Power Vector Intrinsic Programming Reference

Workgroup Specification

Revision 1.0.0 (August 11, 2020)

OpenPOWER™

www.openpowerfoundation.org

Power Vector Intrinsic Programming Reference:

System Software Work Group <syssw-chair@openpowerfoundation.org>
OpenPower Foundation

Revision 1.0.0 (August 11, 2020)

Abstract

The purpose of this document is to provide a guide for vector programming on OpenPOWER systems, with an emphasis on examples of best practices. It further provides a reference for intrinsics provided by compliant compilers on OpenPOWER systems.

This document is a Standard Track, Work Group Specification work product owned by the System Software Workgroup and handled in compliance with the requirements outlined in the *OpenPOWER Foundation Work Group (WG) Process* document. It was created using the *Master Template Guide* version 1.0.0. Comments, questions, etc. can be submitted to the public mailing list for this document at <syssw-programming-guides@mailinglist.openpowerfoundation.org>.

Table of Contents

List of Figures

List of Tables

Preface

1. Conventions

The OpenPOWER Foundation documentation uses several typesetting conventions.

Notices

Notices take these forms:

Note

A handy tip or reminder.

Important

Something you must be aware of before proceeding.

Warning

Critical information about the risk of data loss or security issues.

Changes

At certain points in the document lifecycle, knowing what changed in a document is important. In these situations, the following conventions will used.

- *New text will appear like this.* Text marked in this way is completely new.
- ~~Deleted text will appear like this.~~ Text marked in this way was removed from the previous version and will not appear in the final, published document.
- Changed text will appear like this. Text marked in this way appeared in previous versions but has been modified.

Command prompts

In general, examples use commands from the Linux operating system. Many of these are also common with Mac OS, but may differ greatly from the Windows operating system equivalents.

For the Linux-based commands referenced, the following conventions will be followed:

$ prompt Any user, including the `root` user, can run commands that are prefixed with the $ prompt.

 # prompt The `root` user must run commands that are prefixed with the # prompt. You can also prefix these commands with the **sudo** command, if available, to run them.

Document links

Document links frequently appear throughout the documents. Generally, these links include a text for the link, followed by a page number in parenthesis. For example, this link, Preface [ix], references the Preface chapter on page ix.

2. Document change history

This version of the guide replaces and obsoletes all earlier versions.

The following table describes the most recent changes:

Revision Date	Summary of Changes
August 11, 2020	• Version 1.0.0

1. Introduction to Vector Programming on Power

1.1. A Brief History

The history of vector programming on Power processors begins with the AIM (Apple, IBM, Motorola) alliance in the 1990s. The AIM partners developed the Power Vector Media Extension (VMX) to accelerate multimedia applications, particularly image processing. VMX is the name still used by IBM for this instruction set. Freescale (formerly Motorola) used the trademark "AltiVec," while Apple at one time called it "Velocity Engine." While VMX remains the most official name, the term AltiVec is still in common use today. Freescale's AltiVec Technology Programming Interface Manual (the "AltiVec PIM") is still available online for reference (see Section 1.4, “Useful Links” [3]).

The original VMX specification provided for thirty-two 128-bit vector registers (VRs). Each register can be treated as containing sixteen 8-bit character values, eight 16-bit short integer values, four 32-bit integer values, or four 32-bit single-precision floating-point values (the "VMX data types"). Furthermore, the integer data types have signed, unsigned, and boolean variants. An extensive set of arithmetic, logical, comparison, conversion, memory access, and permute class operations were specified to operate on these registers.

The AltiVec PIM documents intrinsic functions to be used by programmers to access the VMX instruction set. Because similar operations are provided for all the VMX data types, the PIM provides for overloaded intrinsics that can operate on different data types. However, such function overloading is not normally acceptable in the C programming language, so compilers compliant with the AltiVec PIM (such as GCC and Clang) were required to add special handling to their parsers to permit this. The PIM suggested (but did not mandate) the use of a header file, `<altivec.h>`, for implementations that provide AltiVec intrinsics. This is common practice for all compliant compilers today.

The first chips incorporating the VMX instruction set were introduced by Freescale in 1999, and used primarily in Apple desktop computers. IBM's last desktop CPU (the PowerPC 970) also included AltiVec support, and was used in the Apple PowerMac G5. IBM initially omitted support for VMX from its server-class computers, but added support for it in the POWER6 server family.

IBM extended VMX by introducing the Vector-Scalar Extension (VSX) for the POWER7 family of processors. VSX adds sixty-four 128-bit vector-scalar registers (VSRs); however, to optimize the amount of per-process register state, the registers overlap with the VRs and the scalar floating-point registers (FPRs) (see Section 1.2, “The Unified Vector Register Set” [2]). The VSRs can represent all the data types representable by the VRs, and can also be treated as containing two 64-bit integers or two 64-bit double-precision floating-point values. However, ISA support for two 64-bit integers in VSRs was limited until Version 2.07 (POWER8) of the Power ISA, and only the VRs are supported for these instructions.

Both the VMX and VSX instruction sets have been expanded for the POWER8 and POWER9 processor families. Starting with POWER8, a VSR can now contain a single 128-bit integer; and starting with POWER9, a VSR can contain a single 128-bit IEEE floating-point value. Again, the ISA currently only supports 128-bit operations on values in the VRs.

The VMX and VSX instruction sets together may be referred to as the Power SIMD (single-instruction, multiple-data) instructions.

1.1.1. Little-Endian Linux

The Power architecture has supported operation in either big-endian (BE) or little-endian (LE) mode from the beginning. However, IBM's Power servers were only shipped with big-endian operating systems (AIX, Linux, i5/OS) prior to the introduction of POWER8. With POWER8, IBM began supporting little-endian Linux distributions for the first time, and introduced a new application binary interface (the 64-Bit ELFv2 ABI Specification Section 1.4, “Useful Links” [3]) that can be used for either big- or little-endian support. In practice, the ELFv2 ABI is currently used only for little-endian Linux.

Although Power has always supported big- and little-endian memory accesses, the introduction of vector register support added a layer of complexity to programming for processors operating in different endian modes. Arrays of elements loaded into a VR or VSR will be indexed from left to right in the register in big-endian mode, but will be indexed from right to left in the register in little-endian mode. However, the VMX and VSX instructions originally assumed that elements will always be indexed from left to right in the register. This is an inconvenience that needs to be hidden from the application programmer wherever possible. To this end, IBM developed a bi-endian vector programming model (see Chapter 2, “The Power Bi-Endian Vector Programming Model” [5]). The intrinsic functions provided for the bi-endian vector programming model are described in Chapter 4, “Vector Intrinsic Reference” [21].

1.2. The Unified Vector Register Set

In OpenPOWER-compliant processors, floating-point and vector operations are implemented using a unified vector-scalar model. As shown in Figure 1.1, “Floating-Point Registers as Part of VSRs” [2] and Figure 1.2, “Vector Registers as Part of VSRs” [3], there are 64 vector-scalar registers; each is 128 bits wide.

Figure 1.1. Floating-Point Registers as Part of VSRs

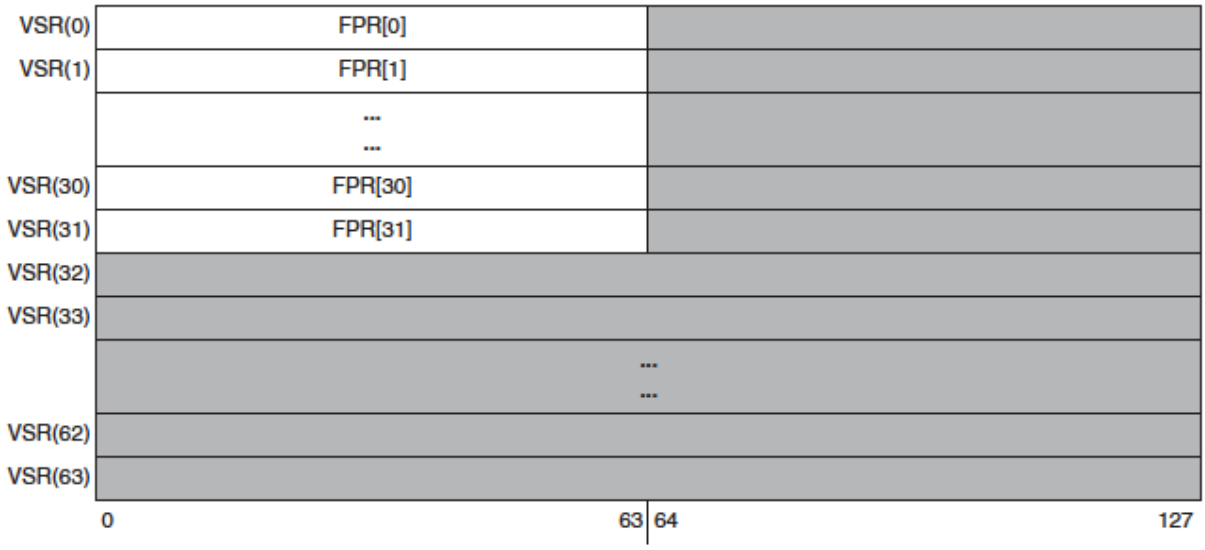
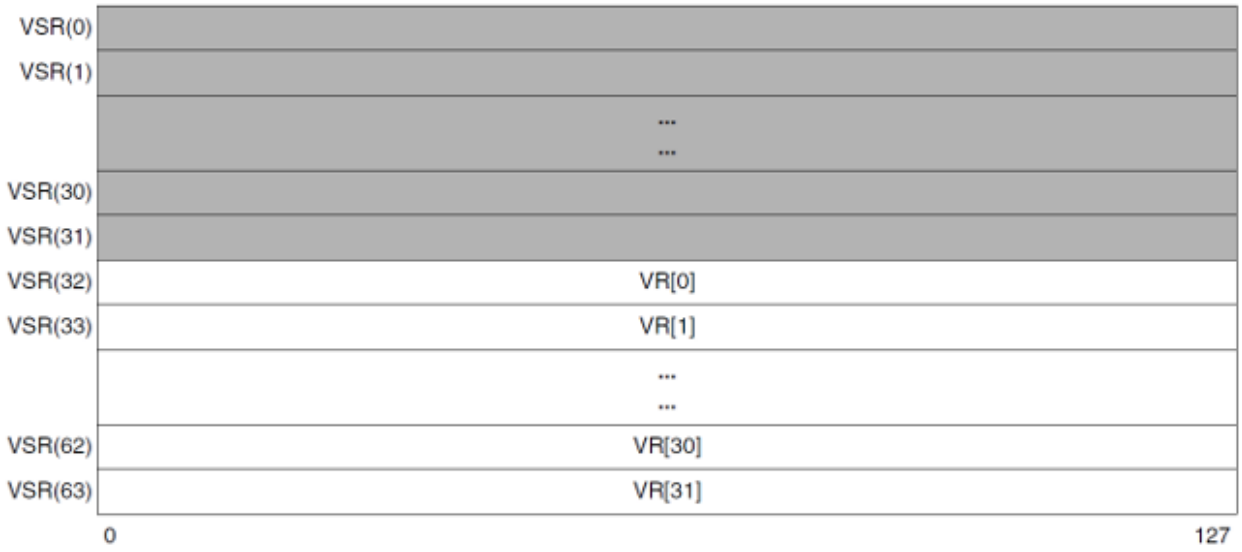


Figure 1.2. Vector Registers as Part of VSRs



The vector-scalar registers can be addressed with VSX 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 these, having 64 bits per register. They can also be addressed with VMX instructions to refer to a 32-register subset of 128-bit registers.

1.3. Where to Report Bugs

This reference provides guidance on using vector intrinsics that are supported by all compatible compilers. If you find a problem when using one of the intrinsics with a compatible compiler, please report a bug! Bug reporting procedures differ depending on which compiler you're using.

- GCC. The reporting procedure for bugs against the GNU Compiler Collection is described at https://gcc.gnu.org/bugs/. The GCC bugzilla tracker is located at https://gcc.gnu.org/bugzilla/.
- Clang/LLVM. The reporting procedure for bugs against the Clang compiler is described at https://llvm.org/docs/HowToSubmitABug.html. The LLVM bug tracking system is located at https://bugs.llvm.org/enter_bug.cgi.
- The XL compilers. For XL compilers provided with the Linux Community Edition, you can provide feedback to the XL compiler team via email (<`compinfo@cn.ibm.com`>); for other editions of XL compilers, please open a Case.

1.4. Useful Links

The following documents provide additional reference materials.

- *64-Bit ELF V2 ABI Specification - Power Architecture.* https://openpowerfoundation.org/?resource_lib=64-bit-elf-v2-abi-specification-power-architecture
- *AltiVec Technology Program Interface Manual.* https://www.nxp.com/docs/en/reference-manual/ALTIVECPIM.pdf

- *Intel Architecture Instruction Set Extensions and Future Features Programming Reference.* https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf
- *Power Instruction Set Architecture*, Version 3.0B Specification. https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0
- *POWER8 Processor User's Manual for the Single-Chip Module.* https://ibm.ent.box.com/s/649rlau0zjcc0yrulqf4cgx5wk3pgbfk
- *POWER9 Processor User's Manual.* https://ibm.ent.box.com/s/tmklq90ze7aj8f4n32er1mu3sy9u8k3k
- *Power Vector Library.* https://github.com/open-power-sdk/pveclib
- *POWER8 In-Core Cryptography: The Unofficial Guide.* https://github.com/noloader/POWER8-crypto/blob/master/power8-crypto.pdf
- *Using the GNU Compiler Collection.* https://gcc.gnu.org/onlinedocs/gcc.pdf
- *GCC's Assembler Syntax.* Felix Cloutier. https://www.felixcloutier.com/documents/gcc-asm.html

1.5. Conformance to this Specification

1. Vector programs on OpenPOWER systems should follow the guide and best practices for vector programming as outlined in Chapter 2, “The Power Bi-Endian Vector Programming Model” [5] and in Chapter 3, “Vector Programming Techniques” [18].
2. Compliant compilers on OpenPOWER systems should provide suitable support for intrinsic functions, preferably as built-in vector functions that translate to one or more Power ISA instructions as described in Chapter 2, “The Power Bi-Endian Vector Programming Model” [5] and in Chapter 4, “Vector Intrinsic Reference” [21]. Compliant compilers targeting a supported ISA level (2.7 or 3.0, for example) should provide support for all intrinsic functions valid for that ISA level, except where an intrinsic function is marked as phased in, deferred, or deprecated.

2. The Power Bi-Endian Vector Programming Model

To ensure portability of applications optimized to exploit the SIMD functions of Power ISA processors, this reference defines a set of functions and data types for SIMD programming. 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 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.

As we've seen, the Power SIMD instructions operate on groups of 1, 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.

Note

Much of the information in this chapter was formerly part of Chapter 6 of the 64-Bit ELF V2 ABI Specification for Power.

2.1. Language Elements

The C and C++ languages are extended to use new identifiers `vector`, `pixel`, `bool`, `__vector`, `__pixel`, and `__bool`. These keywords are used to specify vector data types (Section 2.2, “Vector Data Types” [6]). Because these identifiers may conflict with keywords in more recent language standards for C and C++, compilers may implement these in one of two ways.

- `__vector`, `__pixel`, `__bool`, and `bool` are defined as keywords, with `vector` and `pixel` as predefined macros that expand to `__vector` and `__pixel`, respectively.
- `__vector`, `__pixel`, and `__bool` are defined as keywords in all contexts, while `vector`, `pixel`, and `bool` are treated as keywords only within the context of a type declaration.

As a motivating example, the **vector** token is used as a type in the C++ Standard Template Library, and hence cannot be used as an unrestricted keyword, but can be used in the context-sensitive

implementation. For example, **vector char** is distinct from **std::vector** in the context-sensitive implementation.

Vector literals may be specified using a type cast and a set of literal initializers in parentheses or braces. For example,

```
vector int x = (vector int) (4, -1, 3, 6);
vector double g = (vector double) { 3.5, -24.6 };
```

Current C compilers do not support literals for `__int128` types. A `vector __int128` constant can be constructed from smaller literals with appropriate cast-shift-or logic. For example,

```
vector unsigned __int128 x = { (((unsigned __int128)0x1020304050607080) << 64) |
 0x90A0B0C0D0E0F000 };
```

2.2. Vector Data Types

Languages provide support for the data types in Table 2.1, “Vector Types” [7] to represent vector data types stored in vector registers.

For the C and C++ programming languages (and related/derived languages), the "Power SIMD C Types" listed in the leftmost column of Table 2.1, “Vector Types” [7] may be used when Power SIMD language extensions are enabled. Either `vector` or `__vector` may be used in the type name. Note that the ELFv2 ABI for Power also includes a `vector _Float16` data type. As of this writing, no current compilers for Power have implemented such a type. This document does not include that type or any intrinsics related to it.

For the Fortran language, Table 2.4, “Fortran Vector Data Types” [14] gives a correspondence between Fortran and C/C++ language types.

The assignment operator always performs a byte-by-byte data copy for vector data types.

Like other C/C++ language types, vector types may be defined to have const or volatile properties. Vector data types can be defined as being in static, auto, and register storage.

Pointers to vector types are defined like pointers of other C/C++ types. Pointers to vector objects may be defined to have const and volatile properties. Pointers to vector objects must be addresses divisible by 16, as vector objects are always aligned on quadword (16-byte, or 128-bit) boundaries.

The preferred way to access vectors at an application-defined address is by using vector pointers and the C/C++ dereference operator `*`. Similar to other C/C++ data types, the array reference operator `[]` may be used to access vector objects with a vector pointer with the usual definition to access the *N*th vector element from a vector pointer. The dereference operator `*` may *not* be used to access data that is not aligned at least to a quadword boundary. Built-in functions such as `vec_xl` and `vec_xst` and provided for unaligned data access. Please refer to Section 2.7.1, “Unaligned vector access” [15] for an example.

One vector type may be cast to another vector type without restriction. Such a cast is simply a reinterpretation of the bits, and does not change the data. There are no default conversions for vector types.

Compilers are expected to recognize and optimize multiple operations that can be optimized into a single hardware instruction. For example, a load-and-splat hardware instruction (such as **lxvdsx**) might be generated for the following sequence:

```
double *double_ptr;
register vector double vd = vec_splats(*double_ptr);
```

Table 2.1. Vector Types

Power SIMD C Types	sizeof	Alignment	Description
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 pixel	16	Quadword	Vector of 8 halfwords, each interpreted as a 1-bit channel and three 5-bit channels.
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[a] vector unsigned long long	16	Quadword	Vector of 2 unsigned doublewords.
vector signed long[a] vector signed long long	16	Quadword	Vector of 2 signed doublewords.
vector bool long[a] vector bool long long	16	Quadword	Vector of 2 doublewords with a value of either 0 or $2^{64} - 1$.
vector unsigned __int128	16	Quadword	Vector of 1 unsigned quadword.
vector signed __int128	16	Quadword	Vector of 1 signed quadword.
vector float	16	Quadword	Vector of 4 single-precision floats.
vector double	16	Quadword	Vector of 2 double-precision floats.

[a]The vector long types are deprecated due to their ambiguity between 32-bit and 64-bit environments. The use of the vector long long types is preferred.

2.3. Vector Operators

In addition to the dereference and assignment operators, the Power Bi-Endian Vector Programming Model provides the usual operators that are valid on pointers; these operators are also valid for pointers to vector types.

The traditional C/C++ unary operators (+ -, and ~), are defined on vector types. The traditional C/C++ binary operators (+, -, *, %, /, shift, logical, and comparison) and the ternary operator (?:) are defined on like vector types. Other than ?:, these operators perform their operations "elementwise" on the base elements of the operands, as follows.

For unary operators, the specified operation is performed on each base element of the single operand to derive the result value placed into the corresponding element of the vector result. The result type of unary operations is the type of the single operand. For example,

```
vector signed int a, b;
a = -b;
```

produces the same result as

```
vector signed int a, b;
a = vec_neg (b);
```

For binary operators, the specified operation is performed on corresponding base elements of both operands to derive the result value for each vector element of the vector result. Both operands of the binary operators must have the same vector type with the same base element type. The result of binary operators is the same type as the type of the operands. For example,

```
vector signed int a, b;
a = a + b;
```

produces the same result as

```
vector signed int a, b;
a = vec_add (a, b);
```

For the ternary operator (`?:`), the first operand must be an integral type, used to select between the second and third operands which must be of the same vector type. The result of the ternary operator will also have that type. For example,

```
int test_value;
vector signed int a, b, r;
r = test_value ? a : b;
```

produces the same result as

```
int test_value;
vector signed int a, b, r;
if (test_value)
  r = a;
else
  r = b;
```

Further, the array reference operator may be applied to vector data types, yielding an l-value corresponding to the specified element in accordance with the vector element numbering rules (see Section 2.4, “Vector Layout and Element Numbering” [8]). An l-value may either be assigned a new value or accessed for reading its value. For example,

```
vector signed int a;
signed int b, c;
b = a[0];
a[3] = c;
```

2.4. 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. To understand how vector elements are represented in memory and in registers, it is best to start with some simple concepts of endianness.

