_srl (vector bool int, vector unsigned char);
	vector bool int vec_srl (vector bool int, vector unsigned int);
	vector bool int vec_srl (vector bool int, vector unsigned short);
	vector signed int vec_srl (vector signed int, vector unsigned char);
	vector signed int vec_srl (vector signed int, vector unsigned int);
	vector signed int vec_srl (vector signed int, vector unsigned short);
	vector unsigned int vec_srl (vector unsigned int, vector unsigned char);
	vector unsigned int vec_srl (vector unsigned int, vector unsigned int);
	vector unsigned int vec_srl (vector unsigned int, vector unsigned short);
	vector bool short vec_srl (vector bool short, vector unsigned char);
	vector bool short vec_srl (vector bool short, vector unsigned int);
	vector bool short vec_srl (vector bool short, vector unsigned short);
	vector signed short vec_srl (vector signed short, vector unsigned char);
	vector signed short vec_srl (vector signed short, vector unsigned int);
	vector signed short vec_srl (vector signed short, vector unsigned short);
	vector unsigned short vec_srl (vector unsigned short, vector unsigned char);
	vector unsigned short vec_srl (vector unsigned short, vector unsigned int);
	vector unsigned short vec_srl (vector unsigned short, vector unsigned short);
	vector pixel vec_srl (vector pixel, vector unsigned char);
	vector pixel vec_srl (vector pixel, vector unsigned int);
	vector pixel vec_srl (vector pixel, vector unsigned short);
VEC_SRO(ARG1, ARG2)	Purpose: Right shifts a vector by a given number of bytes (octets). Result value: The result is the contents of ARG1, shifted right by the number of bytes specified by bits 121 - 124 of ARG2. The bits that are shifted out are replaced by zeros.
	vector signed char vec_sro (vector signed char, vector signed char);
	vector signed char vec_sro (vector signed char, vector unsigned char);
	vector unsigned char vec_sro (vector unsigned char, vector signed char);
	vector unsigned char vec_sro (vector unsigned char, vector unsigned char);
	vector signed int vec_sro (vector signed int, vector signed char);
	vector signed int vec_sro (vector signed int, vector unsigned char);
	vector unsigned int vec_sro (vector unsigned int, vector signed char);
	vector unsigned int vec_sro (vector unsigned int, vector unsigned char);
	vector signed short vec_sro (vector signed short, vector signed char);
	vector signed short vec_sro (vector signed short, vector unsigned char);

Table A-2. Vector Built-In Function (with Prototype) (Page 29 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector unsigned short vec_sro (vector unsigned short, vector signed char);
	vector unsigned short vec_sro (vector unsigned short, vector unsigned char);
	vector float vec_sro (vector float, vector signed char);
	vector float vec_sro (vector float, vector unsigned char);
	vector pixel vec_sro (vector pixel, vector signed char);
	vector pixel vec_sro (vector pixel, vector unsigned char);
VEC_SUB(ARG1, ARG2)	Purpose: Returns a vector containing the result of subtracting each element of ARG2 from the corresponding element of ARG1. This function emulates the operation on long long vectors. Result value: The value of each element of the result is the result of subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1. The arithmetic is modular for integer vectors.
	vector signed char vec_sub (vector bool char, vector signed char);
	vector signed char vec_sub (vector signed char, vector bool char);
	vector signed char vec_sub (vector signed char, vector signed char);
	vector unsigned char vec_sub (vector bool char, vector unsigned char);
	vector unsigned char vec_sub (vector unsigned char, vector bool char);
	vector unsigned char vec_sub (vector unsigned char, vector unsigned char);
	vector signed int vec_sub (vector bool int, vector signed int);
	vector signed int vec_sub (vector signed int, vector bool int);
	vector signed int vec_sub (vector signed int, vector signed int);
	vector unsigned int vec_sub (vector bool int, vector unsigned int);
	vector unsigned int vec_sub (vector unsigned int, vector bool int);
	vector unsigned int vec_sub (vector unsigned int, vector unsigned int);
	vector signed long long vec_sub (vector signed long long, vector signed long long);
	vector unsigned long long vec_sub (vector unsigned long long, vector unsigned long long);
	vector signed short vec_sub (vector bool short, vector signed short);
	vector signed short vec_sub (vector signed short, vector bool short);
	vector signed short vec_sub (vector signed short, vector signed short);
	vector unsigned short vec_sub (vector bool short, vector unsigned short);
	vector unsigned short vec_sub (vector unsigned short, vector bool short);
	vector unsigned short vec_sub (vector unsigned short, vector unsigned short);
	vector double vec_sub (vector double, vector double);
	vector float vec_sub (vector float, vector float);

Table A-2. Vector Built-In Function (with Prototype) (Page 30 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_SUBC(ARG1, ARG2)	Purpose: Returns a vector containing the carry produced by subtracting each set of corresponding elements of the given vectors. Result value: The value of each element of the result is the value of the carry produced by subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1. The value is 0 if a borrow occurred, or 1 if no borrow occurred.
	vector unsigned int vec_subc (vector unsigned int, vector unsigned int);
VEC_SUBS(ARG1, ARG2)	Purpose: Returns a vector containing the saturated differences of each set of corresponding elements of the given vectors. Result value: The value of each element of the result is the saturated result of subtracting the value of the corresponding element of ARG2 from the value of the corresponding element of ARG1.
	vector signed char vec_subs (vector bool char, vector signed char);
	vector signed char vec_subs (vector signed char, vector bool char);
	vector signed char vec_subs (vector signed char, vector signed char);
	vector unsigned char vec_subs (vector bool char, vector unsigned char);
	vector unsigned char vec_subs (vector unsigned char, vector bool char);
	vector unsigned char vec_subs (vector unsigned char, vector unsigned char);
	vector signed int vec_subs (vector bool int, vector signed int);
	vector signed int vec_subs (vector signed int, vector bool int);
	vector signed int vec_subs (vector signed int, vector signed int);
	vector unsigned int vec_subs (vector bool int, vector unsigned int);
	vector unsigned int vec_subs (vector unsigned int, vector bool int);
	vector unsigned int vec_subs (vector unsigned int, vector unsigned int);
	vector signed short vec_subs (vector bool short, vector signed short);
	vector signed short vec_subs (vector signed short, vector bool short);
	vector signed short vec_subs (vector signed short, vector signed short);
	vector unsigned short vec_subs (vector bool short, vector unsigned short);
	vector unsigned short vec_subs (vector unsigned short, vector bool short);
	vector unsigned short vec_subs (vector unsigned short, vector unsigned short);
VEC_SUM2S(ARG1, ARG2)	Purpose: Returns a vector containing the results of performing a sum-across-doublewords vector operation on the given vectors. Result value: The first and third element of the result are 0. The second element of the result contains the saturated sum of the first and second elements of ARG1 and the second element of ARG2. The fourth element of the result contains the saturated sum of the third and fourth elements of ARG1 and the fourth element of ARG2.
	vector signed int vec_sum2s (vector signed int, vector signed int);

Table A-2. Vector Built-In Function (with Prototype) (Page 31 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_SUM4S(ARG1, ARG2)	Purpose: Returns a vector containing the results of performing a sum-across-words vector operation on the given vectors. Result value: Assume that the elements of each vector are numbered beginning with 0. If ARG1 is a vector signed char vector or a vector unsigned char vector, then let m be 4. Otherwise, let m be 2. For each element n of the result vector, the value is obtained by adding elements mn through mn+m-1 of ARG1 and element n of ARG2 using saturated addition.
	vector signed int vec_sum4s (vector signed char, vector signed int);
	vector signed int vec_sum4s (vector signed short, vector signed int);
	vector unsigned int vec_sum4s (vector unsigned char, vector unsigned int);
VEC_SUMS(ARG1, ARG2)	Purpose: Returns a vector containing the results of performing a sum across vector operation on the given vectors. Result value: The first three elements of the result are 0. The fourth element is the saturated sum of all the elements of ARG1 and the fourth element of ARG2.
	vector signed int vec_sums (vector signed int, vector signed int);
VEC_TRUNC(ARG1)	Purpose: Returns a vector containing the truncated values of the corresponding elements of the given vector. **Note:** VEC_TRUNC is another name for VEC_ROUNDZ. Result value: Each element of the result contains the value of the corresponding element of ARG1, truncated to an integral value.
	vector double vec_trunc (vector double);
	vector float vec_trunc (vector float);
VEC_UNPACKH(ARG1)	Purpose: Unpacks the most-significant (“high”) half of a vector into a vector with larger elements. Result value: If ARG1 is an integer vector, the value of each element of the result is the value of the corresponding element of the most-significant half of ARG1. If ARG1 is a pixel vector, then the value of each element of the result is taken from the corresponding element of the most-significant half of ARG1 as follows: • All bits in the first byte of the element of the result are set to the value of the first bit of the element of ARG1. • The least-significant 5 bits of the second byte of the element of the result are set to the value of the next 5 bits in the element of ARG1. • The least-significant 5 bits of the third byte of the element of the result are set to the value of the next 5 bits in the element of ARG1. • The least-significant 5 bits of the fourth byte of the element of the result are set to the value of the next 5 bits in the element of ARG1.
	vector bool int vec_unpackh (vector bool short);
	vector signed int vec_unpackh (vector signed short);
	vector unsigned int vec_unpackh (vector pixel);
	vector signed long long vec_unpackh (vector int);
	vector bool short vec_unpackh (vector bool char);
	vector signed short vec_unpackh (vector signed char);