Figure 2.1. Scalar Quantities and Endianness

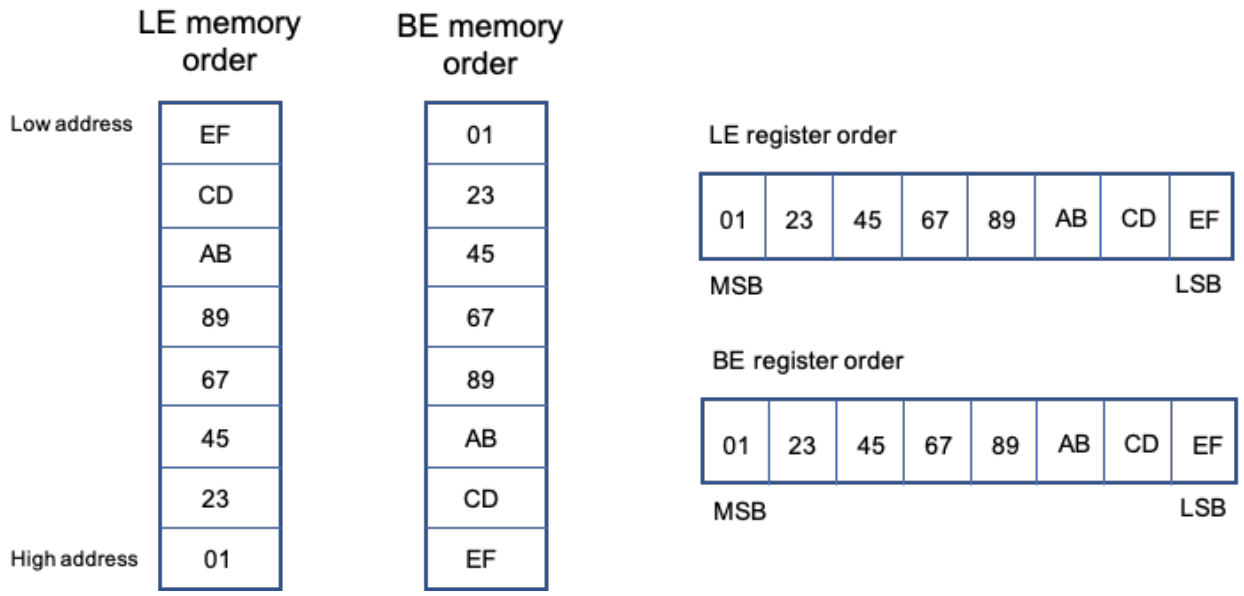
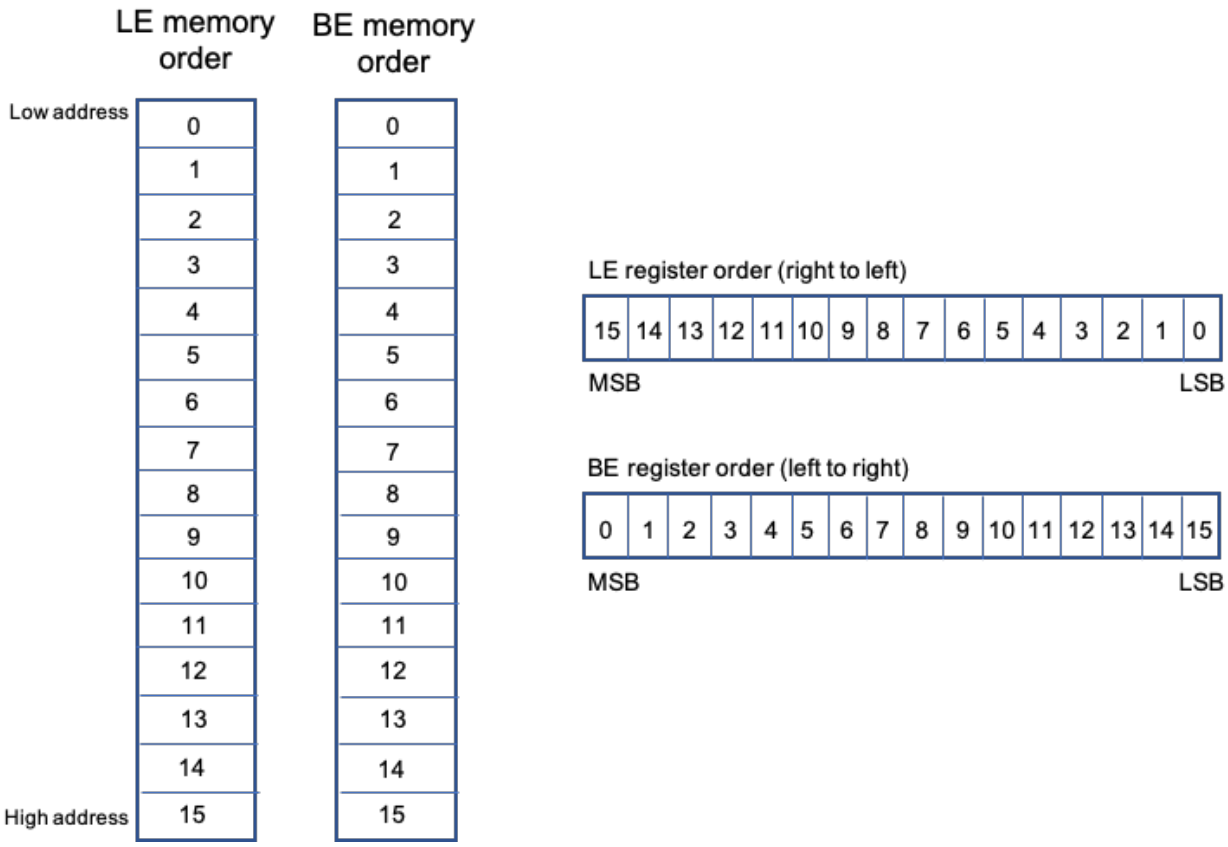


Figure 2.1, “Scalar Quantities and Endianness” [9] shows different representations of a 64-bit scalar integer with the hexadecimal value `0x0123456789ABCDEF`. We say that the most significant byte (MSB) of this value is `0x01`, and its least significant byte (LSB) is `0xEF`. The scalar value is stored using eight bytes of memory. On a little-endian (LE) system, the LSB is stored at the lowest address of these eight bytes, and the MSB is stored at the highest address. On a big-endian (BE) system, the MSB is stored at the lowest address of these eight bytes, and the LSB is stored at the highest address. Regardless of the memory order, the register representation of the scalar value is identical; the MSB is located on the "left" end of the register, and the LSB is located on the "right" end.

Of course, the concept of "left" and "right" is a useful fiction; there is no guarantee that the circuitry of a hardware register is laid out this way. However, we will see, as we deal with vector elements, that the concepts of left and right are more natural for human understanding than byte and element significance. Indeed, most programming languages have operators, such as shift-left and shift-right, that use this same terminology.

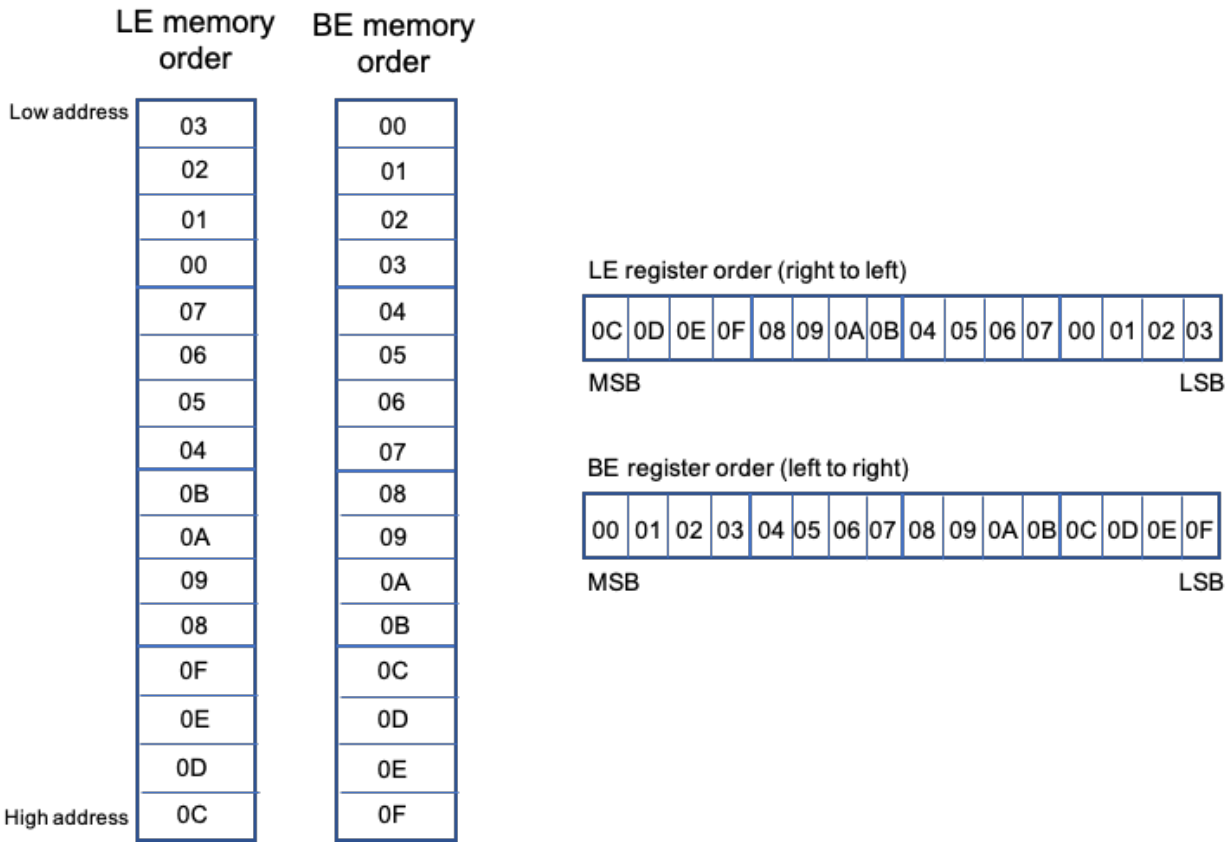
Let's move from scalars to arrays, which are more interesting to us since we can use vector registers to operate on arrays, or portions of larger arrays. Suppose we have an array of bytes with values 0 through 15, as shown in Figure 2.2, “Byte Arrays and Endianness” [10]. Note that each byte is a separate data element with only one possible representation in memory, so the array of bytes looks identical in memory, regardless of whether we are using a BE system or an LE system. But when we load these 16 bytes into a vector register, perhaps by using the ISA 3.0 **lxv** instruction, the byte at the lowest address on an LE system will be placed in the LSB of the vector register, but on a BE system will be placed in the MSB of the vector register. Thus the array elements appear "right to left" in the register on an LE system, and "left to right" in the register on a BE system.

Figure 2.2. Byte Arrays and Endianness



Things become even more interesting when we consider arrays of larger elements. In Figure 2.3, "Word Arrays and Endianness" [11], we see the layout of an array of four 32-bit integers, where the 0th element has hexadecimal value `0x00010203`, the 1st element has value `0x04050607`, the 2nd element has value `0x08090A0B`, and the 3rd element has value `0x0C0D0E0F`. The order of the array elements in memory is the same for both LE and BE systems; but the layout of each element itself is reversed. When the **lxv** instruction is used to load the memory into a vector register, again the low address is loaded into the LSB of the register for LE, but loaded into the MSB of the register for BE. The effect is that the array elements again appear right-to-left on a LE system and left-to-right on a BE system. Note that each 32-bit element of the array has its most significant bit "on the left" whether a LE or BE system is in use. This is of course necessary for proper arithmetic to be performed on the array elements by vector instructions.

Figure 2.3. Word Arrays and Endianness



Thus on a BE system, we number vector elements starting with 0 on the left, while on an LE system, we number vector elements starting with 0 on the right. We will informally refer to these as big-endian and little-endian vector element numberings and vector layouts.

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.

Application programs may query the vector element ordering in use by testing the __VEC_ELEMENT_REG_ORDER__ macro. This macro has two possible values:

__ORDER_LITTLE_ENDIAN__	Vector elements use little-endian element ordering.
__ORDER_BIG_ENDIAN__	Vector elements use big-endian element ordering.

This is no longer as useful as it once was. The primary use case was for big-endian vector layout in little-endian environments, which is now deprecated as discussed in Section 2.5.2, “Big-Endian Vector Layout in Little-Endian Environments (Deprecated)” [13]. It's generally equivalent to test for `__BIG_ENDIAN__` or `__LITTLE_ENDIAN__`.

Note

Remember that each element in a vector has the same representation in both big- and little-endian element orders. That is, an `int` is always 32 bits, with the sign bit in the

high-order position. Programmers must be aware of this when programming with mixed data types, such as an instruction that multiplies two `short` elements to produce an `int` element. Always access entire elements to avoid potential endianness issues.

2.5. Vector Built-In Functions

Some of the Power SIMD 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, `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.

The descriptions of the built-in functions in Chapter 4, “Vector Intrinsic Reference” [21] contain notes on endian issues that apply to each built-in function. Furthermore, a built-in function requiring a different compiler implementation for big-endian than it uses for little-endian has a sample compiler implementation for both BE and LE. These sample implementations are only intended as examples; designers of a compiler are free to use other methods to implement the specified semantics.

Of course, most built-in functions operate only on corresponding sets of elements of input vectors to produce output vectors, and thus are not "endian-sensitive." A complete list of endian-sensitive built-in functions can be found in Table 2.2, “Endian-Sensitive Built-In Functions” [12].

Table 2.2. Endian-Sensitive Built-In Functions

vec_bperm	vec_mergeh	vec_signedo
vec_cipher_be	vec_mergel	vec_sld
vec_cipherlast_be	vec_mergeo	vec_sldw
vec_doublee	vec_mfvscr	vec_sll
vec_doubleh	vec_mule	vec_slo
vec_doublel	vec_mulo	vec_slv
vec_doubleo	vec_ncipher_be	vec_splat
vec_extract	vec_ncipherlast_be	vec_srl
vec_extract_fp32_from_shorth	vec_pack	vec_sro
vec_extract_fp32_from_shortl	vec_pack_to_short_fp32	vec_srv
vec_extract4b	vec_packpx	vec_sum2s
vec_first_match_index	vec_packs	vec_sums
vec_first_match_or_eos_index	vec_packsu	vec_unpackh
vec_first_mismatch_index	vec_perm	vec_unpackl
vec_first_mismatch_or_eos_index	vec_permxor	vec_unsigned2
vec_float2	vec_pmsum_be	vec_unsignede
vec_floate	vec_reve	vec_unsignedo

vec_floato	vec_sbox_be	vec_xl (ISA 2.07 only)
vec_insert	vec_shasigma_be	vec_xl_be
vec_insert4b	vec_signed2	vec_xst (ISA 2.07 only)
vec_mergee	vec_signede	vec_xst_be

2.5.1. Extended Data Movement Functions

The built-in functions in Table 2.3, “VMX Memory Access Built-In Functions” [13] map to Altivec/VMX load and store instructions and provide access to the “auto-aligning” memory instructions of the VMX ISA where low-order address bits are discarded before performing a memory access. These instructions 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 operation during code generation.

Before the bi-endian programming model was introduced, the `vec_lvsl` and `vec_lvsr` intrinsics were supported. These could be used in conjunction with `vec_perm` and VMX load and store instructions for unaligned access. The `vec_lvsl` and `vec_lvsr` interfaces are deprecated in accordance with the interfaces specified here. For compatibility, the built-in pseudo sequences published in previous VMX documents continue to work with little-endian data layout and the little-endian vector layout described in this document. However, the use of these sequences in new code is discouraged and usually results in worse performance. It is recommended that compilers issue a warning when these functions are used in little-endian environments.

Table 2.3. VMX 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.

Instead, it is recommended that programmers use the `vec_xl` and `vec_xst` vector built-in functions to access unaligned data streams. See the descriptions of these instructions in Chapter 4, “Vector Intrinsic Reference” [21] for further description and implementation details.

2.5.2. Big-Endian Vector Layout in Little-Endian Environments (Deprecated)

Versions 1.0 through 1.4 of the 64-Bit ELFv2 ABI Specification for Power provided for optional compiler support for using big-endian element ordering in little-endian environments. This was initially deemed useful for porting certain libraries that assumed big-endian element ordering regardless of the endianness of their input streams. In practice, this introduced serious compiler complexity without much utility. Thus this support (previously controlled by switches `-maltivec=be` and/or `-qaltivec=be`) is now deprecated. Current versions of the GCC and Clang open-source compilers do not implement this support.

2.6. Language-Specific Vector Support for Other Languages

2.6.1. Fortran

Table 2.4, “Fortran Vector Data Types” [14] shows the correspondence between the C/C++ types described in this document and their Fortran equivalents. In Fortran, the Boolean vector data types are represented by `VECTOR(UNSIGNED(`*n*`))`.

Table 2.4. Fortran Vector Data Types

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[a]
VECTOR(INTEGER(16))	vector signed __int128
VECTOR(UNSIGNED(1))	vector unsigned char
VECTOR(UNSIGNED(2))	vector unsigned short
VECTOR(UNSIGNED(4))	vector unsigned int
VECTOR(UNSIGNED(8))	vector unsigned long long, vector unsigned long[a]
VECTOR(UNSIGNED(16))	vector unsigned __int128
VECTOR(REAL(4))	vector float
VECTOR(REAL(8))	vector double
VECTOR(PIXEL)	vector pixel

[a]The vector long types are deprecated due to their ambiguity between 32-bit and 64-bit environments. The use of the vector long long types is preferred.

Because the Fortran language does not support pointers, vector built-in functions that expect pointers to a base type take an array element reference to indicate the address of a memory location that is the subject of a memory access built-in function.

Because the Fortran language does not support type casts, the `vec_convert` and `vec_concat` built-in functions shown in Table 2.5, “Built-In Vector Conversion Functions” [14] are provided to perform bit-exact type conversions between vector types.

Table 2.5. Built-In Vector Conversion Functions

Group	Description
VEC_CONCAT (ARG1, ARG2) (Fortran)	Purpose: Concatenates two elements to form a vector. Result value: The resulting vector consists of the two scalar elements, ARG1 and ARG2, assigned to elements 0 and 1 (using the environment’s native endian numbering), respectively. • **Note:** This function corresponds to the C/C++ vector constructor (vector type){a,b}. It is provided only for languages without vector constructors.
	vector signed long long vec_concat (signed long long, signed long long);

Group	Description
	vector unsigned long long vec_concat (unsigned long long, unsigned long long);
	vector double vec_concat (double, double);
VEC_CONVERT(V, MOLD)	Purpose: Converts a vector to a vector of a given type. Class: Pure function Argument type and attributes: • V Must be an INTENT(IN) vector. • MOLD Must be an INTENT(IN) vector. If it is a variable, it need not be defined. Result type and attributes: The result is a vector of the same type as MOLD. Result value: The result is as if it were on the left-hand side of an intrinsic assignment with V on the right-hand side.

2.7. Examples and Limitations

2.7.1. Unaligned vector access

A common programming error is to cast a pointer to a base type (such as `int`) to a pointer of the corresponding vector type (such as `vector int`), and then dereference the pointer. This constitutes undefined behavior, because it casts a pointer with a smaller alignment requirement to a pointer with a larger alignment requirement. Compilers may not produce code that you expect in the presence of undefined behavior.

Thus, do not write the following:

```
int a[4096];
vector int x = *((vector int *) a);
```

Instead, write this:

```
int a[4096];
vector int x = vec_xl (0, a);
```

2.7.2. vec_sld and vec_sro are not bi-endian

One oddity in the bi-endian vector programming model is that `vec_sld` has big-endian semantics for code compiled for both big-endian and little-endian targets. That is, any code that uses `vec_sld` without guarding it with a test on endianness is likely to be incorrect.

At the time that the bi-endian model was being developed, it was discovered that existing code in several Linux packages was using `vec_sld` in order to perform multiplies, or to otherwise shift portions of base elements left. A straightforward little-endian implementation of `vec_sld` would concatenate the two input vectors in reverse order and shift bytes to the right. This would only give compatible results for `vector char` types. Those using this intrinsic as a cheap multiply, or to shift bytes within larger elements, would see different results on little-endian versus big-endian with such

an implementation. Therefore it was decided that `vec_sld` would not have a bi-endian implementation.

`vec_sro` is not bi-endian for similar reasons.

2.7.3. Limitations on bi-endianness of vec_perm

The `vec_perm` intrinsic is bi-endian, provided that it is used to reorder entire elements of the input vectors.

To see why this is, let's examine the code generation for

```
vector int t;
vector int a = (vector int){0x00010203, 0x04050607, 0x08090a0b, 0x0c0d0e0f};
vector int b = (vector int){0x10111213, 0x14151617, 0x18191a1b, 0x1c1d1e1f};
vector char c = (vector char){0,1,2,3,28,29,30,31,12,13,14,15,20,21,22,23};
t = vec_perm (a, b, c);
```

For big endian, a compiler should generate:

```
vperm  t,a,b,c
```

For little endian targeting a POWER8 system, a compiler should generate:

```
vnand  d,c,c
vperm  t,b,a,d
```

For little endian targeting a POWER9 system, a compiler should generate:

```
vpermr  t,b,a,c
```

Note that the `vpermr` instruction takes care of modifying the permute control vector (PCV) c that was done using the `vnand` instruction for POWER8. Because only the bottom 5 bits of each element of the PCV are read by the hardware, this has the effect of subtracting the original elements of the PCV from 31.

Note also that the PCV c has element values that are contiguous in groups of 4. This selects entire elements from the input vectors a and b to reorder. Thus the intent of the code is to select the first integer element of a, the last integer element of b, the last integer element of a, and the second integer element of b, in that order.

The big endian result is {0x00010203, 0x1c1d1e1f, 0x0c0d0e0f, 0x14151617}, as shown here:

a	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
b	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
c	0	1	2	3	28	29	30	31	12	13	14	15	20	21	22	23
t	00	01	02	03	1C	1D	1E	1F	0C	0D	0E	0F	14	15	16	17

For little endian, the modified PCV is elementwise subtracted from 31, giving {31,30,29,28,3,2,1,0,19,18,17,16,11,10,9,8}. Since the elements appear in reverse order in a register when loaded from little-endian memory, the elements appear in the register from left to right as {8,9,10,11,16,17,18,19,0,1,2,3,28,29,30,31}. So the following `vperm` instruction will again select entire elements using the groups of 4 contiguous bytes, and the values of the integers will be reordered without compromising each integer's contents. The little-endian result matches the big-endian result, as shown. Observe that **a** and **b** switch positions for little endian code generation.

b	1C	1D	1E	1F	18	19	1A	1B	14	15	16	17	10	11	12	13
a	0C	0D	0E	0F	08	09	0A	0B	04	05	06	07	00	01	02	03
c	8	9	10	11	16	17	18	19	0	1	2	3	28	29	30	31
t	14	15	16	17	0C	0D	0E	0F	1C	1D	1E	1F	00	01	02	03

Now, suppose instead that the original PCV does not reorder entire integers at once:

```
vector char c = (vector char){0,20,31,4,7,17,6,19,30,3,2,8,9,13,5,22};
```

The result of the big-endian implementation would be:

```
t = {0x00141f04, 0x07110613, 0x1e030208, 0x090d0516};
```

a	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
b	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
c	0	20	31	4	7	17	6	19	30	3	2	8	9	13	5	22
t	00	14	1F	04	07	11	06	13	1E	03	02	08	09	0D	05	16

For little-endian, the modified PCV would be {31,11,0,27,24,14,25,12,1,28,29,23,22,18,26,9}, appearing in the register as {9,26,18,22,23,29,28,1,12,25,14,24,27,0,11,31}. The final little-endian result would be

```
t = {0x071c1703, 0x10051204, 0x0b01001d, 0x15060e0a};
```

which bears no resemblance to the big-endian result.

b	1C	1D	1E	1F	18	19	1A	1B	14	15	16	17	10	11	12	13
a	0C	0D	0E	0F	08	09	0A	0B	04	05	06	07	00	01	02	03
c	9	26	18	22	23	29	28	1	12	25	14	24	27	0	11	31
t	15	06	0E	0A	0B	01	00	1D	10	05	12	04	07	1C	17	03

The lesson here is to only use `vec_perm` to reorder entire elements of a vector. If you must use vec_perm for another purpose, your code must include a test for endianness and separate algorithms for big- and little-endian. Examples of this may be seen in the Power Vector Library project (see Section 1.4, “Useful Links” [3]).

3. Vector Programming Techniques

3.1. Help the Compiler Help You

The best way to use vector intrinsics is often *not to use them at all*.

This may seem counterintuitive at first. Aren't vector intrinsics the best way to ensure that the compiler does exactly what you want? Well, sometimes. But the problem is that the best instruction sequence today may not be the best instruction sequence tomorrow. As the Power ISA moves forward, new instruction capabilities appear, and the old code you wrote can easily become obsolete. Then you start having to create different versions of the code for different levels of the Power ISA, and it can quickly become difficult to maintain.

Most often programmers use vector intrinsics to increase the performance of loop kernels that dominate the performance of an application or library. However, modern compilers are often able to optimize such loops to use vector instructions without having to resort to intrinsics, using an optimization called autovectorization (or auto-SIMD). Your first focus when writing loop kernels should be on making the code amenable to autovectorization by the compiler. Start by writing the code naturally, using scalar memory accesses and data operations, and see whether the compiler autovectorizes your code. If not, here are some steps you can try:

- Check your optimization level. Different compilers enable autovectorization at different optimization levels. For example, at this writing the GCC compiler requires `-O3` to enable autovectorization by default.

- Consider using `-ffast-math`. This option assumes that certain fussy aspects of IEEE floating-point can be ignored, such as the presence of Not-a-Numbers (NaNs), signed zeros, and so forth. `-ffast-math` may also affect precision of results that may not matter to your application. Turning on this option can simplify the control flow of loops generated for your application by removing tests for NaNs and so forth. (Note that `-Ofast` turns on both -O3 and -ffast-math in GCC.)

- Align your data wherever possible. For most effective auto-vectorization, arrays of data should be aligned on at least a 16-byte boundary, and pointers to that data should be identified as having the appropriate alignment. For example:

  ```
  float fdata[4096] __attribute__((aligned(16)));
  ```

  ensures that the compiler can use an efficient, aligned vector load to bring data from `fdata` into a vector register. Autovectorization will appear more profitable to the compiler when data is known to be aligned.

  You can also declare pointers to point to aligned data, which is particularly useful in function arguments:

  ```
  void foo (__attribute__((aligned(16))) double * aligned_fptr)
  ```

- Tell the compiler when data can't overlap. In C and C++, use of pointers can cause compilers to pessimistically analyze which memory references can refer to the same memory. This can prevent important optimizations, such as reordering memory references, or keeping previously loaded values in memory rather than reloading them. Inefficiently optimized scalar loops are less likely to be autovectorized. You can annotate your pointers with the `restrict` or

`__restrict__` keyword to tell the compiler that your pointers don't "alias" with any other memory references. (`restrict` can be used only in C when compiling for the C99 standard or later. `__restrict__` is a language extension, available in GCC, Clang, and the XL compilers, that can be used without restriction for both C and C++. See your compiler's user manual for details.)

Suppose you have a function that takes two pointer arguments, one that points to data your function writes to, and one that points to data your function reads from. By default, the compiler may believe that the data being read and written could overlap. To disabuse the compiler of this notion, do the following:

```
void foo (double *__restrict__ outp, double *__restrict__ inp)
```

3.2. Use Portable Intrinsics

If you can't convince the compiler to autovectorize your code, or you want to access specific processor features not appropriate for autovectorization, you should use intrinsics. However, you should go out of your way to use intrinsics that are as portable as possible, in case you need to change compilers in the future.

This reference provides intrinsics that are guaranteed to be portable across compliant compilers. In particular, both the GCC and Clang compilers for Power implement the intrinsics in this manual. The compilers may each implement many more intrinsics, but the ones in this manual are the only ones guaranteed to be portable. So if you are using an interface not described here, you should look for an equivalent one in this manual and change your code to use that.

Where an intrinsic may not be available from all compilers or at all ISA levels, this information is called out in the description of the intrinsic in Section 4.2, “Built-In Vector Functions” [23].

There are also other vector APIs that may be of use to you (see Section 3.4, “Other Vector Programming APIs” [20]). In particular, the Power Vector Library (see Section 3.4.2, “The Power Vector Library (pveclib)” [20]) provides additional portability across compiler and ISA versions, as well as interfaces that hide cases where assembly language is needed.

3.3. Use Assembly Code Sparingly

Sometimes the compiler will absolutely not cooperate in giving you the code you need. You might not get the instruction you want, or you might get extra instructions that are slowing down your ideal performance. When that happens, the first thing you should do is report this to the compiler community! This will allow them to get the problem fixed in the next release of the compiler. See Section 1.3, “Where to Report Bugs” [3] if you need to report an issue.

In the meanwhile, though, what are your options? As a workaround, your best option may be to use assembly code. There are two ways to go about this. Using inline assembly is generally appropriate only for very small snippets of code (1-5 instructions, say). If you want to write a whole function in assembly code, though, it is better to create a separate `.s` or `.S` file. The only difference in these two file types is that a `.S` file will be processed by the C preprocessor before being assembled.

Assembly programming is beyond the scope of this manual. Getting inline assembly correct can be quite tricky, and it is best to look at existing examples to learn how to use it properly. However, there is a good introduction to inline assembly in *Using the GNU Compiler Collection*, in section 6.47 at

the time of this writing. Felix Cloutier has also written a very nice guide. See Section 1.4, “Useful Links” [3].

If you write a function entirely in assembly, you are responsible for following the calling conventions established by the ABI (see Section 1.4, “Useful Links” [3]). Again, it is best to look at examples. One place to find well-written `.S` files is in the GLIBC project. You can also study the assembly output from your favorite compiler, which can be obtained with the `-S` or similar option, or by using the **objdump** utility.

3.4. Other Vector Programming APIs

In addition to the intrinsic functions provided in this reference, programmers should be aware of other vector programming API resources.

3.4.1. x86 Vector Portability Headers

Recent versions of the GCC and Clang open-source compilers for Power provide "drop-in" portability headers for portions of the Intel Architecture Instruction Set Extensions (see Section 1.4, “Useful Links” [3]). These headers mirror the APIs of Intel headers having the same names. As of this writing, support is provided for the MMX and SSE layers, up through SSE3 and portions of SSE4. No support for the AVX layers is envisioned. The portability headers are available starting with GCC 8.1 and Clang 9.0.0.

The portability headers provide the same semantics as the corresponding Intel APIs, but using VMX and VSX instructions to emulate the Intel vector instructions. It should be emphasized that these headers are provided for portability, and will not necessarily perform optimally (although in many cases the performance is very good). Using these headers is often a good first step in porting a library using Intel intrinsics to Power, after which more detailed rewriting of algorithms is usually desirable for best performance.

Access to the portability APIs occurs automatically when including one of the corresponding Intel header files, such as `<mmintrin.h>`.

3.4.2. The Power Vector Library (pveclib)

The Power Vector Library, also known as `pveclib`, is a separate project available from github (see Section 1.4, “Useful Links” [3]). The `pveclib` project builds on top of the intrinsics described in this manual to provide higher-level vector interfaces that are highly portable. The goals of the project include:

- Providing equivalent functions across versions of the Power ISA. For example, the *Vector Multiply-by-10 Unsigned Quadword* operation introduced in Power ISA 3.0 (POWER9) can be implemented using a few vector instructions on earlier Power ISA versions.
- Providing equivalent functions across compiler versions. For example, intrinsics provided in later versions of the compiler can be implemented as inline functions with inline asm in earlier compiler versions.
- Providing higher-order functions not provided directly by the Power ISA. One example is a vector SIMD implementation for ASCII `__isalpha` and similar functions. Another example is full `__int128` implementations of *Count Leading Zeroes*, *Population Count*, and *Multiply*.

4. Vector Intrinsic Reference

4.1. How to Use This Reference

This chapter contains reference material for each supported vector intrinsic. The information for each intrinsic includes:

- The intrinsic name and extended name;
- A type-free example of the intrinsic's usage;
- A description of the intrinsic's purpose;
- A description of the value(s) returned from the intrinsic, if any;
- A description of any unusual characteristics of the intrinsic when different target endiannesses are in force. If the semantics of the intrinsic in big-endian and little-endian modes are identical, the description will read "None.";
- Optionally, additional explanatory notes about the intrinsic; and
- A table of supported type signatures for the intrinsic.

Most intrinsics are overloaded, supporting multiple type signatures. The types of the input arguments always determine the type of the result argument; that is, it is not possible to define two intrinsic overloads with the same input argument types and different result argument types.

The type-free example of the intrinsic's usage uses the convention that **r** represents the result of the intrinsic, and **a**, **b**, etc., represent the input arguments. The allowed type combinations of these variables are shown as rows in the table of supported type signatures.

Each row contains at least one example implementation. This shows one way that a conforming compiler might achieve the intended semantics of the intrinsic, but compilers are not required to generate this code specifically. The letters **r**, **a**, **b**, etc., in the examples represent vector registers containing the values of those variables. The letters **t**, **u**, etc., represent vector registers containing temporary intermediate results. The same register is assumed to be used for each instance of one of these letters.

When implementations differ for big- and little-endian targets, separate example implementations are shown for each endianness.

The implementations show a sequence of instructions that may be used in the implementation of a particular intrinsic, and usually include vector instructions. When trying to determine which intrinsic to use, it can be useful to have a cross-reference from a specific vector instruction to the intrinsics whose implementations make use of it. This manual contains such a cross-reference (Chapter 5, “Instruction/Intrinsic Cross-Reference” [257]) for the programmer's convenience.

For some intrinsics, restrictions are shown in the implementation table for some of the rows. Possible restrictions include:

- **ISA 3.0 or later**. This form is only available starting with PowerISA 3.0, corresponding to POWER9 servers. The Power Vector Library (see Section 1.4, “Useful Links” [3] provides

equivalent POWER7/POWER8 implementations for many ISA 3.0 vector instructions, which may be preferred for portability.

- **Deprecated**. This form of the intrinsic is currently available, but may be removed in the future. Programmers are discouraged from using it.

4.1.1. Terminology

Some intrinsic descriptions indicate that either *modular* arithmetic or *saturating* arithmetic is used. This refers to what happens when an operation overflows the number of available bits. A modular operation that overflows truncates the result on the left, also known as wrapping the result. A saturating operation that overflows produces the largest (or smallest) possible result representable in the output element type.

Operands are sometimes represented as having a `const int` type. In such cases, the programmer is expected to provide an integer literal. When the literal has specific required bounds, this is often represented instead by such phrases as "5-bit signed literal" or "2-bit unsigned literal" to specify them. In such cases, compilers are encouraged to at least warn upon detecting an out-of-range value. Providing a variable when a literal is required is a compile-time error.

4.2. Built-In Vector Functions

vec_abs

Vector Absolute Value

```
r = vec_abs (a)
```

Purpose: Returns a vector **r** that contains the absolute values of the contents of the vector **a**.

Result value: The value of each element of **r** is the absolute value of the corresponding element of **a**. For integer vectors, the arithmetic is modular.

Endian considerations: None.

Table 4.1. Supported type signatures for vec_abs

r	a	Example Implementation

vector signed char	vector signed char	`vspltisw t,0`
		`vsububm t,t,a`
		`vmaxsb r,t,a`

vector signed short	vector signed short	`vspltisw t,0`
		`vsubuhm t,t,a`
		`vmaxsh r,t,a`

vector signed int	vector signed int	`vspltisw t,0`
		`vsubuwm t,t,a`
		`vmaxsw r,t,a`

vector signed long long	vector signed long long	`vspltisw t,0`
		`vsubudm t,t,a`
		`vmaxsd r,t,a`

vector float	vector float	`xvabssp r,a`
vector double	vector double	`xvabsdp r,a`

vec_absd

Vector Absolute Difference

```
r = vec_absd (a, b)
```

Purpose: Computes the absolute difference of two vectors.

Result value: The value of each element of **r** is the absolute difference of the corresponding elements of **a** and **b**, using modular arithmetic.

Endian considerations: None.

Table 4.2. Supported type signatures for vec_absd

r	a	b	Example Implementation	Restrictions
vector unsigned char	vector unsigned char	vector unsigned char	`vabsdub r,a,b`	ISA 3.0 or later
vector unsigned short	vector unsigned short	vector unsigned short	`vabsduh r,a,b`	ISA 3.0 or later
vector unsigned int	vector unsigned int	vector unsigned int	`vabsduw r,a,b`	ISA 3.0 or later

vec_abss

Vector Absolute Value Saturated

```
r = vec_abss (a)
```

Purpose: Returns a vector **r** that contains the saturated absolute values of the contents of the vector **a**.

Result value: The value of each element of **r** is the saturated absolute value of the corresponding element of **a**.

Endian considerations: None.

Table 4.3. Supported type signatures for vec_abss

r	a	Example Implementation

vector signed char	vector signed char	`vspltisb t,0`
		`vsubsbs t,t,a`
		`vmaxsb r,t,a`

vector signed short	vector signed short	`vspltish t,0`
		`vsubshs t,t,a`
		`vmaxsh r,t,a`

vector signed int	vector signed int	`vspltisw t,0`
		`vsubsws t,t,a`
		`vmaxsw r,t,a`

vec_add

Vector Addition

```
r = vec_add (a, b)
```

Purpose: Computes the sum of two vectors.

Result value: The value of each element of **r** is the sum of the corresponding elements of **a** and **b**. Modular arithmetic is used for both signed and unsigned integers.

Endian considerations: None.

Table 4.4. Supported type signatures for vec_add

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vaddubm r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vaddubm r,a,b`
vector signed short	vector signed short	vector signed short	`vadduhm r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vadduhm r,a,b`
vector signed int	vector signed int	vector signed int	`vadduwm r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vadduwm r,a,b`
vector signed long long	vector signed long long	vector signed long long	`vaddudm r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vaddudm r,a,b`
vector signed __int128	vector signed __int128	vector signed __int128	`vadduqm r,a,b`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vadduqm r,a,b`
vector float	vector float	vector float	`xvaddsp r,a,b`
vector double	vector double	vector double	`xvadddp r,a,b`

vec_addc

Vector Add Carrying

```
r = vec_addc (a, b)
```

Purpose: Returns a vector of carries produced by adding two vectors.

Result value: The value of each element of **r** is the carry produced by adding the corresponding elements of **a** and **b** (1 if there is a carry, 0 otherwise).

Endian considerations: None.

Table 4.5. Supported type signatures for vec_addc

r	a	b	Example Implementation
vector signed int	vector signed int	vector signed int	`vaddcuw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vaddcuw r,a,b`
vector signed __int128	vector signed __int128	vector signed __int128	`vaddcuq r,a,b`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vaddcuq r,a,b`

vec_adde

Vector Add Extended

```
r = vec_adde (a, b, c)
```

Purpose: Returns a vector formed as the sum of two vectors and a carry vector.

Result value: The value of each element of **r** is produced by adding the corresponding elements of **a** and **b** with a carry specified in the corresponding element of **c** (1 if there is a carry, 0 otherwise).

Endian considerations: None.

Notes: Code generated for this intrinsic should ensure only the low-order bit of **c** participates in the sum.

Table 4.6. Supported type signatures for vec_adde

r	a	b	c	Example Implementation

vector signed int	vector signed int	vector signed int	vector signed int	`vspltisw t,1`
				`vadduwm r,a,b`
				`xxland c,c,t`
				`vadduwm r,r,c`

vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	`vspltisw t,1`
				`vadduwm r,a,b`
				`xxland c,c,t`
				`vadduwm r,r,c`

vector signed __int128	vector signed __int128	vector signed __int128	vector signed __int128	`vaddeuqm r,a,b,c`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vaddeuqm r,a,b,c`

vec_addec

Vector Add Extended Carrying

```
r = vec_addec (a, b, c)
```

Purpose: Returns a vector of carries produced by adding two vectors and a carry vector.

Result value: The value of each element of **r** is the carry produced by adding the corresponding elements of **a** and **b** and a carry specified in the corresponding element of **c** (1 if there is a carry, 0 otherwise).

Endian considerations: None.

Notes: Code generated for this intrinsic should ensure only the low-order bit of **c** participates in the sum.

Table 4.7. Supported type signatures for vec_addec

r	a	b	c	Example Implementation

vector signed int	vector signed int	vector signed int	vector signed int	`vspltisw t,1`
				`xxland u,c,t`
				`vadduwm v,a,b`
				`vaddcuw w,a,b`
				`vaddcuw x,v,u`
				`xxlor r,w,x`

vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	`vspltisw t,1`
				`xxland u,c,t`
				`vadduwm v,a,b`
				`vaddcuw w,a,b`
				`vaddcuw x,v,u`
				`xxlor r,w,x`

vector signed __int128	vector signed __int128	vector signed __int128	vector signed __int128	`vaddecuq r,a,b,c`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vaddecuq r,a,b,c`

vec_adds

Vector Add Saturating

```
r = vec_adds (a, b)
```

Purpose: Computes the saturated sum of two vectors.

Result value: The value of each element of **r** is the saturated sum of the corresponding elements of **a** and **b**.

Endian considerations: None.

Table 4.8. Supported type signatures for vec_adds

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vaddsbs r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vaddubs r,a,b`
vector signed short	vector signed short	vector signed short	`vaddshs r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vadduhs r,a,b`
vector signed int	vector signed int	vector signed int	`vaddsws r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vadduws r,a,b`

vec_all_eq

Vector All Equal

```
r = vec_all_eq (a, b)
```

Purpose: Tests whether all elements of **a** are equal to the corresponding elements of **b**.

Result value: **r** is 1 if each element of **a** is equal to the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.9. Supported type signatures for vec_all_eq

r	a	b	Example Implementation

int	vector bool char	vector bool char	`vcmpequb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed char	vector signed char	`vcmpequb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned char	vector unsigned char	`vcmpequb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector bool short	vector bool short	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed short	vector signed short	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned short	vector unsigned short	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector pixel	vector pixel	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector bool int	vector bool int	`vcmpequw. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed int	vector signed int	`vcmpequw. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned int	vector unsigned int	`vcmpequw. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

r	a	b	Example Implementation

int	vector bool long long	vector bool long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed long long	vector signed long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned long long	vector unsigned long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector float	vector float	`xvcmpeqsp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector double	vector double	`xvcmpeqdp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

vec_all_ge

Vector All Greater or Equal

```
r = vec_all_ge (a, b)
```

Purpose: Tests whether all elements of **a** are greater than or equal to the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are greater than or equal to the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.10. Supported type signatures for vec_all_ge

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector signed short	vector signed short	`vcmpgtsh. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector signed int	vector signed int	`vcmpgtsw. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector float	vector float	`xvcmpgesp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

r	a	b	Example Implementation

int	vector double	vector double	`xvcmpgedp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

vec_all_gt

Vector All Greater Than

```
r = vec_all_gt (a, b)
```

Purpose: Tests whether all elements of **a** are greater than the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are greater than the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.11. Supported type signatures for vec_all_gt

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed short	vector signed short	`vcmpgtsh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed int	vector signed int	`vcmpgtsw. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector float	vector float	`xvcmpgtsp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector double	vector double	`xvcmpgtdp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

vec_all_in

Vector All In Range

```
r = vec_all_in (a, b)
```

Purpose: Tests whether all elements of a vector are within a given range.

Result value: **r** is 1 if each element of **a** has a value less than or equal to the value of the corresponding element of **b**, and greater than or equal to the negative of the value of the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.12. Supported type signatures for vec_all_in

r	a	b	Example Implementation

int	vector float	vector float	`vcmpbfp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

vec_all_le

Vector All Less or Equal

```
r = vec_all_le (a, b)
```

Purpose: Tests whether all elements of **a** are less than or equal to the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are less than or equal to the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.13. Supported type signatures for vec_all_le

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector signed short	vector signed short	`vcmpgtsh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector signed int	vector signed int	`vcmpgtsw. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector float	vector float	`xvcmpgesp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector double	vector double	`xvcmpgedp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

vec_all_lt

Vector All Less Than

```
r = vec_all_lt (a, b)
```

Purpose: Tests whether all elements of **a** are less than the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are less than the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.14. Supported type signatures for vec_all_lt

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed short	vector signed short	`vcmpgtsh. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed int	vector signed int	`vcmpgtsw. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector float	vector float	`xvcmpgtsp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector double	vector double	`xvcmpgtdp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

vec_all_nan

Vector All Not-a-Number

```
r = vec_all_nan (a)
```

Purpose: Tests whether all elements of **a** are not-a-number (NaN).