Table A-2. Vector Built-In Function (with Prototype) (Page 32 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
VEC_UNPACKL(ARG1)	Purpose: Unpacks the least-significant (“low”) half of a vector into a vector with larger elements. Result value: If ARG1 is an integer vector, the value of each element of the result is the value of the corresponding element of the least-significant half of ARG1. If ARG1 is a pixel vector, then the value of each element of the result is taken from the corresponding element of the least-significant half of ARG1 as follows: • All bits in the first byte of the element of the result are set to the value of the first bit of the element of ARG1. • The least-significant 5 bits of the second byte of the element of the result are set to the value of the next 5 bits in the element of ARG1. • The least-significant 5 bits of the third byte of the element of the result are set to the value of the next 5 bits in the element of ARG1. • The least-significant 5 bits of the fourth byte of the element of the result are set to the value of the next 5 bits in the element of ARG1.
	vector bool int vec_unpackl (vector bool short);
	vector signed int vec_unpackl (vector signed short);
	vector unsigned int vec_unpackl (vector pixel);
	vector signed long long vec_unpackl (vector signed int);
	vector unsigned long long vec_unpackl (vector unsigned int);
	vector bool short vec_unpackl (vector bool char);
	vector signed short vec_unpackl (vector signed char);
VEC_VCLZ(ARG1)	Purpose: Returns a vector containing the number of most-significant bits equal to 0 of each corresponding element of the given vector. Result value: The value of each element of the result is the sum of the corresponding single-precision floating-point elements of ARG1 and ARG2.
	vector signed char vec_vclz (vector signed char);
	vector unsigned char vec_vclz (vector unsigned char);
	vector unsigned int vec_vclz (vector unsigned int);
	vector signed long long vec_vclz (vector signed long long);
	vector unsigned long long vec_vclz (vector unsigned long long);
	vector unsigned short vec_vclz (vector unsigned short);
	vector int vec_vclz (vector int);
	vector short vec_vclz (vector short);
VEC_XOR(ARG1, ARG2)	Purpose: Performs a bitwise XOR of the given vectors. Result value: The result is the bitwise XOR of ARG1 and ARG2.
	vector bool char vec_xor (vector bool char, vector bool char);
	vector signed char vec_xor (vector bool char, vector signed char);
	vector signed char vec_xor (vector signed char, vector bool char);
	vector signed char vec_xor (vector signed char, vector signed char);
	vector unsigned char vec_xor (vector bool char, vector unsigned char);

Table A-2. Vector Built-In Function (with Prototype) (Page 33 of 33)

Group	Description/ Vector Built-In Function (with Prototype)
	vector unsigned char vec_xor (vector unsigned char, vector bool char);
	vector unsigned char vec_xor (vector unsigned char, vector unsigned char);
	vector bool int vec_xor (vector bool int, vector bool int);
	vector signed int vec_xor (vector bool int, vector signed int);
	vector signed int vec_xor (vector signed int, vector bool int);
	vector signed int vec_xor (vector signed int, vector signed int);
	vector unsigned int vec_xor (vector bool int, vector unsigned int);
	vector unsigned int vec_xor (vector unsigned int, vector bool int);
	vector unsigned int vec_xor (vector unsigned int, vector unsigned int);
	vector bool short vec_xor (vector bool short, vector bool short);
	vector signed short vec_xor (vector bool short, vector signed short);
	vector signed short vec_xor (vector signed short, vector bool short);
	vector signed short vec_xor (vector signed short, vector signed short);
	vector unsigned short vec_xor (vector bool short, vector unsigned short);
	vector unsigned short vec_xor (vector unsigned short, vector bool short);
	vector unsigned short vec_xor (vector unsigned short, vector unsigned short);
	vector double vec_xor (vector bool long, vector double);
	vector double vec_xor (vector double, vector bool long);
	vector double vec_xor (vector double, vector double);
	vector float vec_xor (vector bool int, vector float);
	vector float vec_xor (vector float, vector bool int);
	vector float vec_xor (vector float, vector float);
Phased in.[a]	vector signed long long vec_xor (vector signed long long, vector signed long long)
Phased in.[a]	vector signed long long vec_xor (vector signed long long, vector bool long long)
Phased in.[a]	vector signed long long vec_xor (vector bool long long, vector signed long long)
Phased in.[a]	vector unsigned long long vec_xor (vector unsigned long long, vector unsigned long long)
Phased in.[a]	vector unsigned long long vec_xor (vector unsigned long long, vector bool long long)
Phased in.[a]	vector unsigned long long vec_xor (vector bool long long, vector unsigned long long)

a.This optional function is being phased in and it may not be available on all implementations.

A.2 Built-In Vector Predicate Functions

Table A-3. Built-in Vector Predicate Functions (Page 1 of 12)

Group	Description / Built-in Vector Predicate Functions
VEC_ALL_EQ(ARG1, ARG2)	Purpose: Tests whether all sets of corresponding elements of the given vectors are equal. Result value: The result is 1 if each element of ARG1 is equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_all_eq (vector bool char, vector bool char);
	int vec_all_eq (vector bool char, vector signed char);
	int vec_all_eq (vector bool char, vector unsigned char);
	int vec_all_eq (vector bool int, vector bool int);
	int vec_all_eq (vector bool int, vector signed int);
	int vec_all_eq (vector bool int, vector unsigned int);
	int vec_all_eq (vector bool short, vector bool short);
	int vec_all_eq (vector bool short, vector signed short);
	int vec_all_eq (vector bool short, vector unsigned short);
	int vec_all_eq (vector double, vector double);
	int vec_all_eq (vector float, vector float);
	int vec_all_eq (vector signed long long, vector signed long long);
	int vec_all_eq (vector pixel, vector pixel);
	int vec_all_eq (vector signed char, vector bool char);
	int vec_all_eq (vector signed char, vector signed char);
	int vec_all_eq (vector signed int, vector bool int);
	int vec_all_eq (vector signed int, vector signed int);
	int vec_all_eq (vector signed short, vector bool short);
	int vec_all_eq (vector signed short, vector signed short);
	int vec_all_eq (vector unsigned char, vector bool char);
	int vec_all_eq (vector unsigned char, vector unsigned char);
	int vec_all_eq (vector unsigned int, vector bool int);
	int vec_all_eq (vector unsigned int, vector unsigned int);
	int vec_all_eq (vector unsigned short, vector bool short);
	int vec_all_eq (vector unsigned short, vector unsigned short);
VEC_ALL_GE(ARG1, ARG2)	Purpose: Tests whether all elements of the first argument are greater than or equal to the corresponding elements of the second argument. Result value: The result is 1 if all elements of ARG1 are greater than or equal to the corresponding elements of ARG2. Otherwise, the result is 0.
	int vec_all_ge (vector bool char, vector signed char);
	int vec_all_ge (vector bool char, vector unsigned char);

Table A-3. Built-in Vector Predicate Functions (Page 2 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_all_ge (vector bool int, vector signed int);
	int vec_all_ge (vector bool int, vector unsigned int);
	int vec_all_ge (vector bool short, vector signed short);
	int vec_all_ge (vector bool short, vector unsigned short);
	int vec_all_ge (vector double, vector double);
	int vec_all_ge (vector float, vector float);
	int vec_all_ge (vector signed long long, vector signed long long);
	int vec_all_ge (vector signed char, vector bool char);
	int vec_all_ge (vector signed char, vector signed char);
	int vec_all_ge (vector signed int, vector bool int);
	int vec_all_ge (vector signed int, vector signed int);
	int vec_all_ge (vector signed short, vector bool short);
	int vec_all_ge (vector signed short, vector signed short);
	int vec_all_ge (vector unsigned char, vector bool char);
	int vec_all_ge (vector unsigned char, vector unsigned char);
	int vec_all_ge (vector unsigned int, vector bool int);
	int vec_all_ge (vector unsigned int, vector unsigned int);
	int vec_all_ge (vector unsigned short, vector bool short);
	int vec_all_ge (vector unsigned short, vector unsigned short);
VEC_ALL_GT(ARG1, ARG2)	Purpose: Tests whether all elements of the first argument are greater than the corresponding elements of the second argument. Result value: The result is 1 if all elements of ARG1 are greater than the corresponding elements of ARG2. Otherwise, the result is 0.
	int vec_all_gt (vector bool char, vector signed char);
	int vec_all_gt (vector bool char, vector unsigned char);
	int vec_all_gt (vector bool int, vector signed int);
	int vec_all_gt (vector bool int, vector unsigned int);
	int vec_all_gt (vector bool short, vector signed short);
	int vec_all_gt (vector bool short, vector unsigned short);
	int vec_all_gt (vector double, vector double);
	int vec_all_gt (vector float, vector float);
	int vec_all_gt (vector signed long long, vector signed long long);
	int vec_all_gt (vector signed char, vector bool char);
	int vec_all_gt (vector signed char, vector signed char);
	int vec_all_gt (vector signed int, vector bool int);
	int vec_all_gt (vector signed int, vector signed int);

Table A-3. Built-in Vector Predicate Functions (Page 3 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_all_gt (vector signed short, vector bool short);
	int vec_all_gt (vector signed short, vector signed short);
	int vec_all_gt (vector unsigned char, vector bool char);
	int vec_all_gt (vector unsigned char, vector unsigned char);
	int vec_all_gt (vector unsigned int, vector bool int);
	int vec_all_gt (vector unsigned int, vector unsigned int);
	int vec_all_gt (vector unsigned short, vector bool short);
	int vec_all_gt (vector unsigned short, vector unsigned short);
VEC_ALL_IN(ARG1, ARG2)	Purpose: Tests whether each element of a given vector is within a given range. Result value: The result is 1 if all elements of ARG1 have values less than or equal to the value of the corresponding element of ARG2, and greater than or equal to the negative of the value of the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_all_in (vector float, vector float);
VEC_ALL_LE(ARG1, ARG2)	Purpose: Tests whether all elements of the first argument are less than or equal to the corresponding elements of the second argument. Result value: The result is 1 if all elements of ARG1 are less than or equal to the corresponding elements of ARG2. Otherwise, the result is 0.
	int vec_all_le (vector bool char, vector signed char);
	int vec_all_le (vector bool char, vector unsigned char);
	int vec_all_le (vector bool int, vector signed int);
	int vec_all_le (vector bool int, vector unsigned int);
	int vec_all_le (vector bool short, vector signed short);
	int vec_all_le (vector bool short, vector unsigned short);
	int vec_all_le (vector double, vector double);
	int vec_all_le (vector float, vector float);
	int vec_all_le (vector signed long long, vector signed long long);
	int vec_all_le (vector signed char, vector bool char);
	int vec_all_le (vector signed char, vector signed char);
	int vec_all_le (vector signed int, vector bool int);
	int vec_all_le (vector signed int, vector signed int);
	int vec_all_le (vector signed short, vector bool short);
	int vec_all_le (vector signed short, vector signed short);
	int vec_all_le (vector unsigned char, vector bool char);
	int vec_all_le (vector unsigned char, vector unsigned char);
	int vec_all_le (vector unsigned int, vector bool int);
	int vec_all_le (vector unsigned int, vector unsigned int);

Table A-3. Built-in Vector Predicate Functions (Page 4 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_all_le (vector unsigned short, vector bool short);
	int vec_all_le (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_all_le(vector unsigned long long, vector unsigned long long);
VEC_ALL_LT(ARG1, ARG2)	Purpose: Tests whether all elements of the first argument are less than the corresponding elements of the second argument. Result value: The result is 1 if all elements of ARG1 are less than the corresponding elements of ARG2. Otherwise, the result is 0.
	int vec_all_lt (vector bool char, vector signed char);
	int vec_all_lt (vector bool char, vector unsigned char);
	int vec_all_lt (vector bool int, vector signed int);
	int vec_all_lt (vector bool int, vector unsigned int);
	int vec_all_lt (vector bool short, vector signed short);
	int vec_all_lt (vector bool short, vector unsigned short);
	int vec_all_lt (vector double, vector double);
	int vec_all_lt (vector float, vector float);
	int vec_all_lt (vector signed long long, vector signed long long);
	int vec_all_lt (vector signed char, vector bool char);
	int vec_all_lt (vector signed char, vector signed char);
	int vec_all_lt (vector signed int, vector bool int);
	int vec_all_lt (vector signed int, vector signed int);
	int vec_all_lt (vector signed short, vector bool short);
	int vec_all_lt (vector signed short, vector signed short);
	int vec_all_lt (vector unsigned char, vector bool char);
	int vec_all_lt (vector unsigned char, vector unsigned char);
	int vec_all_lt (vector unsigned int, vector bool int);
	int vec_all_lt (vector unsigned int, vector unsigned int);
	int vec_all_lt (vector unsigned short, vector bool short);
	int vec_all_lt (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_all_lt(vector unsigned long long, vector unsigned long long);
VEC_ALL_NAN(ARG1)	Purpose: Tests whether each element of the given vector is a not-a-number (NaN). Result value: The result is 1 if each element of ARG1 is a NaN. Otherwise, the result is 0.
	int vec_all_nan (vector double);
	int vec_all_nan (vector float);

Table A-3. Built-in Vector Predicate Functions (Page 5 of 12)

Group	Description / Built-in Vector Predicate Functions
VEC_ALL_NE(ARG1, ARG2)	Purpose: Tests whether all sets of corresponding elements of the given vectors are not equal. Result value: The result is 1 if each element of ARG1 is not equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_all_ne (vector bool char, vector bool char);
	int vec_all_ne (vector bool char, vector signed char);
	int vec_all_ne (vector bool char, vector unsigned char);
	int vec_all_ne (vector bool int, vector bool int);
	int vec_all_ne (vector bool int, vector signed int);
	int vec_all_ne (vector bool int, vector unsigned int);
	int vec_all_ne (vector bool short, vector bool short);
	int vec_all_ne (vector bool short, vector signed short);
	int vec_all_ne (vector bool short, vector unsigned short);
	int vec_all_ne (vector double, vector double);
	int vec_all_ne (vector float, vector float);
	int vec_all_ne (vector signed long long, vector signed long long);
	int vec_all_ne (vector pixel, vector pixel);
	int vec_all_ne (vector signed char, vector bool char);
	int vec_all_ne (vector signed char, vector signed char);
	int vec_all_ne (vector signed int, vector bool int);
	int vec_all_ne (vector signed int, vector signed int);
	int vec_all_ne (vector signed short, vector bool short);
	int vec_all_ne (vector signed short, vector signed short);
	int vec_all_ne (vector unsigned char, vector bool char);
	int vec_all_ne (vector unsigned char, vector unsigned char);
	int vec_all_ne (vector unsigned int, vector bool int);
	int vec_all_ne (vector unsigned int, vector unsigned int);
	int vec_all_ne (vector unsigned short, vector bool short);
	int vec_all_ne (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_all_ne(vector unsigned long long, vector unsigned long long);
VEC_ALL_NGE(ARG1, ARG2)	Purpose: Tests whether each element of the first argument is not greater than or equal to the corresponding element of the second argument. Result value: The result is 1 if each element of ARG1 is not greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_all_nge (vector double, vector double);
	int vec_all_nge (vector float, vector float);

Table A-3. Built-in Vector Predicate Functions (Page 6 of 12)