Result value: **r** is 1 if all elements of **a** are NaN. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.15. Supported type signatures for vec_all_nan

r	a	Example Implementation

int	vector float	`xvcmpeqsp. t,a,a`
		`mfocrf u,2`
		`rlwinm r,u,27,1`

int	vector double	`xvcmpeqdp. t,a,a`
		`mfocrf u,2`
		`rlwinm r,u,27,1`

vec_all_ne

Vector All Not Equal

```
r = vec_all_ne (a, b)
```

Purpose: Tests whether all elements of **a** are not equal to the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are not equal to the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.16. Supported type signatures for vec_all_ne

r	a	b	Example Implementation

int	vector bool char	vector bool char	`vcmpneb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed char	vector signed char	`vcmpneb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned char	vector unsigned char	`vcmpneb. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector bool short	vector bool short	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed short	vector signed short	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned short	vector unsigned short	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector pixel	vector pixel	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector bool int	vector bool int	`vcmpnew. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector signed int	vector signed int	`vcmpnew. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

int	vector unsigned int	vector unsigned int	`vcmpnew. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,25,1`

r	a	b	Example Implementation

int	vector bool long long	vector bool long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector signed long long	vector signed long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector unsigned long long	vector unsigned long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector float	vector float	`xvcmpeqsp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector double	vector double	`xvcmpeqdp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

vec_all_nge

Vector All Not Greater or Equal

```
r = vec_all_nge (a, b)
```

Purpose: Tests whether all elements of **a** are not greater than or equal to the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are not greater than or equal to the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.17. Supported type signatures for vec_all_nge

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgesp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector double	vector double	`xvcmpgedp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

vec_all_ngt

Vector All Not Greater Than

```
r = vec_all_ngt (a, b)
```

Purpose: Tests whether all elements of **a** are not greater than the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are not greater than the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.18. Supported type signatures for vec_all_ngt

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgtsp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector double	vector double	`xvcmpgtdp. t,a,b`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

vec_all_nle

Vector All Not Less or Equal

```
r = vec_all_nle (a, b)
```

Purpose: Tests whether all elements of **a** are not less than or equal to the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are not less than or equal to the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.19. Supported type signatures for vec_all_nle

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgesp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector double	vector double	`xvcmpgedp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

vec_all_nlt

Vector All Not Less Than

```
r = vec_all_nlt (a, b)
```

Purpose: Tests whether all elements of **a** are not less than the corresponding elements of **b**.

Result value: **r** is 1 if all elements of **a** are not less than the corresponding elements of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.20. Supported type signatures for vec_all_nlt

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgtsp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

int	vector double	vector double	`xvcmpgtdp. t,b,a`
			`mfocrf u,2`
			`rlwinm r,u,27,1`

vec_all_numeric

Vector All Numeric

```
r = vec_all_numeric (a)
```

Purpose: Tests whether all elements of the vector are numeric (not NaN).

Result value: **r** is 1 if all elements of **a** are numeric (not NaN). Otherwise, **r** is 0.

Endian considerations: None.

Table 4.21. Supported type signatures for vec_all_numeric

r	a	Example Implementation

int	vector float	`xvcmpeqsp. t,a,b`
		`mfocrf u,2`
		`rlwinm r,u,25,1`

int	vector double	`xvcmpeqdp. t,a,b`
		`mfocrf u,2`
		`rlwinm r,u,25,1`

vec_and

Vector AND

```
r = vec_and (a, b)
```

Purpose: Performs a bitwise AND of two vectors.

Result value: The value of **r** is the bitwise AND of **a** and **b**.

Endian considerations: None.

Table 4.22. Supported type signatures for vec_and

r	a	b	Example Implementation	Restrictions
vector bool char	vector bool char	vector bool char	`xxland r,a,b`	
vector signed char	vector signed char	vector signed char	`xxland r,a,b`	
vector unsigned char	vector unsigned char	vector unsigned char	`xxland r,a,b`	
vector bool short	vector bool short	vector bool short	`xxland r,a,b`	
vector signed short	vector signed short	vector signed short	`xxland r,a,b`	
vector unsigned short	vector unsigned short	vector unsigned short	`xxland r,a,b`	
vector signed int	vector signed int	vector signed int	`xxland r,a,b`	
vector bool int	vector bool int	vector bool int	`xxland r,a,b`	
vector unsigned int	vector unsigned int	vector unsigned int	`xxland r,a,b`	
vector bool long long	vector bool long long	vector bool long long	`xxland r,a,b`	
vector signed long long	vector signed long long	vector signed long long	`xxland r,a,b`	
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxland r,a,b`	
vector float	vector float	vector float	`xxland r,a,b`	
vector double	vector double	vector double	`xxland r,a,b`	

vec_andc

Vector AND with Complement

```
r = vec_andc (a, b)
```

Purpose: Performs a bitwise AND of one vector with the bitwise complement of another vector.

Result value: The value of **r** is the bitwise AND of **a** with the bitwise complement of **b**.

Endian considerations: None.

Table 4.23. Supported type signatures for vec_andc

r	a	b	Example Implementation	Restrictions
vector bool char	vector bool char	vector bool char	`xxlandc r,a,b`	
vector signed char	vector signed char	vector signed char	`xxlandc r,a,b`	
vector unsigned char	vector unsigned char	vector unsigned char	`xxlandc r,a,b`	
vector bool short	vector bool short	vector bool short	`xxlandc r,a,b`	
vector signed short	vector signed short	vector signed short	`xxlandc r,a,b`	
vector unsigned short	vector unsigned short	vector unsigned short	`xxlandc r,a,b`	
vector signed int	vector signed int	vector signed int	`xxlandc r,a,b`	
vector bool int	vector bool int	vector bool int	`xxlandc r,a,b`	
vector unsigned int	vector unsigned int	vector unsigned int	`xxlandc r,a,b`	
vector bool long long	vector bool long long	vector bool long long	`xxlandc r,a,b`	
vector signed long long	vector signed long long	vector signed long long	`xxlandc r,a,b`	
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxlandc r,a,b`	
vector float	vector float	vector float	`xxlandc r,a,b`	
vector double	vector double	vector double	`xxlandc r,a,b`	

vec_any_eq

Vector Any Equal

```
r = vec_any_eq (a, b)
```

Purpose: Tests whether any element of **a** is equal to the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is equal to the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.24. Supported type signatures for vec_any_eq

r	a	b	Example Implementation

int	vector bool char	vector bool char	`vcmpneb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed char	vector signed char	`vcmpneb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned char	vector unsigned char	`vcmpneb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector bool short	vector bool short	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed short	vector signed short	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned short	vector unsigned short	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector pixel	vector pixel	`vcmpneh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

r	a	b	Example Implementation

int	vector bool int	vector bool int	`vcmpnew. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed int	vector signed int	`vcmpnew. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned int	vector unsigned int	`vcmpnew. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector bool long long	vector bool long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed long long	vector signed long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned long long	vector unsigned long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector float	vector float	`xvcmpeqsp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpeqdp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_ge

Vector Any Greater or Equal

```
r = vec_any_ge (a, b)
```

Purpose: Tests whether any element of **a** is greater than or equal to the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is greater than or equal to the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.25. Supported type signatures for vec_any_ge

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed int	vector signed int	`vcmpgtsw. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed short	vector signed short	`vcmpgtsh. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

r	a	b	Example Implementation

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgedp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector float	vector float	`vcmpgesp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_gt

Vector Any Greater Than

```
r = vec_any_gt (a, b)
```

Purpose: Tests whether any element of **a** is greater than the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is greater than the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.26. Supported type signatures for vec_any_gt

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed short	vector signed short	`vcmpgtsh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed int	vector signed int	`vcmpgtsw. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

r	a	b	Example Implementation

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector float	vector float	`xvcmpgtsp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgtdp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_le

Vector Any Less or Equal

```
r = vec_any_le (a, b)
```

Purpose: Tests whether any element of **a** is less than or equal to the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is less than or equal to the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.27. Supported type signatures for vec_any_le

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed short	vector signed short	`vcmpgtsh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed int	vector signed int	`vcmpgtsw. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

r	a	b	Example Implementation

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector float	vector float	`xvcmpgesp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgedp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_lt

Vector Any Less Than

```
r = vec_any_lt (a, b)
```

Purpose: Tests whether any element of **a** is less than the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is less than the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.28. Supported type signatures for vec_any_lt

r	a	b	Example Implementation

int	vector signed char	vector signed char	`vcmpgtsb. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned char	vector unsigned char	`vcmpgtub. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed short	vector signed short	`vcmpgtsh. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned short	vector unsigned short	`vcmpgtuh. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed int	vector signed int	`vcmpgtsw. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned int	vector unsigned int	`vcmpgtuw. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed long long	vector signed long long	`vcmpgtsd. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

r	a	b	Example Implementation

int	vector unsigned long long	vector unsigned long long	`vcmpgtud. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector float	vector float	`xvcmpgtsp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgtdp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_nan

Vector Any Not-a-Number

```
r = vec_any_nan (a)
```

Purpose: Tests whether any element of the source vector is a NaN.

Result value: **r** is 1 if any element of **a** is a NaN. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.29. Supported type signatures for vec_any_nan

r	a	Example Implementation

int	vector float	`xvcmpeqsp. t,a,a`
		`mfocrf u,2`
		`rlwinm v,u,25,1`
		`cntlzw w,v`
		`srwi r,w,5`

int	vector double	`xvcmpeqdp. t,a,a`
		`mfocrf u,2`
		`rlwinm v,u,25,1`
		`cntlzw w,v`
		`srwi r,w,5`

vec_any_ne

Vector Any Not Equal

```
r = vec_any_ne (a, b)
```

Purpose: Tests whether any element of **a** is not equal to the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is not equal to the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.30. Supported type signatures for vec_any_ne

r	a	b	Example Implementation

int	vector bool char	vector bool char	`vcmpequb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed char	vector signed char	`vcmpequb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned char	vector unsigned char	`vcmpequb. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector bool short	vector bool short	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed short	vector signed short	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned short	vector unsigned short	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector pixel	vector pixel	`vcmpequh. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

r	a	b	Example Implementation

int	vector bool int	vector bool int	`vcmpequw. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed int	vector signed int	`vcmpequw. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned int	vector unsigned int	`vcmpequw. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector bool long long	vector bool long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector signed long long	vector signed long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector unsigned long long	vector unsigned long long	`vcmpequd. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector float	vector float	`xvcmpeqsp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpeqdp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_nge

Vector Any Not Greater or Equal

```
r = vec_any_nge (a, b)
```

Purpose: Tests whether any element of **a** is not greater than or equal to the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is not greater than or equal to the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.31. Supported type signatures for vec_any_nge

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgesp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgedp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_ngt

Vector Any Not Greater Than

```
r = vec_any_ngt (a, b)
```

Purpose: Tests whether any element of **a** is not greater than the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is not greater than the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.32. Supported type signatures for vec_any_ngt

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgtsp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgtdp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_nle

Vector Any Not Less or Equal

```
r = vec_any_nle (a, b)
```

Purpose: Tests whether any element of **a** is not less than or equal to the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is not less than or equal to the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.33. Supported type signatures for vec_any_nle

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgesp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgedp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_nlt

Vector Any Not Less Than

```
r = vec_any_nlt (a, b)
```

Purpose: Tests whether any element of **a** is not less than the corresponding element of **b**.

Result value: **r** is 1 if any element of **a** is not less than the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.34. Supported type signatures for vec_any_nlt

r	a	b	Example Implementation

int	vector float	vector float	`xvcmpgtdp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

int	vector double	vector double	`xvcmpgtdp. t,b,a`
			`mfocrf u,2`
			`rlwinm v,u,25,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_any_numeric

Vector Any Numeric

```
r = vec_any_numeric (a)
```

Purpose: Tests whether any element of the source vector is numeric (not a NaN).

Result value: **r** is 1 if any element of **a** is numeric (not a NaN). Otherwise, **r** is 0.

Endian considerations: None.

Table 4.35. Supported type signatures for vec_any_numeric

r	a	Example Implementation

int	vector float	`xvcmpeqsp. t,a,a`
		`mfocrf u,2`
		`rlwinm v,u,27,1`
		`cntlzw w,v`
		`srwi r,w,5`

int	vector double	`xvcmpeqdp. t,a,a`
		`mfocrf u,2`
		`rlwinm v,u,27,1`
		`cntlzw w,v`
		`srwi r,w,5`

vec_any_out

Vector Any Out of Range

```
r = vec_any_out (a, b)
```

Purpose: Tests whether the value of any element of a vector is outside of a given range.

Result value: **r** is 1 if the value of any element of **a** is greater than the value of the corresponding element of **b** or less than the negative of the value of the corresponding element of **b**. Otherwise, **r** is 0.

Endian considerations: None.

Table 4.36. Supported type signatures for vec_any_out

r	a	b	Example Implementation

int	vector float	vector float	`vcmpbfp. t,a,b`
			`mfocrf u,2`
			`rlwinm v,u,27,1`
			`cntlzw w,v`
			`srwi r,w,5`

vec_avg

Vector Average

```
r = vec_avg (a, b)
```

Purpose: Returns a vector containing the elementwise average of two vectors.

Result value: The value of each element of **r** is the average of the value of the corresponding elements of **a** and **b**.

Endian considerations: None.

Table 4.37. Supported type signatures for vec_avg

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vavgsb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vavgub r,a,b`
vector signed short	vector signed short	vector signed short	`vavgsh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vavguh r,a,b`
vector signed int	vector signed int	vector signed int	`vavgsw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vavguw r,a,b`

vec_bperm

Vector Bit Permute

```
r = vec_bperm (a, b)
```

Purpose: Gathers up to 16 one-bit values from a quadword or from each doubleword element in the specified order, zeroing other bits.

Result value: When the type of **a** is vector unsigned char or vector unsigned __int128:

- For each *i* (0 ≤ *i* < 16), let bit index *j* denote the byte value of the i^{th} element of **b**.
- If bit index *j* is greater than or equal to 128, bit *i* of **r** is set to 0.
- If bit index *j* is smaller than 128, bit 48+*i* of **r** is set to the value of the j^{th} bit of **a**.
- All other bits of **r** are zeroed.

When the type of **a** is vector unsigned long long:

- For each doubleword element *i* (0 ≤ *i* < 2) of **a**:
  - For each *j* (0 ≤ *j* < 8), let bit index *k* denote the byte value of the $(8i + j)^{th}$ element of **b**.
  - If bit index *k* is greater than or equal to 64, bit 56+*j* of element *i* of **r** is set to 0.
  - If bit index *k* is less than 64, bit 56+*j* of element *i* of **r** is set to the value of the k^{th} bit of element *i* of input **a**.
  - All other bits are zeroed.

An example for input **a** of type vector unsigned char follows:

byte index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	A9
b	7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70

Zoom in to view just the two bytes in **a** ([14..15]) containing the bits referenced by the bit indices **b**[i] (0 ≤ *i* < 16), ([7F..70]):

byte index	**14**								**15**							
bit index	***70***	***71***	***72***	***73***	***74***	***75***	***76***	***77***	***78***	***79***	***7A***	***7B***	***7C***	***7D***	***7E***	***7F***
a	FF								A9							
bit $\mathbf{a}_{\text{bit index}}$	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	1
bit $\mathbf{a}_{\text{b[i]}}$	1	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1
intermediate result	95								FF							

byte index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
r	00	00	00	00	00	00	95	FF	00	00	00	00	00	00	00	00

Endian considerations: All bit and byte numberings within each element in the above description denote big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_bperm` does not follow the bi-endian programming model.

Table 4.38. Supported type signatures for vec_bperm

r	a	b	Example Implementation	Restrictions
vector unsigned char	vector unsigned char	vector unsigned char	`vbpermq r,a,b`	
vector unsigned long long	vector unsigned __int128	vector unsigned char	`vbpermq r,a,b`	
vector unsigned long long	vector unsigned long long	vector unsigned char	`vbpermd r,a,b`	ISA 3.0 or later

vec_ceil

Vector Ceiling

```
r = vec_ceil (a)
```

Purpose: Returns a vector **r** that contains the result of applying the floating-point ceiling function to each element of **a**.

Result value: The value of each element of **r** is the smallest representable floating-point integral value greater than or equal to the value of the corresponding element of **a**.

Endian considerations: None.

Table 4.39. Supported type signatures for vec_ceil

r	a	Example Implementation
vector float	vector float	`xvrspip r,a`
vector double	vector double	`xvrdpip r,a`

vec_cipher_be

Vector AES Cipher Big-Endian

```
r = vec_cipher_be (a, b)
```

Purpose: Performs one round of the AES cipher operation on an intermediate state array **a** by using a given round key **b**.

Result value: r contains the resulting intermediate state, after one round of the AES cipher operation on intermediate state array **a**, using the round key specified by **b**.

Endian considerations: All element and bit numberings of the AES cipher operation use big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_cipher_be` does not follow the bi-endian programming model.

Table 4.40. Supported type signatures for vec_cipher_be

r	a	b	Example Implementation
vector unsigned char	vector unsigned char	vector unsigned char	`vcipher r,a,b`

vec_cipherlast_be

Vector AES Cipher Last Big-Endian

```
r = vec_cipherlast_be (a, b)
```

Purpose: Performs the final round of the AES cipher operation on an intermediate state array **a** using the specified round key **b**.

Result value: r contains the resulting final state, after the final round of the AES cipher operation on intermediate state array **a**, using the round key specified by **b**.

Endian considerations: All element and bit numberings of the AES cipher-last operation use big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_cipherlast_be` does not follow the bi-endian programming model.

Table 4.41. Supported type signatures for vec_cipherlast_be

r	a	b	Example Implementation
vector unsigned char	vector unsigned char	vector unsigned char	`vcipherlast r,a,b`

vec_cmpb

Vector Compare Bytes

```
r = vec_cmpb (a, b)
```

Purpose: Performs a bounds comparison of each set of corresponding elements of two vectors.

Result value: Each element of **r** has the value 0 if the value of the corresponding element of **a** is less than or equal to the value of the corresponding element of **b** and greater than or equal to the negated value of the corresponding element of **b**. Otherwise:

- If an element of **b** is greater than or equal to 0, then the value of the corresponding element of **r** is 0 if the absolute value of the corresponding element of **a** is equal to the value of the corresponding element of **b**. The value is negative if it is greater than the value of the corresponding element of **b**. It is positive if it is less than the value of the corresponding element of **b**.
- If an element of **b** is less than 0, then the value of the element of **r** is positive if the value of the corresponding element of **a** is less than or equal to the value of the element of **b**. Otherwise, it is negative.

Endian considerations: None.

Table 4.42. Supported type signatures for vec_cmpb

r	a	b	Example Implementation
vector signed int	vector float	vector float	`vcmpbfp r,a,b`

vec_cmpeq

Vector Compare Equal

```
r = vec_cmpeq (a, b)
```

Purpose: Returns a vector containing the results of comparing each set of corresponding elements of two vectors for equality.

Result value: For each element of **r**, the value of each bit is 1 if the corresponding elements of **a** and **b** are equal. Otherwise, the value of each bit is 0.

Endian considerations: None.

Table 4.43. Supported type signatures for vec_cmpeq

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`vcmpequb r,a,b`
vector bool char	vector signed char	vector signed char	`vcmpequb r,a,b`
vector bool char	vector unsigned char	vector unsigned char	`vcmpequb r,a,b`
vector bool short	vector bool short	vector bool short	`vcmpequh r,a,b`
vector bool short	vector signed short	vector signed short	`vcmpequh r,a,b`
vector bool short	vector unsigned short	vector unsigned short	`vcmpequh r,a,b`
vector bool int	vector bool int	vector bool int	`vcmpequw r,a,b`
vector bool int	vector signed int	vector signed int	`vcmpequw r,a,b`
vector bool int	vector unsigned int	vector unsigned int	`vcmpequw r,a,b`
vector bool long long	vector bool long long	vector bool long long	`vcmpequd r,a,b`
vector bool long long	vector signed long long	vector signed long long	`vcmpequd r,a,b`
vector bool long long	vector unsigned long long	vector unsigned long long	`vcmpequd r,a,b`
vector bool int	vector float	vector float	`xvcmpeqsp r,a,b`
vector bool long long	vector double	vector double	`xvcmpeqdp r,a,b`

vec_cmpge

Vector Compare Greater or Equal

```
r = vec_cmpge (a, b)
```

Purpose: Returns a vector containing the results of a greater-than-or-equal-to comparison between each set of corresponding elements of two vectors.

Result value: For each element of **r**, the value of each bit is 1 if the corresponding element of **a** is greater than or equal to the corresponding element of **b**. Otherwise, the value of each bit is 0.

Endian considerations: None.

Table 4.44. Supported type signatures for vec_cmpge

r	a	b	Example Implementation
vector bool char	vector signed char	vector signed char	`vcmpgtsb t,b,a` `xxlnor r,t,t`
vector bool char	vector unsigned char	vector unsigned char	`vcmpgtub t,b,a` `xxlnor r,t,t`
vector bool short	vector signed short	vector signed short	`vcmpgtsh t,b,a` `xxlnor r,t,t`
vector bool short	vector unsigned short	vector unsigned short	`vcmpgtuh t,b,a` `xxlnor r,t,t`
vector bool int	vector signed int	vector signed int	`vcmpgtsw t,b,a` `xxlnor r,t,t`
vector bool int	vector unsigned int	vector unsigned int	`vcmpgtuw t,b,a` `xxlnor r,t,t`
vector bool long long	vector signed long long	vector signed long long	`vcmpgtsd t,b,a` `xxlnor r,t,t`
vector bool long long	vector unsigned long long	vector unsigned long long	`vcmpgtud t,b,a` `xxlnor r,t,t`
vector bool int	vector float	vector float	`xvcmpgesp r,a,b`
vector bool long long	vector double	vector double	`xvcmpgedp r,a,b`

vec_cmpgt

Vector Compare Greater Than

```
r = vec_cmpgt (a, b)
```

Purpose: Returns a vector containing the results of a greater-than comparison between each set of corresponding elements of two vectors.