Group	Description / Built-in Vector Predicate Functions
VEC_ALL_NGT(ARG1, ARG2)	Purpose: Tests whether each element of the first argument is not greater than the corresponding element of the second argument. Result value: The result is 1 if each element of ARG1 is not greater than the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_all_ngt (vector double, vector double);
	int vec_all_ngt (vector float, vector float);
VEC_ALL_NLE(ARG1, ARG2)	Purpose: Tests whether each element of the first argument is not less than or equal to the corresponding element of the second argument. Result value: The result is 1 if each element of ARG1 is not less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_all_nle (vector double, vector double);
	int vec_all_nle (vector float, vector float);
VEC_ALL_NLT(ARG1, ARG2)	Purpose: Tests whether each element of the first argument is not less than the corresponding element of the second argument. Result value: The result is 1 if each element of ARG1 is not less than the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_all_nlt (vector double, vector double);
	int vec_all_nlt (vector float, vector float);
VEC_ALL_NUMERIC(ARG1)	Purpose: Tests whether each element of the given vector is numeric (not a NaN). Result value: The result is 1 if each element of ARG1 is numeric (not a NaN). Otherwise, the result is 0.
	int vec_all_numeric (vector double);
	int vec_all_numeric (vector float);
VEC_ANY_EQ(ARG1, ARG2)	Purpose: Tests whether any set of corresponding elements of the given vectors are equal. Result value: The result is 1 if any element of ARG1 is equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_eq (vector bool char, vector bool char);
	int vec_any_eq (vector bool char, vector signed char);
	int vec_any_eq (vector bool char, vector unsigned char);
	int vec_any_eq (vector bool int, vector bool int);
	int vec_any_eq (vector bool int, vector signed int);
	int vec_any_eq (vector bool int, vector unsigned int);
	int vec_any_eq (vector bool short, vector bool short);
	int vec_any_eq (vector bool short, vector signed short);
	int vec_any_eq (vector bool short, vector unsigned short);

Table A-3. Built-in Vector Predicate Functions (Page 7 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_any_eq (vector double, vector double);
	int vec_any_eq (vector float, vector float);
	int vec_any_eq (vector signed long long, vector signed long long);
	int vec_any_eq (vector pixel, vector pixel);
	int vec_any_eq (vector signed char, vector bool char);
	int vec_any_eq (vector signed char, vector signed char);
	int vec_any_eq (vector signed int, vector bool int);
	int vec_any_eq (vector signed int, vector signed int);
	int vec_any_eq (vector signed short, vector bool short);
	int vec_any_eq (vector signed short, vector signed short);
	int vec_any_eq (vector unsigned char, vector bool char);
	int vec_any_eq (vector unsigned char, vector unsigned char);
	int vec_any_eq (vector unsigned int, vector bool int);
	int vec_any_eq (vector unsigned int, vector unsigned int);
	int vec_any_eq (vector unsigned short, vector bool short);
	int vec_any_eq (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_any_eq (vector unsigned long long, vector unsigned long long);
VEC_ANY_GE(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is greater than or equal to the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_ge (vector bool char, vector signed char);
	int vec_any_ge (vector bool char, vector unsigned char);
	int vec_any_ge (vector bool int, vector signed int);
	int vec_any_ge (vector bool int, vector unsigned int);
	int vec_any_ge (vector bool short, vector signed short);
	int vec_any_ge (vector bool short, vector unsigned short);
	int vec_any_ge (vector double, vector double);
	int vec_any_ge (vector float, vector float);
	int vec_any_ge (vector signed long long, vector signed long long);
	int vec_any_ge (vector signed char, vector bool char);
	int vec_any_ge (vector signed char, vector signed char);
	int vec_any_ge (vector signed int, vector bool int);
	int vec_any_ge (vector signed int, vector signed int);
	int vec_any_ge (vector signed short, vector bool short);
	int vec_any_ge (vector signed short, vector signed short);

Table A-3. Built-in Vector Predicate Functions (Page 8 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_any_ge (vector unsigned char, vector bool char);
	int vec_any_ge (vector unsigned char, vector unsigned char);
	int vec_any_ge (vector unsigned int, vector bool int);
	int vec_any_ge (vector unsigned int, vector unsigned int);
	int vec_any_ge (vector unsigned short, vector bool short);
	int vec_any_ge (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_any_ge (vector unsigned long long, vector unsigned long long);
VEC_ANY_GT(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is greater than the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is greater than the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_gt (vector bool char, vector signed char);
	int vec_any_gt (vector bool char, vector unsigned char);
	int vec_any_gt (vector bool int, vector signed int);
	int vec_any_gt (vector bool int, vector unsigned int);
	int vec_any_gt (vector bool short, vector signed short);
	int vec_any_gt (vector bool short, vector unsigned short);
	int vec_any_gt (vector double, vector double);
	int vec_any_gt (vector float, vector float);
	int vec_any_gt (vector signed long long, vector signed long long);
	int vec_any_gt (vector signed char, vector bool char);
	int vec_any_gt (vector signed char, vector signed char);
	int vec_any_gt (vector signed int, vector bool int);
	int vec_any_gt (vector signed int, vector signed int);
	int vec_any_gt (vector signed short, vector bool short);
	int vec_any_gt (vector signed short, vector signed short);
	int vec_any_gt (vector unsigned char, vector bool char);
	int vec_any_gt (vector unsigned char, vector unsigned char);
	int vec_any_gt (vector unsigned int, vector bool int);
	int vec_any_gt (vector unsigned int, vector unsigned int);
	int vec_any_gt (vector unsigned short, vector bool short);
	int vec_any_gt (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_any_gt (vector unsigned long long, vector unsigned long long);

Table A-3. Built-in Vector Predicate Functions (Page 9 of 12)

Group	Description / Built-in Vector Predicate Functions
VEC_ANY_LE(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is less than or equal to the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_le (vector bool char, vector signed char);
	int vec_any_le (vector bool char, vector unsigned char);
	int vec_any_le (vector bool int, vector signed int);
	int vec_any_le (vector bool int, vector unsigned int);
	int vec_any_le (vector bool short, vector signed short);
	int vec_any_le (vector bool short, vector unsigned short);
	int vec_any_le (vector double, vector double);
	int vec_any_le (vector float, vector float);
	int vec_any_le (vector signed long long, vector signed long long);
	int vec_any_le (vector signed char, vector bool char);
	int vec_any_le (vector signed char, vector signed char);
	int vec_any_le (vector signed int, vector bool int);
	int vec_any_le (vector signed int, vector signed int);
	int vec_any_le (vector signed short, vector bool short);
	int vec_any_le (vector signed short, vector signed short);
	int vec_any_le (vector unsigned char, vector bool char);
	int vec_any_le (vector unsigned char, vector unsigned char);
	int vec_any_le (vector unsigned int, vector bool int);
	int vec_any_le (vector unsigned int, vector unsigned int);
	int vec_any_le (vector unsigned short, vector bool short);
	int vec_any_le (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_any_le (vector unsigned long long, vector unsigned long long);
VEC_ANY_LT(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is less than the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is less than the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_lt (vector bool char, vector signed char);
	int vec_any_lt (vector bool char, vector unsigned char);
	int vec_any_lt (vector bool int, vector signed int);
	int vec_any_lt (vector bool int, vector unsigned int);
	int vec_any_lt (vector bool short, vector signed short);
	int vec_any_lt (vector bool short, vector unsigned short);

Table A-3. Built-in Vector Predicate Functions (Page 10 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_any_lt (vector double, vector double);
	int vec_any_lt (vector float, vector float);
	int vec_any_lt (vector signed long long, vector signed long long);
	int vec_any_lt (vector signed char, vector bool char);
	int vec_any_lt (vector signed char, vector signed char);
	int vec_any_lt (vector signed int, vector bool int);
	int vec_any_lt (vector signed int, vector signed int);
	int vec_any_lt (vector signed short, vector bool short);
	int vec_any_lt (vector signed short, vector signed short);
	int vec_any_lt (vector unsigned char, vector bool char);
	int vec_any_lt (vector unsigned char, vector unsigned char);
	int vec_any_lt (vector unsigned int, vector bool int);
	int vec_any_lt (vector unsigned int, vector unsigned int);
	int vec_any_lt (vector unsigned short, vector bool short);
	int vec_any_lt (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_any_lt (vector unsigned long long, vector unsigned long long);
VEC_ANY_NAN(ARG1)	Purpose: Tests whether any element of the given vector is a NaN. Result value: The result is 1 if any element of ARG1 is a NaN. Otherwise, the result is 0.
	int vec_any_nan (vector double);
	int vec_any_nan (vector float);
VEC_ANY_NE(ARG1, ARG2)	Purpose: Tests whether any set of corresponding elements of the given vectors are not equal. Result value: The result is 1 if any element of ARG1 is not equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_ne (vector bool char, vector bool char);
	int vec_any_ne (vector bool char, vector signed char);
	int vec_any_ne (vector bool char, vector unsigned char);
	int vec_any_ne (vector bool int, vector bool int);
	int vec_any_ne (vector bool int, vector signed int);
	int vec_any_ne (vector bool int, vector unsigned int);
	int vec_any_ne (vector bool short, vector bool short);
	int vec_any_ne (vector bool short, vector signed short);
	int vec_any_ne (vector bool short, vector unsigned short);
	int vec_any_ne (vector double, vector double);
	int vec_any_ne (vector float, vector float);
	int vec_any_ne (vector signed long long, vector signed long long);

Table A-3. Built-in Vector Predicate Functions (Page 11 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_any_ne (vector pixel, vector pixel);
	int vec_any_ne (vector signed char, vector bool char);
	int vec_any_ne (vector signed char, vector signed char);
	int vec_any_ne (vector signed int, vector bool int);
	int vec_any_ne (vector signed int, vector signed int);
	int vec_any_ne (vector signed short, vector bool short);
	int vec_any_ne (vector signed short, vector signed short);
	int vec_any_ne (vector unsigned char, vector bool char);
	int vec_any_ne (vector unsigned char, vector unsigned char);
	int vec_any_ne (vector unsigned int, vector bool int);
	int vec_any_ne (vector unsigned int, vector unsigned int);
	int vec_any_ne (vector unsigned short, vector bool short);
	int vec_any_ne (vector unsigned short, vector unsigned short);
Phased in.[a]	int vec_any_ne (vector unsigned long long, vector unsigned long long);
VEC_ANY_NGE(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is not greater than or equal to the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is not greater than or equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_nge (vector double, vector double);
	int vec_any_nge (vector float, vector float);
VEC_ANY_NGT(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is not greater than the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is not greater than the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_ngt (vector double, vector double);
	int vec_any_ngt (vector float, vector float);
VEC_ANY_NLE(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is not less than or equal to the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is not less than or equal to the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_nle (vector double, vector double);
	int vec_any_nle (vector float, vector float);
VEC_ANY_NLT(ARG1, ARG2)	Purpose: Tests whether any element of the first argument is not less than the corresponding element of the second argument. Result value: The result is 1 if any element of ARG1 is not less than the corresponding element of ARG2. Otherwise, the result is 0.

Table A-3. Built-in Vector Predicate Functions (Page 12 of 12)

Group	Description / Built-in Vector Predicate Functions
	int vec_any_nlt (vector double, vector double);
	int vec_any_nlt (vector float, vector float);
VEC_ANY_NUMERIC(ARG1)	Purpose: Tests whether any element of the given vector is numeric (not a NaN). Result value: The result is 1 if any element of ARG1 is numeric (not a NaN). Otherwise, the result is 0.
	int vec_any_numeric (vector double);
	int vec_any_numeric (vector float);
VEC_ANY_OUT(ARG1, ARG2)	Purpose: Tests whether the value of any element of a given vector is outside of a given range. Result value: The result is 1 if the value of any element of ARG1 is greater than the value of the corresponding element of ARG2 or less than the negative of the value of the corresponding element of ARG2. Otherwise, the result is 0.
	int vec_any_out (vector float, vector float);

a.This optional function is being phased in and it may not be available on all implementations.

A.3 Optional Built-In Functions

The following built-in functions may be supported in some environments based on a Power ISA Level 2.07 (corresponding to POWER8), or later.

The second argument to the built-in crypto vshasigmad and built-in crypto vshasigmaw functions must be a constant integer that is 0 or 1. The third argument to these built-in functions must be a constant integer in the range 0 - 15.

Table A-4. Optional Built-In Functions (Page 1 of 2)

vector unsigned long long __builtin_crypto_vsbox (vector unsigned long long);
vector unsigned long long __builtin_crypto_vcipher (vector unsigned long long, vector unsigned long long);
vector unsigned long long __builtin_crypto_vcipherlast (vector unsigned long long, vector unsigned long long);
vector unsigned long long __builtin_crypto_vncipher (vector unsigned long long, vector unsigned long long);
vector unsigned long long __builtin_crypto_vncipherlast (vector unsigned long long, vector unsigned long long);
vector unsigned char __builtin_crypto_vpermxor (vector unsigned char, vector unsigned char, vector unsigned char);
vector unsigned short __builtin_crypto_vpermxor (vector unsigned short, vector unsigned short, vector unsigned short);
vector unsigned int __builtin_crypto_vpermxor (vector unsigned int, vector unsigned int, vector unsigned int);
vector unsigned long long __builtin_crypto_vpermxor (vector unsigned long long, vector unsigned long long, vector unsigned long long);
vector unsigned char __builtin_crypto_vpmsumb (vector unsigned char, vector unsigned char);
vector unsigned short __builtin_crypto_vpmsumh (vector unsigned short, vector unsigned short);
vector unsigned int __builtin_crypto_vpmsumw (vector unsigned int, vector unsigned int);

Table A-4. Optional Built-In Functions (Page 2 of 2)

vector unsigned long long __builtin_crypto_vpmsumd (vector unsigned long long, vector unsigned long long);
vector unsigned long long __builtin_crypto_vshasigmad (vector unsigned long long, int, int);
vector unsigned int __builtin_crypto_vshasigmaw (vector unsigned int, int, int);

A.4 VSCR Management Built-in Functions

Table A-5. VSCR Management Functions

Group	Description / VSCR Management Functions
VEC_MTVSCR(ARG1)	Purpose: Copies the given value into the Vector Status and Control Register. The low-order 32 bits of ARG1 are copied into the VSCR. Result value: This function multiplies corresponding elements in the given vectors and then assigns the result to corresponding elements in the result vector.
	void vec_mtvscr (vector bool char);
	void vec_mtvscr (vector bool int);
	void vec_mtvscr (vector bool short);
	void vec_mtvscr (vector pixel);
	void vec_mtvscr (vector signed char);
	void vec_mtvscr (vector signed int);
	void vec_mtvscr (vector signed short);
	void vec_mtvscr (vector unsigned char);
	void vec_mtvscr (vector unsigned int);
	void vec_mtvscr (vector unsigned short);
VEC_MFVSCR	Purpose: Copies the contents of the Vector Status and Control Register into the result vector. Result value: The high-order 16 bits of the VSCR are copied into the seventh element of the result. The low-order 16 bits of the VSCR are copied into the eighth element of the result. All other elements are set to zero.
	vector unsigned short vec_mfvscr (void);

A.5 Compatibility Functions

It is recommended to provide the following functions for compatibility with previous versions of the Power SIMD vector environment. Where possible (subject to being supported by all targeted implementations of the Power SIMD environment), the use of type-generic built-in names is recommended.

Note: The type-specific vector built-in types are provided for legacy code compatibility only. The functions are deprecated, and support may be discontinued in the future. It is recommended that programmers use the respective overloaded vector built-in functions in conjunction with the appropriate vector type.