Result value: For each element of **r**, the value of each bit is 1 if the corresponding element of **a** is greater than the corresponding element of **b**. Otherwise, the value of each bit is 0.

Endian considerations: None.

Table 4.45. Supported type signatures for vec_cmpgt

r	a	b	Example Implementation
vector bool char	vector signed char	vector signed char	`vcmpgtsb r,a,b`
vector bool char	vector unsigned char	vector unsigned char	`vcmpgtub r,a,b`
vector bool short	vector signed short	vector signed short	`vcmpgtsh r,a,b`
vector bool short	vector unsigned short	vector unsigned short	`vcmpgtuh r,a,b`
vector bool int	vector signed int	vector signed int	`vcmpgtsw r,a,b`
vector bool int	vector unsigned int	vector unsigned int	`vcmpgtuw r,a,b`
vector bool long long	vector signed long long	vector signed long long	`vcmpgtsd r,a,b`
vector bool long long	vector unsigned long long	vector unsigned long long	`vcmpgtud r,a,b`
vector bool int	vector float	vector float	`xvcmpgtsp r,a,b`
vector bool long long	vector double	vector double	`xvcmpgtdp r,a,b`

vec_cmple

Vector Compare Less Than or Equal

```
r = vec_cmple (a, b)
```

Purpose: Returns a vector containing the results of a less-than-or-equal comparison between each set of corresponding elements of two vectors.

Result value: For each element of **r**, the value of each bit is 1 if the corresponding element of **a** is less than or equal to the corresponding element of **b**. Otherwise, the value of each bit is 0.

Endian considerations: None.

Table 4.46. Supported type signatures for vec_cmple

r	a	b	Example Implementation
vector bool char	vector signed char	vector signed char	`vcmpgtsb t,a,b` `xxlnor r,t,t`
vector bool char	vector unsigned char	vector unsigned char	`vcmpgtub t,a,b` `xxlnor r,t,t`
vector bool short	vector signed short	vector signed short	`vcmpgtsh t,a,b` `xxlnor r,t,t`
vector bool short	vector unsigned short	vector unsigned short	`vcmpgtuh t,a,b` `xxlnor r,t,t`
vector bool int	vector signed int	vector signed int	`vcmpgtsw t,a,b` `xxlnor r,t,t`
vector bool int	vector unsigned int	vector unsigned int	`vcmpgtuw t,a,b` `xxlnor r,t,t`
vector bool long long	vector signed long long	vector signed long long	`vcmpgtsd t,a,b` `xxlnor r,t,t`
vector bool long long	vector unsigned long long	vector unsigned long long	`vcmpgtud t,a,b` `xxlnor r,t,t`
vector bool int	vector float	vector float	`xvcmpgesp r,b,a`
vector bool long long	vector double	vector double	`xvcmpgedp r,b,a`

vec_cmplt

Vector Compare Less Than

```
r = vec_cmplt (a, b)
```

Purpose: Returns a vector containing the results of a less-than comparison between each set of corresponding elements of two vectors.

Result value: For each element of **r**, the value of each bit is 1 if the corresponding element of **a** is less than the corresponding element of **b**. Otherwise, the value of each bit is 0.

Endian considerations: None.

Table 4.47. Supported type signatures for vec_cmplt

r	a	b	Example Implementation
vector bool char	vector signed char	vector signed char	`vcmpgtsb r,b,a`
vector bool char	vector unsigned char	vector unsigned char	`vcmpgtub r,b,a`
vector bool short	vector signed short	vector signed short	`vcmpgtsh r,b,a`
vector bool short	vector unsigned short	vector unsigned short	`vcmpgtuh r,b,a`
vector bool int	vector signed int	vector signed int	`vcmpgtsw r,b,a`
vector bool int	vector unsigned int	vector unsigned int	`vcmpgtuw r,b,a`
vector bool long long	vector signed long long	vector signed long long	`vcmpgtsd r,b,a`
vector bool long long	vector unsigned long long	vector unsigned long long	`vcmpgtud r,b,a`
vector bool int	vector float	vector float	`xvcmpgtsp r,b,a`
vector bool long long	vector double	vector double	`xvcmpgtdp r,b,a`

vec_cmpne

Vector Compare Not Equal

```
r = vec_cmpne (a, b)
```

Purpose: Returns a vector containing the results of comparing each set of corresponding elements of two vectors for inequality.

Result value: For each element of **r**, the value of each bit is 1 if the corresponding elements of **a** and **b** are not equal. Otherwise, the value of each bit is 0.

Endian considerations: None.

Table 4.48. Supported type signatures for vec_cmpne

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`vcmpneb r,a,b`
vector bool char	vector signed char	vector signed char	`vcmpneb r,a,b`
vector bool char	vector unsigned char	vector unsigned char	`vcmpneb r,a,b`
vector bool short	vector bool short	vector bool short	`vcmpneh r,a,b`
vector bool short	vector signed short	vector signed short	`vcmpneh r,a,b`
vector bool short	vector unsigned short	vector unsigned short	`vcmpneh r,a,b`
vector bool int	vector bool int	vector bool int	`vcmpnew r,a,b`
vector bool int	vector signed int	vector signed int	`vcmpnew r,a,b`
vector bool int	vector unsigned int	vector unsigned int	`vcmpnew r,a,b`
vector bool long long	vector bool long long	vector bool long long	`vcmpequd t,a,b` `xxlnor r,t,t`
vector bool long long	vector signed long long	vector signed long long	`vcmpequd t,a,b` `xxlnor r,t,t`
vector bool long long	vector unsigned long long	vector unsigned long long	`vcmpequd t,a,b` `xxlnor r,t,t`
vector bool int	vector float	vector float	`xvcmpeqsp t,a,b` `xxlnor r,t,t`

r	a	b	Example Implementation
vector bool long long	vector double	vector double	`xvcmpeqdp t,a,b` `xxlnor r,t,t`

vec_cmpnez

Vector Compare Not Equal or Zero

```
r = vec_cmpnez (a, b)
```

Purpose: Returns a vector containing the results of comparing each set of corresponding elements of two vectors for inequality, or for an element with a zero value.

Result value: For each element of **r**, the value of each bit is 1 if the corresponding elements of **a** and **b** are not equal, or if the **a** element or the **b** element is zero. Otherwise, the value of each bit is 0.

Endian considerations: None.

Table 4.49. Supported type signatures for vec_cmpnez

r	a	b	Example Implementation	Restrictions
vector bool char	vector signed char	vector signed char	`vcmpnezb r,a,b`	ISA 3.0 or later
vector bool char	vector unsigned char	vector unsigned char	`vcmpnezb r,a,b`	ISA 3.0 or later
vector bool short	vector signed short	vector signed short	`vcmpnezh r,a,b`	ISA 3.0 or later
vector bool short	vector unsigned short	vector unsigned short	`vcmpnezh r,a,b`	ISA 3.0 or later
vector bool int	vector signed int	vector signed int	`vcmpnezw r,a,b`	ISA 3.0 or later
vector bool int	vector unsigned int	vector unsigned int	`vcmpnezw r,a,b`	ISA 3.0 or later

vec_cntlz

Vector Count Leading Zeros

```
r = vec_cntlz (a)
```

Purpose: Returns a vector containing the number of most-significant bits equal to zero of each corresponding element of the source vector.

Result value: The value of each element of **r** is set to the number of leading zeros of the corresponding element of **a**.

An example for input **a** of type vector unsigned char follows:

byte index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
a	00	10	20	30	48	58	64	74	82	92	A1	B1	C0	D0	E0	F0
r	08	03	02	02	01	01	01	01	00	00	00	00	00	00	00	00

Endian considerations: None.

Table 4.50. Supported type signatures for vec_cntlz

r	a	Example Implementation	Restrictions
vector signed char	vector signed char	`vclzb r,a`	
vector unsigned char	vector unsigned char	`vclzb r,a`	
vector signed short	vector signed short	`vclzh r,a`	
vector unsigned short	vector unsigned short	`vclzh r,a`	
vector signed int	vector signed int	`vclzw r,a`	
vector unsigned int	vector unsigned int	`vclzw r,a`	
vector signed long long	vector signed long long	`vclzd r,a`	
vector unsigned long long	vector unsigned int long long	`vclzd r,a`	

vec_cntlz_lsbb

Vector Count Leading Zero Least-Significant Bits by Byte

```
r = vec_cntlz_lsbb (a)
```

Purpose: Returns the number of leading byte elements (starting at the lowest-numbered element) of a vector that have a least-significant bit of zero.

Result value: The value of **r** is set to the number of leading byte elements (starting at the lowest-numbered element) of **a** that have a least-significant bit of zero.

An example for input **a** of type vector unsigned char follows:

***byte index* n**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	00	10	20	30	48	58	64	74	82	92	A1	B1	C0	D0	E0	F0
least-significant bit of $\mathbf{a}_n$	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
r	0x0A (10)															

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.51. Supported type signatures for vec_cntlz_lsbb

r	a	Example BE Implementation	Example LE Implementation	Restrictions
signed int	vector signed char	`vclzlsbb r,a`	`vctzlsbb r,a`	ISA 3.0 or later
signed int	vector unsigned char	`vclzlsbb r,a`	`vctzlsbb r,a`	ISA 3.0 or later

vec_cnttz

Vector Count Trailing Zeros

```
r = vec_cnttz (a)
```

Purpose: Returns a vector containing the number of least-significant bits equal to zero of each corresponding element of the source vector.

Result value: The value of each element of **r** is set to the number of trailing zeros of the corresponding element of **a**.

An example for input **a** of type vector unsigned char follows:

byte index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
a	00	10	20	30	48	58	64	74	82	92	A1	B1	C0	D0	E0	F0
r	08	04	05	04	03	03	02	02	01	01	00	00	06	04	05	04

Endian considerations: None.

Table 4.52. Supported type signatures for vec_cnttz

r	a	Example Implementation	Restrictions
vector signed char	vector signed char	`vctzb r,a`	ISA 3.0 or later
vector unsigned char	vector unsigned char	`vctzb r,a`	ISA 3.0 or later
vector signed short	vector signed short	`vctzh r,a`	ISA 3.0 or later
vector unsigned short	vector unsigned short	`vctzh r,a`	ISA 3.0 or later
vector signed int	vector signed int	`vctzw r,a`	ISA 3.0 or later
vector unsigned int	vector unsigned int	`vctzw r,a`	ISA 3.0 or later
vector signed long long	vector signed long long	`vctzd r,a`	ISA 3.0 or later
vector unsigned long long	vector unsigned int long long	`vctzd r,a`	ISA 3.0 or later

vec_cnttz_lsbb

Vector Count Trailing Zero Least-Significant Bits by Byte

```
r = vec_cnttz_lsbb (a)
```

Purpose: Returns the number of trailing byte elements (starting at the highest-numbered element) of a vector that have a least-significant bit of zero.

Result value: The value of **r** is set to the number of trailing byte elements (starting at the highest-numbered element) of **a** that have a least-significant bit of zero.

An example for input **a** of type vector unsigned char follows:

***byte index* n**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	00	10	20	30	48	58	64	74	82	92	A1	B1	C0	D0	E0	F0
least-significant bit of $\mathbf{a}_n$	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
r	0x04 (4)															

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.53. Supported type signatures for vec_cnttz_lsbb

r	a	Example BE Implementation	Example LE Implementation	Restrictions
signed int	vector signed char	`vctzlsbb r,a`	`vclzlsbb r,a`	ISA 3.0 or later
signed int	vector unsigned char	`vctzlsbb r,a`	`vclzlsbb r,a`	ISA 3.0 or later

vec_cpsgn

Vector Copy Sign

```
r = vec_cpsgn (a, b)
```

Purpose: Returns a vector by copying the sign of the elements in one vector to the sign of the corresponding elements of another vector.

Result value: The value of each element of **r** is set to the corresponding element of **b** with its sign replaced by the sign from the corresponding element of **a**.

Endian considerations: None.

Table 4.54. Supported type signatures for vec_cpsgn

r	a	b	Example Implementation	Restrictions
vector float	vector float	vector float	`xvcpsgnsp r,b,a`	
vector double	vector double	vector double	`xvcpsgndp r,b,a`	

vec_ctf

Vector Convert to Floating-Point

```
r = vec_ctf (a, b)
```

Purpose: Converts an integer vector into a floating-point vector.

Result value: The value of each element of **r** is the closest floating-point approximation of the value of the corresponding element of **a** divided by 2 to the power of **b**, which must be in the range 0–31.

Endian considerations: None.

Notes: The example implementations below assume **b** is zero, so that the scaling code is omitted. Scaling is accomplished by multiplying each element of **r** by 2 to the power of –**b**.

Table 4.55. Supported type signatures for vec_ctf

r	a	b	Example Implementation
vector float	vector signed int	5-bit unsigned literal	`vcfsx r,a,b`
vector float	vector unsigned int	5-bit unsigned literal	`vcfux r,a,b`
vector double	vector signed long long	5-bit unsigned literal	`xvcvsxddp r,a,b`
vector double	vector unsigned long long	5-bit unsigned literal	`xvcvuxddp r,a,b`

vec_cts

Vector Convert to Signed Integer

```
r = vec_cts (a, b)
```

Purpose: Converts a floating-point vector into a signed integer vector.

Result value: The value of each element of **r** is the saturated signed-integer value, truncated towards zero, obtained by multiplying the corresponding element of **a** multiplied by 2 to the power of **b**, which must be in the range 0–31.

Endian considerations: None.

Table 4.56. Supported type signatures for vec_cts

r	a	b	Example Implementation
vector signed int	vector float	5-bit unsigned literal	`vctsxs r,a,b`

vec_ctu

Vector Convert to Unsigned Integer

```
r = vec_ctu (a, b)
```

Purpose: Converts a floating-point vector into an unsigned integer vector.

Result value: The value of each element of **r** is the saturated unsigned-integer value, truncated towards zero, obtained by multiplying the corresponding element of **a** multiplied by 2 to the power of **b**, which must be in the range 0–31.

Endian considerations: None.

Table 4.57. Supported type signatures for vec_ctu

r	a	b	Example Implementation
vector unsigned int	vector float	5-bit unsigned literal	`vctuxs r,a,b`

vec_div

Vector Divide

```
r = vec_div (a, b)
```

Purpose: Divides the elements in one vector by the corresponding elements in another vector and places the quotients in the result vector.

Result value: The value of each element of **r** is obtained by dividing the corresponding element of **a** by the corresponding element of **b**.

Endian considerations: None.

Table 4.58. Supported type signatures for vec_div

r	a	b	Example Implementation

vector signed long long	vector signed long long	vector signed long long	`xxspltd t,a,1`
			`mfvsrd u,t`
			`xxspltd v,b,1`
			`mfvsrd w,v`
			`divd x,u,w`
			`mfvsrd u,a`
			`mtvsrd y,x`
			`mfvsrd w,b`
			`divd x,u,w`
			`mtvsrd z,x`
			`xxmrghd r,z,y`

vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxspltd t,a,1`
			`mfvsrd u,t`
			`xxspltd v,b,1`
			`mfvsrd w,v`
			`divd x,u,w`
			`mfvsrd u,a`
			`mtvsrd y,x`
			`mfvsrd w,b`
			`divd x,u,w`
			`mtvsrd z,x`
			`xxmrghd r,z,y`

vector float	vector float	vector float	`xvdivsp r,a,b`
vector double	vector double	vector double	`xvdivdp r,a,b`

vec_double

Vector Convert to Double Precision

```
r = vec_double (a)
```

Purpose: Converts a vector of long integers into a vector of double-precision numbers.

Result value: The value of each element of **r** is obtained by converting the corresponding element of **a** to double precision floating-point.

Endian considerations: None.

Table 4.59. Supported type signatures for vec_double

r	a	Example Implementation
vector double	vector signed long long	`xvcvsxddp r,a`
vector double	vector unsigned long long	`xvcvuxddp r,a`

vec_doublee

Vector Convert Even Elements to Double Precision

```
r = vec_doublee (a)
```

Purpose: Converts the even elements of a vector into a vector of double-precision numbers.

Result value: Elements 0 and 1 of **r** are set to the converted values of elements 0 and 2 of **a**.

An example for input **a** of type vector signed int follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
a	00000001	*(ignored)*	FFFFFFFF	*(ignored)*
r	1.0		-1.0	

Endian considerations: Differences in element numbering require different implementations for big- and little-endian code generation.

Table 4.60. Supported type signatures for vec_doublee

r	a	Example LE Implementation	Example BE Implementation	Restrictions
vector double	vector signed int	`xxsldwi t,a,a,1` `xvcvsxwdp r,t`	`xvcvsxwdp r,a`	
vector double	vector unsigned int	`xxsldwi t,a,a,1` `xvcvuxwdp r,t`	`xvcvuxwdp r,a`	
vector double	vector float	`xxsldwi t,a,a,1` `xvcvspdp r,t`	`xvcvspdp r,a`	

vec_doubleh

Vector Convert High Elements to Double Precision

```
r = vec_doubleh (a)
```

Purpose: Converts the high-order elements of a vector into a vector of double-precision numbers.

Result value: Elements 0 and 1 of **r** are set to the converted values of elements 0 and 1 of **a**.

An example for input **a** of type vector signed int follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
a	00000001	FFFFFFFF	*(ignored)*	*(ignored)*
r	1.0		-1.0	

Endian considerations: Differences in element numbering require different implementations for big- and little-endian code generation.

Table 4.61. Supported type signatures for vec_doubleh

r	a	Example LE Implementation	Example BE Implementation	Restrictions

vector double	vector signed int	`xxsldwi t,a,a,3`
		`xxsldwi u,a,t,2`
		`xvcvsxwdp r,u`
			`xxsldwi t,a,a,1`
			`xxsldwi u,t,a,3`
			`xvcvsxwdp r,u`	

vector double	vector unsigned int	`xxsldwi t,a,a,3`
		`xxsldwi u,a,t,2`
		`xvcvuxwdp r,u`
			`xxsldwi t,a,a,1`
			`xxsldwi u,t,a,3`
			`xvcvuxwdp r,u`	

vector double	vector float	`xxsldwi t,a,a,3`
		`xxsldwi u,a,t,2`
		`xvcvspdp r,u`
			`xxsldwi t,a,a,1`
			`xxsldwi u,t,a,3`
			`xvcvspdp r,u`	

vec_doublel

Vector Convert Low Elements to Double Precision

```
r = vec_doublel (a)
```

Purpose: Converts the low-order elements of a vector into a vector of double-precision numbers.

Result value: Elements 0 and 1 of **r** are set to the converted values of elements 2 and 3 of **a**.

An example for input **a** of type vector signed int follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
a	*(ignored)*	*(ignored)*	00000001	FFFFFFFF
r	1.0		-1.0	

Endian considerations: Differences in element numbering require different implementations for big- and little-endian code generation.

Table 4.62. Supported type signatures for vec_doublel

r	a	Example LE Implementation	Example BE Implementation	Restrictions

vector double	vector signed int	`xxsldwi t,a,a,1`
		`xxsldwi u,t,a,3`
		`xvcvsxwdp r,u`
			`xxsldwi t,a,a,3`
			`xxsldwi u,a,t,2`
			`xvcvsxwdp r,u`	

vector double	vector unsigned int	`xxsldwi t,a,a,1`
		`xxsldwi u,t,a,3`
		`xvcvuxwdp r,u`
			`xxsldwi t,a,a,3`
			`xxsldwi u,a,t,2`
			`xvcvuxwdp r,u`	

vector double	vector float	`xxsldwi t,a,a,1`
		`xxsldwi u,t,a,3`
		`xvcvspdp r,u`
			`xxsldwi t,a,a,3`
			`xxsldwi u,a,t,2`
			`xvcvspdp r,u`	

vec_doubleo

Vector Convert Odd Elements to Double Precision

```
r = vec_doubleo (a)
```

Purpose: Converts the odd elements of a vector into a vector of double-precision numbers.

Result value: Elements 0 and 1 of **r** are set to the converted values of elements 1 and 3 of **a**.

An example for input **a** of type vector signed int follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
a	*(ignored)*	00000001	*(ignored)*	FFFFFFFF
r	1.0		-1.0	

Endian considerations: Differences in element numbering require different implementations for big- and little-endian code generation.

Table 4.63. Supported type signatures for vec_doubleo

r	a	Example LE Implementation	Example BE Implementation	Restrictions
vector double	vector signed int	`xvcvsxwdp r,a`	`xxsldwi t,a,a,1` `xvcvsxwdp r,t`	
vector double	vector unsigned int	`xvcvuxwdp r,a`	`xxsldwi t,a,a,1` `xvcvuxwdp r,t`	
vector double	vector float	`xvcvspdp r,a`	`xxsldwi t,a,a,1` `xvcvspdp r,t`	

vec_eqv

Vector Equivalence

```
r = vec_eqv (a, b)
```

Purpose: Performs a bitwise equivalence (exclusive NOR) of two vectors.

Result value: The value of **r** is the bitwise XNOR of **a** and **b**.

Endian considerations: None.

Table 4.64. Supported type signatures for vec_eqv

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`xxleqv r,a,b`
vector signed char	vector signed char	vector signed char	`xxleqv r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`xxleqv r,a,b`
vector bool short	vector bool short	vector bool short	`xxleqv r,a,b`
vector signed short	vector signed short	vector signed short	`xxleqv r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`xxleqv r,a,b`
vector signed int	vector signed int	vector signed int	`xxleqv r,a,b`
vector bool int	vector bool int	vector bool int	`xxleqv r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`xxleqv r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxleqv r,a,b`
vector signed long long	vector signed long long	vector signed long long	`xxleqv r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxleqv r,a,b`
vector float	vector float	vector float	`xxleqv r,a,b`
vector double	vector double	vector double	`xxleqv r,a,b`

vec_expte

Vector Exponential Estimate

```
r = vec_expte (a)
```

Purpose: Returns a vector **r** containing estimates of 2 raised to the power of the corresponding elements of **a**.

Result value: The value of each element of **r** is the estimated value of 2 raised to the power of the corresponding element of **a**.

Endian considerations: None.

Table 4.65. Supported type signatures for vec_expte

r	a	Example Implementation
vector float	vector float	`vexptefp r,a`

vec_extract

Vector Extract

```
r = vec_extract (a, b)
```

Purpose: Returns the value of the **b**th element of vector **a**.

Result value: The value of each element of **r** is the element of **a** at position **b** modulo the number of elements of **a**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Notes: Prior to ISA 3.0, less efficient code sequences must be used to implement vec_extract.

Table 4.66. Supported type signatures for vec_extract

r	a	b	Example ISA 3.0 LE Implementation	Example ISA 3.0 BE Implementation
signed char	vector signed char	signed int	`vextubrx t,b,a` `extsb r,t`	`vextublx t,b,a` `extsb r,t`
unsigned char	vector bool char	signed int	`vextubrx t,b,a`	`vextublx t,b,a`
unsigned char	vector unsigned char	signed int	`vextubrx t,b,a`	`vextublx t,b,a`

signed short	vector signed short	signed int	`slwi t,b,1`
			`vextuhrx u,t,a`
			`extsh r,u`
				`slwi t,b,1`
				`vextuhlx u,t,a`
				`extsh r,u`

unsigned short	vector bool short	signed int	`slwi t,b,1` `vextuhrx r,t,a`	`slwi t,b,1` `vextuhlx r,t,a`
unsigned short	vector unsigned short	signed int	`slwi t,b,1` `vextuhrx r,t,a`	`slwi t,b,1` `vextuhlx r,t,a`

signed int	vector signed int	signed int	`slwi t,b,2`
			`vextuwrx u,t,a`
			`extsw r,u`
				`slwi t,b,2`
				`vextuwlx u,t,a`
				`extsw r,u`

unsigned int	vector bool int	signed int	`slwi t,b,2` `vextuwrx r,t,a`	`slwi t,b,2` `vextuwlx r,t,a`
unsigned int	vector unsigned int	signed int	`slwi t,b,2` `vextuwrx r,t,a`	`slwi t,b,2` `vextuwlx r,t,a`

signed long long	vector signed long long	signed int	`xori t,b,0x1`
			`rldic u,t,6,57`
			`mtvsrdd v,u,u`
			`vslo w,a,v`
			`mfvsrd r,w`
				`rldic t,b,6,57`
				`mtvsrdd u,t,t`
				`vslo v,a,u`
				`mfvsrd r,v`

r	a	b	Example ISA 3.0 LE Implementation	Example ISA 3.0 BE Implementation

unsigned long long	vector bool long long	signed int	`xori t,b,0x1`
			`rldic u,t,6,57`
			`mtvsrdd v,u,u`
			`vslo w,a,v`
			`mfvsrd r,w`
				`rldic t,b,6,57`
				`mtvsrdd u,t,t`
				`vslo v,a,u`
				`mfvsrd r,v`

unsigned long long	vector unsigned long long	signed int	`xori t,b,0x1`
			`rldic u,t,6,57`
			`mtvsrdd v,u,u`
			`vslo w,a,v`
			`mfvsrd r,w`
				`rldic t,b,6,57`
				`mtvsrdd u,t,t`
				`vslo v,a,u`
				`mfvsrd r,v`

float	vector float	signed int	`rldicl t,b,0,62`
			`subfic u,t,3`
			`sldi v,u,5`
			`mtvsrdd w,v,v`
			`vslo x,a,w`
			`xscvspdp r,x`
				`sldi t,b,5`
				`mtvsrdd u,t,t`
				`vslo v,a,u`
				`xscvspdp r,v`

double	vector double	signed int	`xori t,b,0x1`
			`rldic u,t,6,57`
			`mtvsrdd v,u,u`
			`vslo r,a,v`
				`rldic t,b,6,57`
				`mtvsrdd u,t,t`
				`vslo r,a,u`

vec_extract_exp

Vector Extract Exponent

```
r = vec_extract_exp (a)
```

Purpose: Extracts exponents from a vector of floating-point numbers.

Result value: Each element of **r** is extracted from the exponent field of the corresponding floating-point vector element of **a**.

The extracted exponents of **a** are returned as right-justified unsigned integers containing biased exponents, in accordance with the exponent representation specified by IEEE 754, without further processing.

Endian considerations: None.

Table 4.67. Supported type signatures for vec_extract_exp

r	a	Example Implementation	Restrictions
vector unsigned int	vector float	`xvxexpsp r,a`	ISA 3.0 or later
vector unsigned long long	vector double	`xvxexpdp r,a`	ISA 3.0 or later

vec_extract_fp32_from_shorth

Vector Extract Floats from High Elements of Vector Short Int

```
r = vec_extract_fp32_from_shorth (a)
```

Purpose: Extracts four single-precision floating-point numbers from the high elements of a vector of eight 16-bit elements, interpreting each element as a 16-bit floating-point number in IEEE format.

Result value: The first four elements of **a** are interpreted as 16-bit floating-point numbers in IEEE format, and extended to single-precision format, returning a vector with four single-precision IEEE numbers.

An example follows:

halfword index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***
word index	***0***		***1***		***2***		***3***	
a	3800 (0.5)	4200 (3.0)	4700 (7.0)	4B80 (15.0)	4FC0 (31.0)	53E0 (63.0)	57F0 (127.0)	5BF8 (255.0)
r	0.5		3.0		7.0		15.0	

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Notes: The example implementation assumes that the vperm instruction is used for big-endian, and the vpermr instruction is used for little-endian. The permute control vector for the vperm or vpermr instruction is in a memory location identified by pcv. The value located at pcv is identical in natural element order for big- and little-endian: { 15, 14, 0, 0, 13, 12, 0, 0, 11, 10, 0, 0, 9, 8, 0, 0 }.

Table 4.68. Supported type signatures for vec_extract_fp32_from_shorth

r	a	Example Implementation	Restrictions

vector float	vector unsigned short	`lxv t,0(pcv)`
		`vperm[r] u,a,a,t`
		`xvcvhpsp r,u`
			ISA 3.0 or later

vec_extract_fp32_from_shortl

Vector Extract Floats from Low Elements of Vector Short Int

```
r = vec_extract_fp32_from_shortl (a)
```

Purpose: Extracts four single-precision floating-point numbers from the low elements of a vector of eight 16-bit elements, interpreting each element as a 16-bit floating-point number in IEEE format.

Result value: The last four elements of **a** are interpreted as 16-bit floating-point numbers in IEEE format, and extended to single-precision format, returning a vector with four single-precision IEEE numbers.

An example follows:

halfword index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***
word index	***0***		***1***		***2***		***3***	
a	3800 (0.5)	4200 (3.0)	4700 (7.0)	4B80 (15.0)	4FC0 (31.0)	53E0 (63.0)	57F0 (127.0)	5BF8 (255.0)
r	31.0		63.0		127.0		255.0	

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Notes: The example implementation assumes that the vperm instruction is used for big-endian, and the vpermr instruction is used for little-endian. The permute control vector for the vperm or vpermr instruction is in a memory location identified by pcv. The value located at pcv is identical in natural element order for big- and little-endian: { 7, 6, 0, 0, 5, 4, 0, 0, 3, 2, 0, 0, 1, 0, 0, 0 }.

Table 4.69. Supported type signatures for vec_extract_fp32_from_shortl

r	a	Example Implementation	Restrictions

vector float	vector unsigned short	`lxv t,0(pcv)`
		`vperm[r] u,a,a,t`
		`xvcvhpsp r,u`
			ISA 3.0 or later

vec_extract_sig

Vector Extract Significand

```
r = vec_extract_sig (a)
```

Purpose: Extracts a vector of significands (mantissas) from a vector of floating-point numbers.

Result value: Each element of **r** is extracted from the significand (mantissa) field of the corresponding floating-point element of **a**.

The significand is from the corresponding floating-point number in accordance with the IEEE format. The returned result includes the implicit leading digit. The value of that digit is not encoded in the IEEE format, but is implied by the exponent.

Endian considerations: None.

Table 4.70. Supported type signatures for vec_extract_sig

r	a	Example Implementation	Restrictions
vector unsigned int	vector float	`xvxsigsp r,a`	ISA 3.0 or later
vector unsigned long long	vector double	`xvxsigdp r,a`	ISA 3.0 or later

vec_extract4b

Vector Extract Four Bytes

```
r = vec_extract4b (a, b)
```

Purpose: Extracts a word from vector **a** at constant byte position **b**.

Result value: The first doubleword element of **r** contains the zero-extended extracted word from **a**. The second doubleword is set to 0. **b** specifies the least-significant byte number (0–12) of the word to be extracted.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.71. Supported type signatures for vec_extract4b

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions
vector unsigned long long	vector unsigned char	const int (range [0,12])	`xxextractuw r,a,12-b`	`xxextractuw r,a,b`	ISA 3.0 or later

vec_first_match_index

Vector Index of First Match

```
r = vec_first_match_index (a, b)
```

Purpose: Performs a comparison of equality on each of the corresponding elements of **a** and **b**, and returns the first position of equality.

Result value: Returns the element index of the position of the first character match in natural element order. If no match, returns the number of characters as an element count in the vector argument.

An example for input **a** of type vector unsigned char follows:

***byte index* n**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	00	10	20	30	40	??	??	??	??	??	??	??	??	??	??	??
b	FF	FF	FF	FF	40	??	??	??	??	??	??	??	??	??	??	??
r	4															

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.72. Supported type signatures for vec_first_match_index

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions

unsigned int	vector signed char	vector signed char	`vcmpneb t,a,b`
			`xxlnor u,t,t`
			`vctzlsbb r,u`
				`vcmpneb t,a,b`
				`xxlnor u,t,t`
				`vclzlsbb r,u`
					ISA 3.0 or later

unsigned int	vector unsigned char	vector unsigned char	`vcmpneb t,a,b`
			`xxlnor u,t,t`
			`vctzlsbb r,u`
				`vcmpneb t,a,b`
				`xxlnor u,t,t`
				`vclzlsbb r,u`
					ISA 3.0 or later

unsigned int	vector signed short	vector signed short	`vcmpneh t,a,b`
			`xxlnor u,t,t`
			`vctzlsbb v,u`
			`rldicl r,v,63,33`
				`vcmpneh t,a,b`
				`xxlnor u,t,t`
				`vclzlsbb v,u`
				`rldicl r,v,63,33`
					ISA 3.0 or later

unsigned int	vector unsigned short	vector unsigned short	`vcmpneh t,a,b`
			`xxlnor u,t,t`
			`vctzlsbb v,u`
			`rldicl r,v,63,33`
				`vcmpneh t,a,b`
				`xxlnor u,t,t`
				`vclzlsbb v,u`
				`rldicl r,v,63,33`
					ISA 3.0 or later

unsigned int	vector signed int	vector signed int	`vcmpnew t,a,b`
			`xxlnor u,t,t`
			`vctzlsbb v,u`
			`rldicl r,v,62,34`
				`vcmpnew t,a,b`
				`xxlnor u,t,t`
				`vclzlsbb v,u`
				`rldicl r,v,62,34`
					ISA 3.0 or later

unsigned int	vector unsigned int	vector unsigned int	`vcmpnew t,a,b`
			`xxlnor u,t,t`
			`vctzlsbb v,u`
			`rldicl r,v,62,34`
				`vcmpnew t,a,b`
				`xxlnor u,t,t`
				`vclzlsbb v,u`
				`rldicl r,v,62,34`
					ISA 3.0 or later

vec_first_match_or_eos_index

Vector Index of First Match or End of String

```
r = vec_first_match_or_eos_index (a, b)
```

Purpose: Performs a comparison of equality on each of the corresponding elements of **a** and **b**. Returns the first position of equality, or the zero string terminator.

Result value: Returns the element index of the position, in natural element order, of either the first character match or an end-of-string (EOS) terminator. If no match or terminator, returns the number of characters as an element count in the vector argument.

An example for input **a** of type vector unsigned char follows:

***byte index* n**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	01	02	03	00	??	??	??	??	??	??	??	??	??	??	??	??
b	FF	FF	FF	FF	??	??	??	??	??	??	??	??	??	??	??	??
r	3															

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.73. Supported type signatures for vec_first_match_or_eos_index

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions

unsigned int	vector signed char	vector signed char	`xxspltib t,0`
			`vcmpneb u,a,t`
			`vcmpneb v,b,t`
			`vcmpnezb w,a,b`
			`xxland x,u,v`
			`xxlnand y,x,w`
			`vctzlsbb r,y`
				`xxspltib t,0`
				`vcmpneb u,a,t`
				`vcmpneb v,b,t`
				`vcmpnezb w,a,b`
				`xxland x,u,v`
				`xxlnand y,x,w`
				`vclzlsbb r,y`
					ISA 3.0 or later

unsigned int	vector unsigned char	vector unsigned char	`xxspltib t,0`
			`vcmpneb u,a,t`
			`vcmpneb v,b,t`
			`vcmpnezb w,a,b`
			`xxland x,u,v`
			`xxlnand y,x,w`
			`vctzlsbb r,y`
				`xxspltib t,0`
				`vcmpneb u,a,t`
				`vcmpneb v,b,t`
				`vcmpnezb w,a,b`
				`xxland x,u,v`
				`xxlnand y,x,w`
				`vclzlsbb r,y`
					ISA 3.0 or later

unsigned int	vector signed short	vector signed short	`xxspltib t,0`
			`vcmpneh u,a,t`
			`vcmpneh v,b,t`
			`vcmpnezh w,a,b`
			`xxland x,u,v`
			`xxlnand y,x,w`
			`vctzlsbb z,y`
			`rldicl r,z,63,33`
				`xxspltib t,0`
				`vcmpneh u,a,t`
				`vcmpneh v,b,t`
				`vcmpnezh w,a,b`
				`xxland x,u,v`
				`xxlnand y,x,w`
				`vclzlsbb z,y`
				`rldicl r,z,63,33`
					ISA 3.0 or later

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions

unsigned int	vector unsigned short	vector unsigned short	`xxspltib t,0`
			`vcmpneh u,a,t`
			`vcmpneh v,b,t`
			`vcmpnezh w,a,b`
			`xxland x,u,v`
			`xxlnand y,x,w`
			`vctzlsbb z,y`
			`rldicl r,z,63,33`
				`xxspltib t,0`
				`vcmpneh u,a,t`
				`vcmpneh v,b,t`
				`vcmpnezh w,a,b`
				`xxland x,u,v`
				`xxlnand y,x,w`
				`vclzlsbb z,y`
				`rldicl r,z,63,33`
					ISA 3.0 or later

unsigned int	vector signed int	vector signed int	`xxspltib t,0`
			`vcmpnew u,a,t`
			`vcmpnew v,b,t`
			`vcmpnezw w,a,b`
			`xxland x,u,v`
			`xxlnand y,x,w`
			`vctzlsbb z,y`
			`rldicl r,z,62,34`
				`xxspltib t,0`
				`vcmpnew u,a,t`
				`vcmpnew v,b,t`
				`vcmpnezw w,a,b`
				`xxland x,u,v`
				`xxlnand y,x,w`
				`vclzlsbb z,y`
				`rldicl r,z,62,34`
					ISA 3.0 or later

unsigned int	vector unsigned int	vector unsigned int	`xxspltib t,0`
			`vcmpnew u,a,t`
			`vcmpnew v,b,t`
			`vcmpnezw w,a,b`
			`xxland x,u,v`
			`xxlnand y,x,w`
			`vctzlsbb z,y`
			`rldicl r,z,62,34`
				`xxspltib t,0`
				`vcmpnew u,a,t`
				`vcmpnew v,b,t`
				`vcmpnezw w,a,b`
				`xxland x,u,v`
				`xxlnand y,x,w`
				`vclzlsbb z,y`
				`rldicl r,z,62,34`
					ISA 3.0 or later

vec_first_mismatch_index

Vector Index of First Mismatch

```
r = vec_first_mismatch_index (a, b)
```

Purpose: Performs a comparison of inequality on each of the corresponding elements of **a** and **b**, and returns the first position of inequality.

Result value: Returns the element index of the position of the first character mismatch in natural element order. If no mismatch, returns the number of characters as an element count in the vector argument.

An example for input **a** of type vector unsigned char follows:

***byte index* n**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	00	01	??	??	??	??	??	??	??	??	??	??	??	??	??	??
b	00	02	??	??	??	??	??	??	??	??	??	??	??	??	??	??
r	1															

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.74. Supported type signatures for vec_first_mismatch_index

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions
unsigned int	vector signed char	vector signed char	`vcmpneb t,a,b` `vctzlsbb r,t`	`vcmpneb t,a,b` `vclzlsbb r,t`	ISA 3.0 or later
unsigned int	vector unsigned char	vector unsigned char	`vcmpneb t,a,b` `vctzlsbb r,t`	`vcmpneb t,a,b` `vclzlsbb r,t`	ISA 3.0 or later

unsigned int	vector signed short	vector signed short	`vcmpneh t,a,b`
			`vctzlsbb u,t`
			`rldicl r,u,63,33`
				`vcmpneh t,a,b`
				`vclzlsbb u,t`
				`rldicl r,u,63,33`
					ISA 3.0 or later

unsigned int	vector unsigned short	vector unsigned short	`vcmpneh t,a,b`
			`vctzlsbb u,t`
			`rldicl r,u,63,33`
				`vcmpneh t,a,b`
				`vclzlsbb u,t`
				`rldicl r,u,63,33`
					ISA 3.0 or later

unsigned int	vector signed int	vector signed int	`vcmpnew t,a,b`
			`vctzlsbb u,t`
			`rldicl r,u,62,34`
				`vcmpnew t,a,b`
				`vclzlsbb u,t`
				`rldicl r,u,62,34`
					ISA 3.0 or later

unsigned int	vector unsigned int	vector unsigned int	`vcmpnew t,a,b`
			`vctzlsbb u,t`
			`rldicl r,u,62,34`
				`vcmpnew t,a,b`
				`vclzlsbb u,t`
				`rldicl r,u,62,34`
					ISA 3.0 or later

vec_first_mismatch_or_eos_index

Vector Index of First Mismatch or End of String

```
r = vec_first_mismatch_or_eos_index (a, b)
```

Purpose: Performs a comparison of inequality on each of the corresponding elements of **a** and **b**. Returns the first position of inequality, or the zero string terminator.

Result value: Returns the element index of the position, in natural element order, of either the first character mismatch or an end-of-string (EOS) terminator. If no mismatch or terminator, returns the number of characters as an element count in the vector argument.

An example for input **a** of type vector unsigned char follows:

***byte index* n**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	01	02	03	00	??	??	??	??	??	??	??	??	??	??	??	??
b	01	02	03	00	??	??	??	??	??	??	??	??	??	??	??	??
r	3															

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.75. Supported type signatures for vec_first_mismatch_or_eos_index

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions

unsigned int	vector signed char	vector signed char	`xxspltib t,0`
			`vcmpneb u,a,t`
			`vcmpneb v,b,t`
			`vcmpnezb w,a,b`
			`xxland x,u,v`
			`xxlorc y,w,x`
			`vctzlsbb r,y`
				`xxspltib t,0`
				`vcmpneb u,a,t`
				`vcmpneb v,b,t`
				`vcmpnezb w,a,b`
				`xxland x,u,v`
				`xxlorc y,w,x`
				`vclzlsbb r,y`
					ISA 3.0 or later

unsigned int	vector unsigned char	vector unsigned char	`xxspltib t,0`
			`vcmpneb u,a,t`
			`vcmpneb v,b,t`
			`vcmpnezb w,a,b`
			`xxland x,u,v`
			`xxlorc y,w,x`
			`vctzlsbb r,y`
				`xxspltib t,0`
				`vcmpneb u,a,t`
				`vcmpneb v,b,t`
				`vcmpnezb w,a,b`
				`xxland x,u,v`
				`xxlorc y,w,x`
				`vclzlsbb r,y`
					ISA 3.0 or later

unsigned int	vector signed short	vector signed short	`xxspltib t,0`
			`vcmpneh u,a,t`
			`vcmpneh v,b,t`
			`vcmpnezh w,a,b`
			`xxland x,u,v`
			`xxlorc y,w,x`
			`vctzlsbb z,y`
			`rldicl r,z,63,33`
				`xxspltib t,0`
				`vcmpneh u,a,t`
				`vcmpneh v,b,t`
				`vcmpnezh w,a,b`
				`xxland x,u,v`
				`xxlorc y,w,x`
				`vclzlsbb z,y`
				`rldicl r,z,63,33`
					ISA 3.0 or later

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions

unsigned int	vector unsigned short	vector unsigned short	`xxspltib t,0`
			`vcmpneh u,a,t`
			`vcmpneh v,b,t`
			`vcmpnezh w,a,b`
			`xxland x,u,v`
			`xxlorc y,w,x`
			`vctzlsbb z,y`
			`rldicl r,z,63,33`
				`xxspltib t,0`
				`vcmpneh u,a,t`
				`vcmpneh v,b,t`
				`vcmpnezh w,a,b`
				`xxland x,u,v`
				`xxlorc y,w,x`
				`vclzlsbb z,y`
				`rldicl r,z,63,33`
					ISA 3.0 or later

unsigned int	vector signed int	vector signed int	`xxspltib t,0`
			`vcmpnew u,a,t`
			`vcmpnew v,b,t`
			`vcmpnezw w,a,b`
			`xxland x,u,v`
			`xxlorc y,w,x`
			`vctzlsbb z,y`
			`rldicr r,z,62,34`
				`xxspltib t,0`
				`vcmpnew u,a,t`
				`vcmpnew v,b,t`
				`vcmpnezw w,a,b`
				`xxland x,u,v`
				`xxlorc y,w,x`
				`vclzlsbb z,y`
				`rldicr r,z,62,34`
					ISA 3.0 or later

unsigned int	vector unsigned int	vector unsigned int	`xxspltib t,0`
			`vcmpnew u,a,t`
			`vcmpnew v,b,t`
			`vcmpnezw w,a,b`
			`xxland x,u,v`
			`xxlorc y,w,x`
			`vctzlsbb z,y`
			`rldicr r,z,62,34`
				`xxspltib t,0`
				`vcmpnew u,a,t`
				`vcmpnew v,b,t`
				`vcmpnezw w,a,b`
				`xxland x,u,v`
				`xxlorc y,w,x`
				`vclzlsbb z,y`
				`rldicr r,z,62,34`
					ISA 3.0 or later

vec_float

Vector Convert Integer to Floating-Point

```
r = vec_float (a)
```

Purpose: Converts a vector of integers to a vector of single-precision floating-point numbers.