Table A-6. Functions Provided for Compatibility (Page 1 of 8)

ISA Level	Vector Built-In Function with Prototype
vmx	vector float vec_vaddfp (vector float, vector float);
vmx	vector signed char vec_vmaxsb (vector bool char, vector signed char);
vmx	vector signed char vec_vmaxsb (vector signed char, vector bool char);
vmx	vector signed char vec_vmaxsb (vector signed char, vector signed char);
vsx2	vector signed long long vec_vmaxsd (vector signed long long, vector signed long long);
vmx	vector signed short vec_vmaxsh (vector bool short, vector signed short);
vmx	vector signed short vec_vmaxsh (vector signed short, vector bool short);
vmx	vector signed short vec_vmaxsh (vector signed short, vector signed short);
vmx	vector signed int vec_vmaxsw (vector bool int, vector signed int);
vmx	vector signed int vec_vmaxsw (vector signed int, vector bool int);
vmx	vector signed int vec_vmaxsw (vector signed int, vector signed int);
vmx	vector signed char vec_vaddsbs (vector bool char, vector signed char);
vmx	vector signed char vec_vaddsbs (vector signed char, vector bool char);
vmx	vector signed char vec_vaddsbs (vector signed char, vector signed char);
vmx	vector signed short vec_vaddshs (vector signed short, vector bool short);
vmx	vector signed short vec_vaddshs (vector bool short, vector signed short);
vmx	vector signed short vec_vaddshs (vector signed short, vector signed short);
vmx	vector signed int vec_vaddsws (vector bool int, vector signed int);
vmx	vector signed int vec_vaddsws (vector signed int, vector bool int);
vmx	vector signed int vec_vaddsws (vector signed int, vector signed int);
vmx	vector signed char vec_vaddubm (vector bool char, vector signed char);
vmx	vector signed char vec_vaddubm (vector signed char, vector bool char);
vmx	vector signed char vec_vaddubm (vector signed char, vector signed char);
vmx	vector unsigned char vec_vaddubm (vector bool char, vector unsigned char);
vmx	vector unsigned char vec_vaddubm (vector unsigned char, vector bool char);
vmx	vector unsigned char vec_vaddubm (vector unsigned char, vector unsigned char);
vmx	vector unsigned char vec_vaddubs (vector bool char, vector unsigned char);
vmx	vector unsigned char vec_vaddubs (vector unsigned char, vector bool char);
vmx	vector unsigned char vec_vaddubs (vector unsigned char, vector unsigned char);
vsx2	vector signed long long vec_vaddudm (vector bool long long, vector signed long long);
vsx2	vector signed long long vec_vaddudm (vector signed long long, vector bool long long);
vsx2	vector signed long long vec_vaddudm (vector signed long long, vector signed long long);
vsx2	vector unsigned long long vec_vaddudm (vector bool unsigned long long, vector unsigned long long);
vsx2	vector unsigned long long vec_vaddudm (vector unsigned long long, vector bool unsigned long long);
vsx2	vector unsigned long long vec_vaddudm (vector unsigned long long, vector unsigned long long);
vmx	vector signed short vec_vadduhm (vector bool short, vector signed short);

Table A-6. Functions Provided for Compatibility (Page 2 of 8)

ISA Level	Vector Built-In Function with Prototype
vmx	vector signed short vec_vadduhm (vector signed short, vector bool short);
vmx	vector signed short vec_vadduhm (vector signed short, vector signed short);
vmx	vector unsigned short vec_vadduhm (vector bool short, vector unsigned short);
vmx	vector unsigned short vec_vadduhm (vector unsigned short, vector bool short);
vmx	vector unsigned short vec_vadduhm (vector unsigned short, vector unsigned short);
vmx	vector unsigned short vec_vadduhs (vector bool short, vector unsigned short);
vmx	vector unsigned short vec_vadduhs (vector unsigned short, vector bool short);
vmx	vector unsigned short vec_vadduhs (vector unsigned short, vector unsigned short);
vmx	vector unsigned int vec_vadduwm (vector unsigned int, vector bool int);
vmx	vector signed int vec_vadduwm (vector bool int, vector signed int);
vmx	vector signed int vec_vadduwm (vector signed int, vector bool int);
vmx	vector signed int vec_vadduwm (vector signed int, vector signed int);
vmx	vector unsigned int vec_vadduwm (vector bool int, vector unsigned int);
vmx	vector unsigned int vec_vadduwm (vector unsigned int, vector unsigned int);
vmx	vector unsigned int vec_vadduws (vector bool int, vector unsigned int);
vmx	vector unsigned int vec_vadduws (vector unsigned int, vector bool int);
vmx	vector unsigned int vec_vadduws (vector unsigned int, vector unsigned int);
vmx	vector signed char vec_vavgsb (vector signed char, vector signed char);
vmx	vector signed short vec_vavgsh (vector signed short, vector signed short);
vmx	vector signed int vec_vavgsw (vector signed int, vector signed int);
vmx	vector unsigned char vec_vavgub (vector unsigned char, vector unsigned char);
vmx	vector unsigned short vec_vavguh (vector unsigned short, vector unsigned short);
vmx	vector unsigned int vec_vavguw (vector unsigned int, vector unsigned int);
vsx2	vector signed char vec_vclzb (vector signed char);
vsx2	vector unsigned char vec_vclzb (vector unsigned char);
vsx2	vector signed long long vec_vclzd (vector signed long long);
vsx2	vector unsigned long long vec_vclzd (vector unsigned long long);
vsx2	vector unsigned short vec_vclzh (vector unsigned short);
vsx2	vector short vec_vclzh (vector short);
vsx2	vector unsigned int vec_vclzw (vector int);
vsx2	vector int vec_vclzw (vector int);
vmx	vector float vec_vcfsx (vector signed int, const int);
vmx	vector float vec_vcfux (vector unsigned int, const int);
vmx	vector bool int vec_vcmpeqfp (vector float, vector float);
vmx	vector bool char vec_vcmpequb (vector signed char, vector signed char);
vmx	vector bool char vec_vcmpequb (vector unsigned char, vector unsigned char);
vmx	vector bool short vec_vcmpequh (vector signed short, vector signed short);

Table A-6. Functions Provided for Compatibility (Page 3 of 8)

ISA Level	Vector Built-In Function with Prototype
vmx	vector bool short vec_vcmpequh (vector unsigned short, vector unsigned short);
vmx	vector bool int vec_vcmpequw (vector signed int, vector signed int);
vmx	vector bool int vec_vcmpequw (vector unsigned int, vector unsigned int);
vmx	vector bool int vec_vcmpgtfp (vector float, vector float);
vmx	vector bool char vec_vcmpgtsb (vector signed char, vector signed char);
vmx	vector bool short vec_vcmpgtsh (vector signed short, vector signed short);
vmx	vector bool short vec_vcmpgtsh (vector signed short, vector signed short);
vmx	vector bool int vec_vcmpgtsw (vector signed int, vector signed int);
vmx	vector bool char vec_vcmpgtub (vector unsigned char, vector unsigned char);
vmx	vector bool short vec_vcmpgtuh (vector unsigned short, vector unsigned short);
vmx	vector bool int vec_vcmpgtuw (vector unsigned int, vector unsigned int);
vmx	vector float vec_vmaxfp (vector float, vector float);
vmx	vector unsigned char vec_vmaxub (vector bool char, vector unsigned char);
vmx	vector unsigned char vec_vmaxub (vector bool char, vector unsigned char);
vmx	vector unsigned char vec_vmaxub (vector unsigned char, vector bool char);
vmx	vector unsigned char vec_vmaxub (vector unsigned char, vector unsigned char);
vsx2	vector unsigned long long vec_vmaxud (vector unsigned long long, unsigned vector long long);
vmx	vector unsigned short vec_vmaxuh (vector bool short, vector unsigned short);
vmx	vector unsigned short vec_vmaxuh (vector unsigned short, vector bool short);
vmx	vector unsigned short vec_vmaxuh (vector unsigned short, vector unsigned short);
vmx	vector unsigned int vec_vmaxuw (vector bool int, vector unsigned int);
vmx	vector unsigned int vec_vmaxuw (vector unsigned int, vector bool int);
vmx	vector unsigned int vec_vmaxuw (vector unsigned int, vector unsigned int);
vmx	vector float vec_vminfp (vector float, vector float);
vmx	vector signed char vec_vminsb (vector bool char, vector signed char);
vmx	vector signed char vec_vminsb (vector signed char, vector bool char);
vmx	vector signed char vec_vminsb (vector signed char, vector signed char);
vsx2	vector signed long long vec_vminsd (vector signed long long, vector signed long long);
vmx	vector signed short vec_vminsh (vector bool short, vector signed short);
vmx	vector signed short vec_vminsh (vector signed short, vector bool short);
vmx	vector signed short vec_vminsh (vector signed short, vector signed short);
vmx	vector signed int vec_vminsw (vector bool int, vector signed int);
vmx	vector signed int vec_vminsw (vector signed int, vector bool int);
vmx	vector signed int vec_vminsw (vector signed int, vector signed int);
vmx	vector unsigned char vec_vminub (vector bool char, vector unsigned char);
vmx	vector unsigned char vec_vminub (vector unsigned char, vector bool char);
vmx	vector unsigned char vec_vminub (vector unsigned char, vector unsigned char);