Result value: Elements of **r** are obtained by converting the respective elements of **a** to single-precision floating-point numbers.

Endian considerations: None.

Table 4.76. Supported type signatures for vec_float

r	a	Example Implementation
vector float	vector signed int	`xvcvsxwsp r,a`
vector float	vector unsigned int	`xvcvuxwsp r,a`

vec_float2

Vector Convert Two Vectors to Floating-Point

```
r = vec_float2 (a, b)
```

Purpose: Converts two vectors of long long integers or double-precision floating-point numbers to a vector of single-precision numbers.

Result value: Elements of **r** are obtained by converting the elements of **a** and **b** to single-precision numbers. Elements 0 and 1 of **r** are converted from elements 0 and 1 of **a**, respectively, and elements 2 and 3 of **r** are converted from elements 0 and 1 of **b**, respectively.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.77. Supported type signatures for vec_float2

r	a	b	Example LE Implementation	Example BE Implementation

vector float	vector signed long long	vector signed long long	`xxpermdi t,b,a,0`
			`xxpermdi u,b,a,3`
			`xvcvsxdsp v,t`
			`xvcvsxdsp w,u`
			`vmrgow r,v,w`
				`xxpermdi t,b,a,0`
				`xxpermdi u,b,a,3`
				`xvcvsxdsp v,t`
				`xvcvsxdsp w,u`
				`vmrgew r,v,w`

vector float	vector unsigned long long	vector unsigned long long	`xxpermdi t,b,a,0`
			`xxpermdi u,b,a,3`
			`xvcvsxdsp v,t`
			`xvcvsxdsp w,u`
			`vmrgow r,v,w`
				`xxpermdi t,b,a,0`
				`xxpermdi u,b,a,3`
				`xvcvsxdsp v,t`
				`xvcvsxdsp w,u`
				`vmrgew r,v,w`

vector float	vector double	vector double	`xxpermdi t,b,a,0`
			`xxpermdi u,b,a,3`
			`xvcvsxdsp v,t`
			`xvcvsxdsp w,u`
			`vmrgow r,v,w`
				`xxpermdi t,b,a,0`
				`xxpermdi u,b,a,3`
				`xvcvsxdsp v,t`
				`xvcvsxdsp w,u`
				`vmrgew r,v,w`

vec_floate

Vector Convert to Floating-Point in Even Elements

```
r = vec_floate (a)
```

Purpose: Converts the elements of a source vector to single-precision floating-point and stores the results in the even elements of the target vector.

Result value: The even-numbered elements of **r** are obtained by converting the elements of **a** to single-precision numbers, using the current floating-point rounding mode.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.78. Supported type signatures for vec_floate

r	a	Example LE Implementation	Example BE Implementation
vector float	vector signed long long	`xvcvsxdsp r,a`	`xvcvsxdsp t,a` `vsldoi r,t,t,4`
vector float	vector unsigned long long	`xvcvuxdsp r,a`	`xvcvuxdsp t,a` `vsldoi r,t,t,4`
vector float	vector double	`xvcvdpsp r,a`	`xvcvdpsp t,a` `vsldoi r,t,t,4`

vec_floato

Vector Convert to Floating-Point in Odd Elements

```
r = vec_floato (a)
```

Purpose: Converts the elements of a source vector to single-precision floating-point and stores the results in the odd elements of the target vector.

Result value: The odd-numbered elements of **r** are obtained by converting the elements of **a** to single-precision numbers, using the current floating-point rounding mode.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.79. Supported type signatures for vec_floato

r	a	Example LE Implementation	Example BE Implementation
vector float	vector signed long long	`xvcvsxdsp t,a` `vsldoi r,t,t,4`	`xvcvsxdsp r,a`
vector float	vector unsigned long long	`xvcvuxdsp t,a` `vsldoi r,t,t,4`	`xvcvuxdsp r,a`
vector float	vector double	`xvcvdpsp t,a` `vsldoi r,t,t,4`	`xvcvdpsp r,a`

vec_floor

Vector Floor

```
r = vec_floor (a)
```

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 source vector.

Result value: Each element of **r** contains the largest representable floating-point integral value less than or equal to the value of the corresponding element of **a**.

Endian considerations: None.

Table 4.80. Supported type signatures for vec_floor

r	a	Example Implementation
vector float	vector float	`xvrspim r,a`
vector double	vector double	`xvrdpim r,a`

vec_gb

Vector Gather Bits by Byte

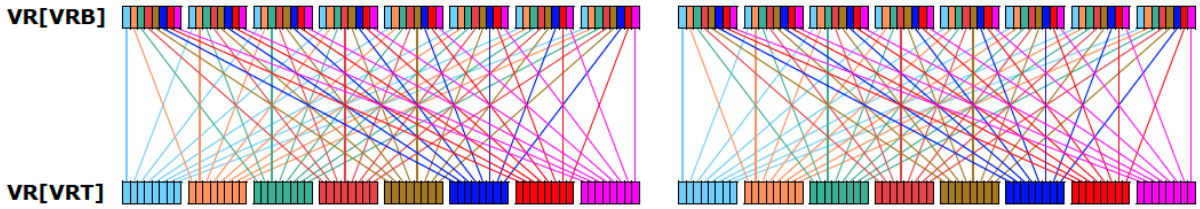
```
r = vec_gb (a)
```

Purpose: Performs a gather-bits operation on the input.

Result value: Within each doubleword, let x(i) (0 ≤ i < 8) denote the byte elements, with x(0) the most-significant byte. For each pair of i and j (0 ≤ i < 8, 0 ≤ j < 8), the *j*th bit of the *i*th byte element of **r** is set to the value of the *i*th bit of the *j*th byte element of **a**.

Figure 4.1, “Operation of vec_gb” [121], taken from the Power ISA, shows how bits are combined by the `vec_gb` intrinsic. Here `VR[VRT]` is equivalent to **r**, and `VR[VRB]` is equivalent to **a**.

Figure 4.1. Operation of vec_gb



Endian considerations: None.

Table 4.81. Supported type signatures for vec_gb

r	a	Example Implementation
vector unsigned char	vector unsigned char	`vgbbd r,a`

vec_insert

Vector Insert

```
r = vec_insert (a, b, c)
```

Purpose: Returns a copy of vector **b** with element **c** replaced by the value of **a**.

Result value: **r** contains a copy of vector **b** with element **c** replaced by the value of **a**. This function uses modular arithmetic on **c** to determine the element number. For example, if **c** is out of range, the compiler uses **c** modulo the number of elements in the vector to determine the element position.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Notes: The sample implementations are given for ISA 3.0 when **c** is a constant. For earlier target architectures, or when **c** is variable, less efficient sequences are required. The sample implementations also assume that **c** is in range; that is, any required modulus operations have already been performed on the constant index.

Table 4.82. Supported type signatures for vec_insert

r	a	b	c	Example ISA 3.0 LE Implementation	Example ISA 3.0 BE Implementation
vector signed char	signed char	vector signed char	signed int	`mtvsrwz t,b` `vinsertb r,t,15-c`	`mtvsrwz t,b` `vinsertb r,t,c`
vector unsigned char	unsigned char	vector unsigned char	signed int	`mtvsrwz t,b` `vinsertb r,t,15-c`	`mtvsrwz t,b` `vinsertb r,t,c`
vector signed short	signed short	vector signed short	signed int	`mtvsrwz t,b` `vinserth r,t,a,(7-c)*2`	`mtvsrd t,b` `vinserth r,t,a,c*2`
vector unsigned short	unsigned short	vector unsigned short	signed int	`mtvsrwz t,b` `vinserth r,t,a,(7-c)*2`	`mtvsrd t,b` `vinserth r,t,a,c*2`
vector signed int	signed int	vector signed int	signed int	`mtvsrwz t,b` `xxinsertw r,t,(3-c)*4`	`mtvsrwz t,b` `vinsertb r,t,c*4`
vector unsigned int	unsigned int	vector unsigned int	signed int	`mtvsrwz t,b` `xxinsertw r,t,(3-c)*4`	`mtvsrwz t,b` `vinsertb r,t,c*4`
vector signed long long	signed long long	vector signed long long	signed int	`mtvsrd t,b` `xxpermdi r,t,a,c`	`mtvsrd t,b` `xxpermdi r,t,a,1-c`
vector unsigned long long	unsigned long long	vector unsigned long long	signed int	`mtvsrd t,b` `xxpermdi r,t,a,c`	`mtvsrd t,b` `xxpermdi r,t,a,1-c`

vector float	float	vector float	signed int	`xscvdpspn t,a`
				`xxextractuw u,t,0`
				`xxinsertw r/b,u,(3-c)*4`
					`xscvdpspn t,a`
					`xxextractuw u,t,0`
					`xxinsertw r/b,u,c*4`

r	a	b	c	Example ISA 3.0 LE Implementation	Example ISA 3.0 BE Implementation

vector double	double	vector double	signed int	`xxpermdi r,b,a,1 [c=0]`
				`[or]`
				`xxpermdi r,a,b,1 [c=1]`
					`xxpermdi r,a,b,1 [c=0]`
					`[or]`
					`xxpermdi r,b,a,1 [c=1]`

vec_insert_exp

Vector Insert Exponent

```
r = vec_insert_exp (a, b)
```

Purpose: Inserts exponents into a vector of floating-point numbers.

Result value: Each element of **r** is generated by combining the exponent specified by the corresponding element of **b** with the sign and significand of the corresponding element of **a**.

The inserted exponent of **b** is treated as a right-justified unsigned integer containing a biased exponent, in accordance with the exponent representation specified by IEEE 754. It is combined with the sign and significand of **a** without further processing.

Endian considerations: None.

Table 4.83. Supported type signatures for vec_insert_exp

r	a	b	Example Implementation	Restrictions
vector float	vector unsigned int	vector unsigned int	`xviexpsp r,a,b`	ISA 3.0 or later
vector float	vector float	vector unsigned int	`xviexpsp r,a,b`	ISA 3.0 or later
vector double	vector unsigned long long	vector unsigned long long	`xviexpdp r,a,b`	ISA 3.0 or later
vector double	vector double	vector unsigned long long	`xviexpdp r,a,b`	ISA 3.0 or later

vec_insert4b

Vector Insert Four Bytes

```
r = vec_insert4b (a, b, c)
```

Purpose: Inserts a word into a vector at a byte position.

Result value: Let W be the first doubleword element of **a**, truncated to 32 bits. The result vector **r** is formed by inserting W into **b** at the byte position (0–12) specified by **c**.

Endian considerations: The element and byte numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.84. Supported type signatures for vec_insert4b

r	a	b	c	Example LE Implementation	Example BE Implementation	Restrictions
vector unsigned char	vector signed int	vector unsigned char	const int (range [0,12])	`xxpermdi t,a,a,1` `xxinsertw b,t,12-c`	`xxinsertw b,t,c`	ISA 3.0 or later
vector unsigned char	vector unsigned int	vector unsigned char	const int (range [0,12])	`xxpermdi t,a,a,1` `xxinsertw b,t,12-c`	`xxinsertw b,t,c`	ISA 3.0 or later

vec_ld

Vector Load Indexed

```
r = vec_ld (a, b)
```

Purpose: Loads a 16-byte vector from the memory address specified by the displacement and the pointer, ignoring the four low-order bits of the calculated address.

Result value: The value of **r** is obtained by adding **a** and **b**, masking off the four low-order bits of the result, and loading the 16-byte vector from the resultant memory address.

Endian considerations: None.

Table 4.85. Supported type signatures for vec_ld

r	a	b	Example ISA 3.0 Implementation
vector bool char	signed long long	const vector bool char *	`lvx r,b,a`
vector signed char	signed long long	const signed char *	`lvx r,b,a`
vector signed char	signed long long	const vector signed char *	`lvx r,b,a`
vector unsigned char	signed long long	const unsigned char *	`lvx r,b,a`
vector unsigned char	signed long long	const vector unsigned char *	`lvx r,b,a`
vector bool short	signed long long	const vector bool short *	`lvx r,b,a`
vector signed short	signed long long	const signed short *	`lvx r,b,a`
vector signed short	signed long long	const vector signed short *	`lvx r,b,a`
vector unsigned short	signed long long	const unsigned short *	`lvx r,b,a`
vector unsigned short	signed long long	const vector unsigned short *	`lvx r,b,a`
vector pixel	signed long long	const vector pixel *	`lvx r,b,a`
vector bool int	signed long long	const vector bool int *	`lvx r,b,a`
vector signed int	signed long long	const signed int *	`lvx r,b,a`
vector signed int	signed long long	const vector signed int *	`lvx r,b,a`

r	a	b	Example ISA 3.0 Implementation
vector unsigned int	signed long long	const unsigned int *	`lvx r,b,a`
vector unsigned int	signed long long	const vector unsigned int *	`lvx r,b,a`
vector bool long long	signed long long	const vector bool long long *	`lvx r,b,a`
vector signed long long	signed long long	const signed long long *	`lvx r,b,a`
vector signed long long	signed long long	const vector signed long long *	`lvx r,b,a`
vector unsigned long long	signed long long	const unsigned long long *	`lvx r,b,a`
vector unsigned long long	signed long long	const vector unsigned long long *	`lvx r,b,a`
vector signed __int128	signed long long	const signed __int128 *	`lvx r,b,a`
vector signed __int128	signed long long	const vector signed __int128 *	`lvx r,b,a`
vector unsigned __int128	signed long long	const unsigned __int128 *	`lvx r,b,a`
vector unsigned __int128	signed long long	const vector unsigned __int128 *	`lvx r,b,a`
vector float	signed long long	const float *	`lvx r,b,a`
vector float	signed long long	const vector float *	`lvx r,b,a`
vector double	signed long long	const double *	`lvx r,b,a`
vector double	signed long long	const vector double *	`lvx r,b,a`

vec_lde

Vector Load Element Indexed

```
r = vec_lde (a, b)
```

Purpose: Loads a single element into the position in the vector register corresponding to its address, leaving the remaining elements of the register undefined.

Result value: The integer value **a** is added to the pointer value **b**. The resulting address is rounded down to the nearest address that is a multiple of *es*, where *es* is 1 for char pointers, 2 for short pointers, and 4 for float or int pointers. The element at this address is loaded into an element of **r**, leaving all other elements of **r** undefined. The position of the loaded element in **r** is determined by taking the address modulo 16.

Endian considerations: None.

Notes: Be careful to note that the address (b+c) is aligned to an element boundary. Do not attempt to load unaligned data with this intrinsic.

Table 4.86. Supported type signatures for vec_lde

r	a	b	Example ISA 3.0 Implementation
vector signed char	signed long long	const signed char *	`lvebx r,b,a`
vector unsigned char	signed long long	const unsigned char *	`lvebx r,b,a`
vector signed short	signed long long	const signed short *	`lvehx r,b,a`
vector unsigned short	signed long long	const unsigned short *	`lvehx r,b,a`
vector signed int	signed long long	const signed int *	`lvewx r,b,a`
vector unsigned int	signed long long	const unsigned int *	`lvewx r,b,a`
vector float	signed long long	const float *	`lvewx r,b,a`

vec_ldl

Vector Load Indexed Least Recently Used

```
r = vec_ldl (a, b)
```

Purpose: Loads a 16-byte vector from the memory address specified by the displacement and the pointer, ignoring the four low-order bits of the calculated address, and marks the cache line loaded from as least recently used.

Result value: The value of **r** is obtained by adding **a** and **b**, masking off the four low-order bits of the result, and loading the 16-byte vector from the resultant memory address.

Endian considerations: None.

Notes: This intrinsic can be used to indicate the last access to a portion of memory, as a hint to the data cache controller that the associated cache line can be replaced without performance loss.

Table 4.87. Supported type signatures for vec_ldl

r	a	b	Example ISA 3.0 Implementation
vector bool char	signed long long	const vector bool char *	`lvxl r,b,a`
vector signed char	signed long long	const signed char *	`lvxl r,b,a`
vector signed char	signed long long	const vector signed char *	`lvxl r,b,a`
vector unsigned char	signed long long	const unsigned char *	`lvxl r,b,a`
vector unsigned char	signed long long	const vector unsigned char *	`lvxl r,b,a`
vector bool short	signed long long	const vector bool short *	`lvxl r,b,a`
vector signed short	signed long long	const signed short *	`lvxl r,b,a`
vector signed short	signed long long	const vector signed short *	`lvxl r,b,a`
vector unsigned short	signed long long	const unsigned short *	`lvxl r,b,a`
vector unsigned short	signed long long	const vector unsigned short *	`lvxl r,b,a`
vector pixel	signed long long	const vector pixel *	`lvxl r,b,a`
vector bool int	signed long long	const vector bool int *	`lvxl r,b,a`
vector signed int	signed long long	const signed int *	`lvxl r,b,a`

r	a	b	Example ISA 3.0 Implementation
vector signed int	signed long long	const vector signed int *	`lvxl r,b,a`
vector unsigned int	signed long long	const unsigned int *	`lvxl r,b,a`
vector unsigned int	signed long long	const vector unsigned int *	`lvxl r,b,a`
vector bool long long	signed long long	const vector bool long long *	`lvxl r,b,a`
vector signed long long	signed long long	const signed long long *	`lvxl r,b,a`
vector signed long long	signed long long	const vector signed long long *	`lvxl r,b,a`
vector unsigned long long	signed long long	const unsigned long long *	`lvxl r,b,a`
vector unsigned long long	signed long long	const vector unsigned long long *	`lvxl r,b,a`
vector float	signed long long	const float *	`lvxl r,b,a`
vector float	signed long long	const vector float *	`lvxl r,b,a`
vector double	signed long long	const double *	`lvxl r,b,a`
vector double	signed long long	const vector double *	`lvxl r,b,a`

vec_loge

Vector Base-2 Logarithm Estimate

```
r = vec_loge (a)
```

Purpose: Returns a vector containing estimates of the base-2 logarithms of the corresponding elements of the source vector.

Result value: Each element of **r** contains an estimated value of the base-2 logarithm of the corresponding element of **a**.

Endian considerations: None.

Table 4.88. Supported type signatures for vec_loge

r	a	Example Implementation
vector float	vector float	`vlogefp r,a`

vec_madd

Vector Multiply-Add

```
r = vec_madd (a, b, c)
```

Purpose: Returns a vector containing the results of performing a fused multiply-add operation for each corresponding set of elements of the source vectors.

Result value: The value of each element of **r** is the product of the values of the corresponding elements of **a** and **b**, added to the value of the corresponding element of **c**.

Endian considerations: None.

Table 4.89. Supported type signatures for vec_madd

r	a	b	c	Example Implementation
vector signed short	vector signed short	vector signed short	vector signed short	`vmladduhm r,a,b,c`
vector signed short	vector signed short	vector unsigned short	vector unsigned short	`vmladduhm r,a,b,c`
vector signed short	vector unsigned short	vector signed short	vector signed short	`vmladduhm r,a,b,c`
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	`vmladduhm r,a,b,c`
vector float	vector float	vector float	vector float	`xvmaddmsp r/a,b,c`
vector double	vector double	vector double	vector double	`xvmaddmdp r/a,b,c`

vec_madds

Vector Multiply-Add Saturated

```
r = vec_madds (a, b, c)
```

Purpose: Returns a vector containing the results of performing a saturated multiply-high-and-add operation for each corresponding set of elements of the source vectors.

Result value: The value of each element of **r** is produced as follows: The values of the corresponding elements of **a** and **b** 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 **c**.

Endian considerations: None.

Table 4.90. Supported type signatures for vec_madds

r	a	b	c	Example Implementation
vector signed short	vector signed short	vector signed short	vector signed short	`vmhaddshs r,a,b,c`

vec_max

Vector Maximum

```
r = vec_max (a, b)
```

Purpose: Returns a vector containing the maximum value from each set of corresponding elements of the source vectors.

Result value: The value of each element of **r** is the maximum of the values of the corresponding elements of **a** and **b**.

Endian considerations: None.

Table 4.91. Supported type signatures for vec_max

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vmaxsb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vmaxub r,a,b`
vector signed short	vector signed short	vector signed short	`vmaxsh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vmaxuh r,a,b`
vector signed int	vector signed int	vector signed int	`vmaxsw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vmaxuw r,a,b`
vector signed long long	vector signed long long	vector signed long long	`vmaxsd r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vmaxud r,a,b`
vector float	vector float	vector float	`xvmaxsp r,a,b`
vector double	vector double	vector double	`xvmaxdp r,a,b`

vec_mergee

Vector Merge Even

```
r = vec_mergee (a, b)
```

Purpose: Merges the even-numbered values from two vectors.

Result value: The even-numbered elements of **a** are stored into the even-numbered elements of **r**. The even-numbered elements of **b** are stored into the odd-numbered elements of **r**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.92. Supported type signatures for vec_mergee

r	a	b	Example LE Implementation	Example BE Implementation
vector bool int	vector bool int	vector bool int	`vmrgow r,b,a`	`vmrgew r,a,b`
vector signed int	vector signed int	vector signed int	`vmrgow r,b,a`	`vmrgew r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vmrgow r,b,a`	`vmrgew r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`
vector signed long long	vector signed long long	vector signed long long	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`
vector float	vector float	vector float	`vmrgow r,b,a`	`vmrgew r,a,b`
vector double	vector double	vector double	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`

vec_mergeh

Vector Merge High

```
r = vec_mergeh (a, b)
```

Purpose: Merges the first halves (in element order) of two vectors.