Table A-6. Functions Provided for Compatibility (Page 4 of 8)

ISA Level	Vector Built-In Function with Prototype
vsx2	vector unsigned long long vec_vminud (vector unsigned long long, vector unsigned long long);
vmx	vector unsigned short vec_vminuh (vector bool short, vector unsigned short);
vmx	vector unsigned short vec_vminuh (vector unsigned short, vector bool short);
vmx	vector unsigned short vec_vminuh (vector unsigned short, vector unsigned short);
vmx	vector unsigned int vec_vminuw (vector bool int, vector unsigned int);
vmx	vector unsigned int vec_vminuw (vector unsigned int, vector bool int);
vmx	vector unsigned int vec_vminuw (vector unsigned int, vector unsigned int);
vmx	vector float vec_vsubfp (vector float, vector float);
vsx	vector bool int vec_vcmpeqdp (vector double, vector double);
vsx	vector bool int vec_vcmpgtdp (vector double, vector double);
vmx	vector bool char vec_vmrghb (vector bool char, vector bool char);
vmx	vector signed char vec_vmrghb (vector signed char, vector signed char);
vmx	vector unsigned char vec_vmrghb (vector unsigned char, vector unsigned char);
vmx	vector bool short vec_vmrghh (vector bool short, vector bool short);
vmx	vector signed short vec_vmrghh (vector signed short, vector signed short);
vmx	vector unsigned short vec_vmrghh (vector unsigned short, vector unsigned short);
vmx	vector pixel vec_vmrghh (vector pixel, vector pixel);
vmx	vector bool int vec_vmrghw (vector bool int, vector bool int);
vmx	vector signed int vec_vmrghw (vector signed int, vector signed int);
vmx	vector unsigned int vec_vmrghw (vector unsigned int, vector unsigned int);
vmx	vector float vec_vmrghw (vector float, vector float);
vmx	vector bool char vec_vmrglb (vector bool char, vector bool char);
vmx	vector signed char vec_vmrglb (vector signed char, vector signed char);
vmx	vector unsigned char vec_vmrglb (vector unsigned char, vector unsigned char);
vmx	vector bool short vec_vmrglh (vector bool short, vector bool short);
vmx	vector signed short vec_vmrglh (vector signed short, vector signed short);
vmx	vector unsigned short vec_vmrglh (vector unsigned short, vector unsigned short);
vmx	vector pixel vec_vmrglh (vector pixel, vector pixel);
vmx	vector bool int vec_vmrglw (vector bool int, vector bool int);
vmx	vector signed int vec_vmrglw (vector signed int, vector signed int);
vmx	vector unsigned int vec_vmrglw (vector unsigned int, vector unsigned int);
vmx	vector float vec_vmrglw (vector float, vector float);
vmx	vector signed int vec_vmsummbm (vector signed char, vector unsigned char, vector signed int);
vmx	vector signed int vec_vmsumshm (vector signed short, vector signed short, vector signed int);
vmx	vector signed int vec_vmsumshs (vector signed short, vector signed short, vector signed int);
vmx	vector unsigned int vec_vmsumubm (vector unsigned char, vector unsigned char, vector unsigned int);

Table A-6. Functions Provided for Compatibility (Page 5 of 8)

ISA Level	Vector Built-In Function with Prototype
vmx	vector unsigned int vec_vmsumuhm (vector unsigned short, vector unsigned short, vector unsigned int);
vmx	vector unsigned int vec_vmsumuhs (vector unsigned short, vector unsigned short, vector unsigned int);
vmx	vector signed short vec_vmulesb (vector signed char, vector signed char);
vmx	vector signed int vec_vmulesh (vector signed short, vector signed short);
vmx	vector unsigned short vec_vmuleub (vector unsigned char, vector unsigned char);
vmx	vector unsigned int vec_vmuleuh (vector unsigned short, vector unsigned short);
vmx	vector signed short vec_vmulosb (vector signed char, vector signed char);
vmx	vector signed int vec_vmulosh (vector signed short, vector signed short);
vmx	vector unsigned short vec_vmuloub (vector unsigned char, vector unsigned char);
vmx	vector unsigned int vec_vmulouh (vector unsigned short, vector unsigned short);
vsx2	vector unsigned int vec_vpksdss (vector unsigned long long, vector unsigned long long);
vsx2	vector int vec_vpksdss (vector signed long long, vector signed long long);
vmx	vector signed char vec_vpkshss (vector signed short, vector signed short);
vmx	vector unsigned char vec_vpkshus (vector signed short, vector signed short);
vmx	vector signed short vec_vpkswss (vector signed int, vector signed int);
vmx	vector unsigned short vec_vpkswus (vector signed int, vector signed int);
vsx2	vector bool int vec_vpkudum (vector bool long long, vector bool long long);
vsx2	vector unsigned int vec_vpkudum (vector unsigned long long, vector unsigned long long);
vsx2	vector int vec_vpkudum (vector signed long long, vector signed long long);
vsx2	vector unsigned int vec_vpkudus (vector unsigned long long, vector unsigned long long);
vmx	vector bool char vec_vpkuhum (vector bool short, vector bool short);
vmx	vector signed char vec_vpkuhum (vector signed short, vector signed short);
vmx	vector unsigned char vec_vpkuhum (vector unsigned short, vector unsigned short);
vmx	vector unsigned char vec_vpkuhus (vector unsigned short, vector unsigned short);
vmx	vector bool short vec_vpkuwum (vector bool int, vector bool int);
vmx	vector signed short vec_vpkuwum (vector signed int, vector signed int);
vmx	vector unsigned short vec_vpkuwum (vector unsigned int, vector unsigned int);
vmx	vector unsigned short vec_vpkuwus (vector unsigned int, vector unsigned int);
vsx2	vector signed char vec_vpopcnt (vector signed char);
vsx2	vector unsigned char vec_vpopcnt (vector unsigned char);
vsx2	vector unsigned int vec_vpopcnt (vector int);
vsx2	vector signed long long vec_vpopcnt (vector signed long long);
vsx2	vector unsigned long long vec_vpopcnt (vector unsigned long long);
vsx2	vector unsigned short vec_vpopcnt (vector unsigned short);
vsx2	vector int vec_vpopcnt (vector int);

Table A-6. Functions Provided for Compatibility (Page 6 of 8)

ISA Level	Vector Built-In Function with Prototype
vsx2	vector short vec_vpopcnt (vector short);
vsx2	vector signed char vec_vpopcntb (vector signed char);
vsx2	vector unsigned char vec_vpopcntb (vector unsigned char);
vsx2	vector signed long long vec_vpopcntd (vector signed long long);
vsx2	vector unsigned long long vec_vpopcntd (vector unsigned long long);
vsx2	vector unsigned short vec_vpopcnth (vector unsigned short);
vsx2	vector short vec_vpopcnth (vector short);
vsx2	vector unsigned int vec_vpopcntw (vector int);
vsx2	vector int vec_vpopcntw (vector int);
vmx	vector signed char vec_vrlb (vector signed char, vector unsigned char);
vmx	vector unsigned char vec_vrlb (vector unsigned char, vector unsigned char);
vsx2	vector signed long long vec_vrld (vector signed long long, vector unsigned long long);
vsx2	vector unsigned long long vec_vrld (vector unsigned long long, vector unsigned long long);
vmx	vector signed short vec_vrlh (vector signed short, vector unsigned short);
vmx	vector unsigned short vec_vrlh (vector unsigned short, vector unsigned short);
vmx	vector signed int vec_vrlw (vector signed int, vector unsigned int);
vmx	vector unsigned int vec_vrlw (vector unsigned int, vector unsigned int);
vmx	vector signed char vec_vslb (vector signed char, vector unsigned char);
vmx	vector unsigned char vec_vslb (vector unsigned char, vector unsigned char);
vsx2	vector signed long long vec_vsld (vector signed long long, vector unsigned long long);
vsx2	vector signed long long vec_vsld (vector unsigned long long, vector unsigned long long);
vmx	vector signed short vec_vslh (vector signed short, vector unsigned short);
vmx	vector unsigned short vec_vslh (vector unsigned short, vector unsigned short);
vmx	vector signed int vec_vslw (vector signed int, vector unsigned int);
vmx	vector unsigned int vec_vslw (vector unsigned int, vector unsigned int);
vmx	vector bool char vec_vspltb (vector bool char, const int);
vmx	vector signed char vec_vspltb (vector signed char, const int);
vmx	vector unsigned char vec_vspltb (vector unsigned char, const int);
vmx	vector bool short vec_vsplth (vector bool short, const int);
vmx	vector signed short vec_vsplth (vector signed short, const int);
vmx	vector unsigned short vec_vsplth (vector unsigned short, const int);
vmx	vector pixel vec_vsplth (vector pixel, const int);
vmx	vector bool int vec_vspltw (vector bool int, const int);
vmx	vector signed int vec_vspltw (vector signed int, const int);
vmx	vector unsigned int vec_vspltw (vector unsigned int, const int);
vmx	vector float vec_vspltw (vector float, const int);
vmx	vector signed char vec_vsrab (vector signed char, vector unsigned char);

Table A-6. Functions Provided for Compatibility (Page 7 of 8)

ISA Level	Vector Built-In Function with Prototype
vmx	vector unsigned char vec_vsrab (vector unsigned char, vector unsigned char);
vsx2	vector signed long long vec_vsrad (vector signed long long, vector unsigned long long);
vsx2	vector unsigned long long vec_vsrad (vector unsigned long long, vector unsigned long long);
vmx	vector signed short vec_vsrah (vector signed short, vector unsigned short);
vmx	vector unsigned short vec_vsrah (vector unsigned short, vector unsigned short);
vmx	vector signed int vec_vsraw (vector signed int, vector unsigned int);
vmx	vector unsigned int vec_vsraw (vector unsigned int, vector unsigned int);
vmx	vector signed char vec_vsrb (vector signed char, vector unsigned char);
vmx	vector unsigned char vec_vsrb (vector unsigned char, vector unsigned char);
vsx2	vector signed long long vec_vsrd (vector signed long long, vector unsigned long long);
vsx2	vector unsigned long long vec_vsrd (vector unsigned long long, vector unsigned long long);
vmx	vector signed short vec_vsrh (vector signed short, vector unsigned short);
vmx	vector unsigned short vec_vsrh (vector unsigned short, vector unsigned short);
vmx	vector signed int vec_vsrw (vector signed int, vector unsigned int);
vmx	vector unsigned int vec_vsrw (vector unsigned int, vector unsigned int);
vmx	vector signed char vec_vsubsbs (vector bool char, vector signed char);
vmx	vector signed char vec_vsubsbs (vector signed char, vector bool char);
vmx	vector signed char vec_vsubsbs (vector signed char, vector signed char);
vmx	vector signed short vec_vsubshs (vector bool short, vector signed short);
vmx	vector signed short vec_vsubshs (vector signed short, vector bool short);
vmx	vector signed short vec_vsubshs (vector signed short, vector signed short);
vmx	vector signed int vec_vsubsws (vector bool int, vector signed int);
vmx	vector signed int vec_vsubsws (vector signed int, vector bool int);
vmx	vector signed int vec_vsubsws (vector signed int, vector signed int);
vmx	vector signed char vec_vsububm (vector bool char, vector signed char);
vmx	vector signed char vec_vsububm (vector signed char, vector bool char);
vmx	vector signed char vec_vsububm (vector signed char, vector signed char);
vmx	vector unsigned char vec_vsububm (vector bool char, vector unsigned char);
vmx	vector unsigned char vec_vsububm (vector unsigned char, vector bool char);
vmx	vector unsigned char vec_vsububm (vector unsigned char, vector unsigned char);
vmx	vector unsigned char vec_vsububs (vector bool char, vector unsigned char);
vmx	vector unsigned char vec_vsububs (vector unsigned char, vector bool char);
vmx	vector unsigned char vec_vsububs (vector unsigned char, vector unsigned char);
vsx2	vector signed long long vec_vsubudm (vector bool long long, vector signed long long);
vsx2	vector signed long long vec_vsubudm (vector signed long long, vector bool long long);
vsx2	vector signed long long vec_vsubudm (vector signed long long, vector signed long long);
vsx2	vector unsigned long long vec_vsubudm (vector bool long long, vector unsigned long long);

Table A-6. Functions Provided for Compatibility (Page 8 of 8)

ISA Level	Vector Built-In Function with Prototype
vsx2	vector unsigned long long vec_vsubudm (vector unsigned long long, vector bool long long);
vsx2	vector unsigned long long vec_vsubudm (vector unsigned long long, vector unsigned long long);
vmx	vector signed short vec_vsubuhm (vector bool short, vector signed short);
vmx	vector signed short vec_vsubuhm (vector signed short, vector bool short);
vmx	vector signed short vec_vsubuhm (vector signed short, vector signed short);
vmx	vector unsigned short vec_vsubuhm (vector bool short, vector unsigned short);
vmx	vector unsigned short vec_vsubuhm (vector unsigned short, vector bool short);
vmx	vector unsigned short vec_vsubuhm (vector unsigned short, vector unsigned short);
vmx	vector unsigned short vec_vsubuhs (vector bool short, vector unsigned short);
vmx	vector unsigned short vec_vsubuhs (vector unsigned short, vector bool short);
vmx	vector unsigned short vec_vsubuhs (vector unsigned short, vector unsigned short);
vmx	vector signed int vec_vsubuwm (vector bool int, vector signed int);
vmx	vector signed int vec_vsubuwm (vector signed int, vector bool int);
vmx	vector signed int vec_vsubuwm (vector signed int, vector signed int);
vmx	vector unsigned int vec_vsubuwm (vector bool int, vector unsigned int);
vmx	vector unsigned int vec_vsubuwm (vector unsigned int, vector bool int);
vmx	vector unsigned int vec_vsubuwm (vector unsigned int, vector unsigned int);
vmx	vector unsigned int vec_vsubuws (vector bool int, vector unsigned int);
vmx	vector unsigned int vec_vsubuws (vector unsigned int, vector bool int);
vmx	vector unsigned int vec_vsubuws (vector unsigned int, vector unsigned int);
vmx	vector signed int vec_vsum4sbs (vector signed char, vector signed int);
vmx	vector signed int vec_vsum4shs (vector signed short, vector signed int);
vmx	vector unsigned int vec_vsum4ubs (vector unsigned char, vector unsigned int);
vmx	vector unsigned int vec_vupkhpx (vector pixel);
vmx	vector bool short vec_vupkhsb (vector bool char);
vmx	vector signed short vec_vupkhsb (vector signed char);
vmx	vector bool int vec_vupkhsh (vector bool short);
vmx	vector signed int vec_vupkhsh (vector signed short);
vsx2	vector signed long long vec_vupkhsw (vector int);
vsx2	vector unsigned long long vec_vupkhsw (vector unsigned int);
vmx	vector unsigned int vec_vupklpx (vector pixel);
vmx	vector bool short vec_vupklsb (vector bool char);
vmx	vector signed short vec_vupklsb (vector signed char);
vmx	vector bool int vec_vupklsh (vector bool short);
vmx	vector signed int vec_vupklsh (vector signed short);
vsx2	vector signed long long vec_vupklsw (vector int);
vsx2	vector unsigned long long vec_vupklsw (vector int);

Glossary

ABI	Application binary interface
BE	Big-endian
CR	Condition Register
DP	Double precision
DSO	Dynamic shared objects
DTV	Dynamic thread vector
DWARF	Debug with arbitrary record format
ELF	Executable and linkable format
FPSCR	Floating-Point Status and Control Register
GOT	Global offset table
GPR	General Purpose Register
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
INF	Infinity
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
LE	Little-endian
LR	Link Register
NaN	Not-a-Number
PIC	Position-independent code
PIE	Position-independent executable
PLT	Procedure linkage table
PMR	Performance Monitor Registers
SIMD	Single instruction, multiple data
SP	Stack pointer
SPR	Special Purpose Register
TCB	Thread control block
TLS	Thread local storage

TOC	Table of contents
TP	Thread pointer
ULP	Unit of least precision
VMX	Vector multimedia extension
VSCR	Vector Status and Control Register
VSX	Vector scalar extension