Result value: The *n*th element of **r**, if *n* is an even number, is given the value of the (*n*/2)th element of **a**. The (*n*+1)th element of **r**, if *n* is an even number, is given the value of the (*n*/2)th element of **b**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.93. Supported type signatures for vec_mergeh

r	a	b	Example LE Implementation	Example BE Implementation
vector bool char	vector bool char	vector bool char	`vmrglb r,b,a`	`vmrghb r,a,b`
vector signed char	vector signed char	vector signed char	`vmrglb r,b,a`	`vmrghb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vmrglb r,b,a`	`vmrghb r,a,b`
vector bool short	vector bool short	vector bool short	`vmrglh r,b,a`	`vmrghh r,a,b`
vector signed short	vector signed short	vector signed short	`vmrglh r,b,a`	`vmrghh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vmrglh r,b,a`	`vmrghh r,a,b`
vector pixel	vector pixel	vector pixel	`vmrglh r,b,a`	`vmrghh r,a,b`
vector bool int	vector bool int	vector bool int	`vmrglw r,b,a`	`vmrghw r,a,b`
vector signed int	vector signed int	vector signed int	`vmrglw r,b,a`	`vmrghw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vmrglw r,b,a`	`vmrghw r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`
vector signed long long	vector signed long long	vector signed long long	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`
vector float	vector float	vector float	`vmrglw r,b,a`	`vmrghw r,a,b`

r	a	b	Example LE Implementation	Example BE Implementation
vector double	vector double	vector double	`xxpermdi r,b,a,3`	`xxpermdi r,a,b,0`

vec_mergel

Vector Merge Low

```
r = vec_mergel (a, b)
```

Purpose: Merges the last halves (in element order) of two vectors.

Result value: Let *m* be the number of elements in **r**. The *n*th element of **r**, if *n* is an even number, is given the value of the *m*/2 + (*n*/2)th element of **a**. The (*n*+1)th element of **r**, if *n* is an even number, is given the value of the *m*/2 + (*n*/2)th element of **b**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.94. Supported type signatures for vec_mergel

r	a	b	Example LE Implementation	Example BE Implementation
vector bool char	vector bool char	vector bool char	`vmrghb r,b,a`	`vmrglb r,a,b`
vector signed char	vector signed char	vector signed char	`vmrghb r,b,a`	`vmrglb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vmrghb r,b,a`	`vmrglb r,a,b`
vector bool short	vector bool short	vector bool short	`vmrghh r,b,a`	`vmrglh r,a,b`
vector signed short	vector signed short	vector signed short	`vmrghh r,b,a`	`vmrglh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vmrghh r,b,a`	`vmrglh r,a,b`
vector pixel	vector pixel	vector pixel	`vmrghh r,b,a`	`vmrglh r,a,b`
vector bool int	vector bool int	vector bool int	`vmrghw r,b,a`	`vmrglw r,a,b`
vector signed int	vector signed int	vector signed int	`vmrghw r,b,a`	`vmrglw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vmrghw r,b,a`	`vmrglw r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`
vector signed long long	vector signed long long	vector signed long long	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`
vector float	vector float	vector float	`vmrghw r,b,a`	`vmrglw r,a,b`

r	a	b	Example LE Implementation	Example BE Implementation
vector double	vector double	vector double	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`

vec_mergeo

Vector Merge Odd

```
r = vec_mergeo (a, b)
```

Purpose: Merges the odd-numbered values from two vectors.

Result value: The odd-numbered elements of **a** are stored into the even-numbered elements of **r**. The odd-numbered elements of **b** are stored into the odd-numbered elements of **r**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.95. Supported type signatures for vec_mergeo

r	a	b	Example LE Implementation	Example BE Implementation
vector bool int	vector bool int	vector bool int	`vmrgew r,b,a`	`vmrgow r,a,b`
vector signed int	vector signed int	vector signed int	`vmrgew r,b,a`	`vmrgow r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vmrgew r,b,a`	`vmrgow r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`
vector signed long long	vector signed long long	vector signed long long	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`
vector float	vector float	vector float	`vmrgew r,b,a`	`vmrgow r,a,b`
vector double	vector double	vector double	`xxpermdi r,b,a,0`	`xxpermdi r,a,b,3`

vec_mfvscr

Vector Move From Vector Status and Control Register

```
r = vec_mfvscr ()
```

Purpose: Copies the contents of the Vector Status and Control Register (VSCR) into the result vector.

Result value: The high-order 16 bits of the VSCR are copied into the seventh element of **r**, using big-endian (left-to-right) order. The low-order 16 bits of the VSCR are copied into the eighth element of **r**, using big-endian order. All other elements of **r** are set to zero.

Endian considerations: The contents of the VSCR are placed in the low-order 32 bits of the result vector, regardless of endianness.

Notes: The use of `vector unsigned short` as the result type eases access to the two bits currently defined in the VSCR. Following execution of `vec_mfvscr`, **r**[6] will contain $VSCR_{NJ}$ in the low-order bit, and **r**[7] will contain $VSCR_{SAT}$ in the low-order bit.

Table 4.96. Supported type signatures for vec_mfvscr

r	Example Implementation
vector unsigned short	`mfvscr a`

vec_min

Vector Minimum

```
r = vec_min (a, b)
```

Purpose: Returns a vector containing the minimum value from each set of corresponding elements of the source vectors.

Result value: The value of each element of **r** is the minimum of the values of the corresponding elements of **a** and **b**.

Endian considerations: None.

Table 4.97. Supported type signatures for vec_min

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vminsb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vminub r,a,b`
vector signed short	vector signed short	vector signed short	`vminsh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vminuh r,a,b`
vector signed int	vector signed int	vector signed int	`vminsw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vminuw r,a,b`
vector signed long long	vector signed long long	vector signed long long	`vminsd r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vminud r,a,b`
vector float	vector float	vector float	`xvminsp r,a,b`
vector double	vector double	vector double	`xvmindp r,a,b`

vec_mradds

Vector Multiply-High Round and Add Saturated

```
r = vec_mradds (a, b, c)
```

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 source vectors.

Result value: The value of each element of **r** is produced as follows. The values of the corresponding elements of **a** and **b** 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 **c**.

Endian considerations: None.

Table 4.98. Supported type signatures for vec_mradds

r	a	b	c	Example Implementation
vector signed short	vector signed short	vector signed short	vector signed short	`vmhraddshs r,a,b,c`

vec_msub

Vector Multiply-Subtract

```
r = vec_msub (a, b, c)
```

Purpose: Returns a vector containing the results of performing a multiply-subtract operation using the source vectors.

Result value: Each element of **r** is produced by multiplying the corresponding element of **a** by the corresponding element of **b** and then subtracting the corresponding element of **c**.

Endian considerations: None.

Table 4.99. Supported type signatures for vec_msub

r	a	b	c	Example Implementation
vector float	vector float	vector float	vector float	`xvmsubmsp r/a,b,c`
vector double	vector double	vector double	vector double	`xvmsubmdp r/a,b,c`

vec_msum

Vector Multiply-Sum

```
r = vec_msum (a, b, c)
```

Purpose: Returns a vector containing the results of performing a multiply-sum operation using the source vectors.

Result value: There are two cases:

- When **a** is of type vector signed char or vector unsigned char, each word element of **r** is computed as follows:
  - Each of the four byte elements contained in the corresponding word element of **a** is multiplied by the corresponding byte element of **b**.
  - The sum of these four halfword products is added to the corresponding word element in **c** and placed in the corresponding word element of **r**.
- When **a** is of type vector signed short or vector unsigned short, each word element of **r** is computed as follows:
  - Each of the two halfword elements contained in the corresponding word element of **a** is multiplied by the corresponding halfword element of **b**.
  - The sum of these two word products is added to the corresponding word element in **c** and placed in the corresponding word element of **r**.

All operations are performed using 32-bit modular arithmetic.

Endian considerations: None.

Table 4.100. Supported type signatures for vec_msum

r	a	b	c	Example Implementation
vector signed int	vector signed char	vector unsigned char	vector signed int	`vmsummbm r,a,b,c`
vector signed int	vector signed short	vector signed short	vector signed int	`vmsumshm r,a,b,c`
vector unsigned int	vector unsigned char	vector unsigned char	vector unsigned int	`vmsumubm r,a,b,c`
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	`vmsumuhm r,a,b,c`

vec_msums

Vector Multiply-Sum Saturated

```
r = vec_msums (a, b, c)
```

Purpose: Returns a vector containing the results of performing a saturated multiply-sum operation using the source vectors.

Result value: Assume that the elements of each vector are numbered beginning with 0. The value of each element *n* of **r** is obtained as follows. For *p* = 2*n* to 2*n*+1, multiply element *p* of **a** by element *p* of **b**. Add the sum of these products to element *n* of **c**. All additions are performed using 32-bit saturated arithmetic.

Endian considerations: None.

Table 4.101. Supported type signatures for vec_msums

r	a	b	c	Example Implementation
vector signed int	vector signed short	vector signed short	vector signed int	`vmsumshs r,a,b,c`
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	`vmsumuhs r,a,b,c`

vec_mtvscr

Vector Move to Vector Status and Control Register

```
vec_mtvscr (a)
```

Purpose: Copies a value into the Vector Status and Control Register (VSCR). The low-order 32 bits of **a** are copied into the VSCR.

Result value: None.

Endian considerations: None.

Table 4.102. Supported type signatures for vec_mtvscr

a	Example Implementation
vector bool char	`mtvscr a`
vector signed char	`mtvscr a`
vector unsigned char	`mtvscr a`
vector bool short	`mtvscr a`
vector signed short	`mtvscr a`
vector unsigned short	`mtvscr a`
vector pixel	`mtvscr a`
vector bool int	`mtvscr a`
vector signed int	`mtvscr a`
vector unsigned int	`mtvscr a`

vec_mul

Vector Multiply

```
r = vec_mul (a, b)
```

Purpose: Compute the products of corresponding elements of two vectors.

Result value: Each element of **r** receives the product of the corresponding elements of **a** and **b**.

Endian considerations: None.

Notes:

- The example implementation for vector char assumes that the address of the permute control vector for the vperm instruction is in a register identified by pcv. Its value is {1,17,3,19,5,21,7,23,9,25,11,27,13,29,15,31}.
- There are currently no vector instructions to support vector long long multiplication, so the compiler must perform two scalar multiplies on the vector elements for this case.

Table 4.103. Supported type signatures for vec_mul

r	a	b	Example Implementation

vector signed char	vector signed char	vector signed char	`vmulesb t,a,b`
			`vmulosb u,a,b`
			`lxvw4x v,0,pcv`
			`vperm r,t,u,v`

vector unsigned char	vector unsigned char	vector unsigned char	`vmulesb t,a,b`
			`vmulosb u,a,b`
			`lxvw4x v,0,pcv`
			`vperm r,t,u,v`

vector signed short	vector signed short	vector signed short	`xxspltib t,0` `vmladduhm r,a,b,t`
vector unsigned short	vector unsigned short	vector unsigned short	`xxspltib t,0` `vmladduhm r,a,b,t`
vector signed int	vector signed int	vector signed int	`vmuluwm r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vmuluwm r,a,b`
vector signed long long	vector signed long long	vector signed long long	`[scalarized]`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`[scalarized]`
vector float	vector float	vector float	`xvmulsp r,a,b`
vector double	vector double	vector double	`xvmuldp r,a,b`

vec_mule

Vector Multiply Even

```
r = vec_mule (a, b)
```

Purpose: Multiplies the even-numbered elements of the source vectors to produce the target vector.

Result value: Each element *n* of **r** is the product of element 2*n* of **a** and element 2*n* of **b**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.104. Supported type signatures for vec_mule

r	a	b	Example LE Implementation	Example BE Implementation
vector signed short	vector signed char	vector signed char	`vmulosb r,a,b`	`vmulesb r,a,b`
vector unsigned short	vector unsigned char	vector unsigned char	`vmuloub r,a,b`	`vmuleub r,a,b`
vector signed int	vector signed short	vector signed short	`vmulosh r,a,b`	`vmulesh r,a,b`
vector unsigned int	vector unsigned short	vector unsigned short	`vmulouh r,a,b`	`vmuleuh r,a,b`
vector signed long long	vector signed int	vector signed int	`vmulosw r,a,b`	`vmulesw r,a,b`
vector unsigned long long	vector unsigned int	vector unsigned int	`vmulouw r,a,b`	`vmuleuw r,a,b`

vec_mulo

Vector Multiply Odd

```
r = vec_mulo (a, b)
```

Purpose: Multiplies the odd-numbered elements of the source vectors to produce the target vector.

Result value: Each element *n* of **r** is the product of element 2*n*+1 of **a** and element 2*n*+1 of **b**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.105. Supported type signatures for vec_mulo

r	a	b	Example LE Implementation	Example BE Implementation
vector signed short	vector signed char	vector signed char	`vmulesb r,a,b`	`vmulosb r,a,b`
vector unsigned short	vector unsigned char	vector unsigned char	`vmuleub r,a,b`	`vmuloub r,a,b`
vector signed int	vector signed short	vector signed short	`vmulesh r,a,b`	`vmulosh r,a,b`
vector unsigned int	vector unsigned short	vector unsigned short	`vmuleuh r,a,b`	`vmulouh r,a,b`
vector signed long long	vector signed int	vector signed int	`vmulesw r,a,b`	`vmulosw r,a,b`
vector unsigned long long	vector unsigned int	vector unsigned int	`vmuleuw r,a,b`	`vmulouw r,a,b`

vec_nabs

Vector Negated Absolute Value

```
r = vec_nabs (a)
```

Purpose: Returns a vector containing the negated absolute values of the contents of the source vector.

Result value: The value of each element of **r** is the negated absolute value of the corresponding element of **a**. For integer vectors, the arithmetic is modular.

Endian considerations: None.

Table 4.106. Supported type signatures for vec_nabs

r	a	Example Implementation

vector signed char	vector signed char	`vspltisw t,0`
		`vsububm u,t,a`
		`vminsb r,u,a`

vector signed short	vector signed short	`vspltisw t,0`
		`vsubuhm u,t,a`
		`vminsh r,u,a`

vector signed int	vector signed int	`vspltisw t,0`
		`vsubuwm u,t,a`
		`vminsw r,u,a`

vector signed long long	vector signed long long	`vspltisw t,0`
		`vsubudm u,t,a`
		`vminsd r,u,a`

vector float	vector float	`xvnabssp r,a`
vector double	vector double	`xvnabsdp r,a`

vec_nand

Vector NAND

```
r = vec_nand (a, b)
```

Purpose: Performs a bitwise NAND of two vectors.

Result value: **r** is the bitwise NAND of **a** and **b**.

Endian considerations: None.

Table 4.107. Supported type signatures for vec_nand

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`xxlnand r,a,b`
vector signed char	vector signed char	vector signed char	`xxlnand r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`xxlnand r,a,b`
vector bool short	vector bool short	vector bool short	`xxlnand r,a,b`
vector signed short	vector signed short	vector signed short	`xxlnand r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`xxlnand r,a,b`
vector bool int	vector bool int	vector bool int	`xxlnand r,a,b`
vector signed int	vector signed int	vector signed int	`xxlnand r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`xxlnand r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxlnand r,a,b`
vector signed long long	vector signed long long	vector signed long long	`xxlnand r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxlnand r,a,b`
vector float	vector float	vector float	`xxlnand r,a,b`
vector double	vector double	vector double	`xxlnand r,a,b`

vec_ncipher_be

Vector AES Inverse Cipher Big-Endian

```
r = vec_ncipher_be (a, b)
```

Purpose: Performs one round of the AES inverse cipher operation on an intermediate state array **a** by using a given round key **b**.

Result value: r contains the resulting intermediate state, after one round of the AES inverse cipher operation on intermediate state array **a**, using the round key specified by **b**.

Endian considerations: All element and bit numberings of the AES inverse cipher operation use big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_ncipher_be` does not follow the bi-endian programming model.

Table 4.108. Supported type signatures for vec_ncipher_be

r	a	b	Example Implementation
vector unsigned char	vector unsigned char	vector unsigned char	`vncipher r,a,b`

vec_ncipherlast_be

Vector AES Inverse Cipher Last Big-Endian

```
r = vec_ncipherlast_be (a, b)
```

Purpose: Performs the final round of the AES inverse cipher operation on an intermediate state array **a** using the specified round key **b**.

Result value: r contains the resulting final state, after the final round of the AES inverse cipher operation on intermediate state array **a**, using the round key specified by **b**.

Endian considerations: All element and bit numberings of the AES inverse cipher-last operation use big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_ncipherlast_be` does not follow the bi-endian programming model.

Table 4.109. Supported type signatures for vec_ncipherlast_be

r	a	b	Example Implementation
vector unsigned char	vector unsigned char	vector unsigned char	`vncipherlast r,a,b`

vec_nearbyint

Vector Nearby Integer

```
r = vec_nearbyint (a)
```

Purpose: Returns a vector containing the floating-point integral values nearest to the values of the corresponding elements of the source vector.

Result value: Each element of **r** contains the nearest representable floating-point integral value to the value of the corresponding element of **a**. When an input element value is exactly between two integer values, the input value with the larger absolute value is selected. The current floating-point rounding mode is ignored.

Endian considerations: None.

Table 4.110. Supported type signatures for vec_nearbyint

r	a	Example Implementation
vector float	vector float	`xvrspi r,a`
vector double	vector double	`xvrdpi r,a`

vec_neg

Vector Negate

```
r = vec_neg (a)
```

Purpose: Returns a vector containing the negated values of the contents of the source vector.

Result value: The value of each element of **r** is the negated value of the corresponding element of **a**. For integer vectors, the arithmetic is modular.

Endian considerations: None.

Table 4.111. Supported type signatures for vec_neg

r	a	Example Implementation
vector signed char	vector signed char	`vspltisw t,0` `vsububm r,t,a`
vector signed short	vector signed short	`vspltisw t,0` `vsubuhm r,t,a`
vector signed int	vector signed int	`vspltisw t,0` `vsubuwm r,t,a`
vector signed long long	vector signed long long	`vspltisw t,0` `vsubudm r,t,a`
vector float	vector float	`xvnegsp r,a`
vector double	vector double	`xvnegdp r,a`

vec_nmadd

Vector Negated Multiply-Add

```
r = vec_nmadd (a, b, c)
```

Purpose: Returns a vector containing the results of performing a negated multiply-add operation on the source vectors.

Result value: The value of each element of **r** is the product of the corresponding elements of **a** and **b**, added to the corresponding elements of **c**, then multiplied by –1.0.

Endian considerations: None.

Table 4.112. Supported type signatures for vec_nmadd

r	a	b	c	Example Implementation
vector float	vector float	vector float	vector float	`xvnmaddasp r/c,a,b`
vector double	vector double	vector double	vector double	`xvnmaddadp r/c,a,b`

vec_nmsub

Vector Negated Multiply-Subtract

```
r = vec_nmsub (a, b, c)
```

Purpose: Returns a vector containing the results of performing a negated multiply-subtract operation on the source vectors.

Result value: The value of each element of **r** is the value of the corresponding element of **c** subtracted from the product of the corresponding elements of **a** and **b**, and then multiplied by –1.0.

Endian considerations: None.

Table 4.113. Supported type signatures for vec_nmsub

r	a	b	c	Example Implementation
vector float	vector float	vector float	vector float	`xvnmsubmsp r/a,b,c`
vector double	vector double	vector double	vector double	`xvnmsubmdp r/a,b,c`

vec_nor

Vector NOR

```
r = vec_nor (a, b)
```

Purpose: Performs a bitwise NOR of two vectors.

Result value: **r** is the bitwise NOR of **a** and **b**.

Endian considerations: None.

Table 4.114. Supported type signatures for vec_nor

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`xxlnor r,a,b`
vector signed char	vector signed char	vector signed char	`xxlnor r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`xxlnor r,a,b`
vector bool short	vector bool short	vector bool short	`xxlnor r,a,b`
vector signed short	vector signed short	vector signed short	`xxlnor r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`xxlnor r,a,b`
vector bool int	vector bool int	vector bool int	`xxlnor r,a,b`
vector signed int	vector signed int	vector signed int	`xxlnor r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`xxlnor r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxlnor r,a,b`
vector signed long long	vector signed long long	vector signed long long	`xxlnor r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxlnor r,a,b`
vector float	vector float	vector float	`xxlnor r,a,b`
vector double	vector double	vector double	`xxlnor r,a,b`

vec_or

Vector OR

```
r = vec_or (a, b)
```

Purpose: Performs a bitwise OR of two vectors.

Result value: **r** is the bitwise OR of **a** and **b**.

Endian considerations: None.

Table 4.115. Supported type signatures for vec_or

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`xxlor r,a,b`
vector signed char	vector signed char	vector signed char	`xxlor r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`xxlor r,a,b`
vector bool short	vector bool short	vector bool short	`xxlor r,a,b`
vector signed short	vector signed short	vector signed short	`xxlor r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`xxlor r,a,b`
vector bool int	vector bool int	vector bool int	`xxlor r,a,b`
vector signed int	vector signed int	vector signed int	`xxlor r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`xxlor r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxlor r,a,b`
vector signed long long	vector signed long long	vector signed long long	`xxlor r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxlor r,a,b`
vector float	vector float	vector float	`xxlor r,a,b`
vector double	vector double	vector double	`xxlor r,a,b`

vec_orc

Vector OR with Complement

```
r = vec_orc (a, b)
```

Purpose: Performs a bitwise OR of the first vector with the bitwise-complemented second vector.

Result value: **r** is the bitwise OR of **a** and the bitwise complement of **b**.

Endian considerations: None.

Table 4.116. Supported type signatures for vec_orc

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`xxlorc r,a,b`
vector signed char	vector signed char	vector signed char	`xxlorc r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`xxlorc r,a,b`
vector bool short	vector bool short	vector bool short	`xxlorc r,a,b`
vector signed short	vector signed short	vector signed short	`xxlorc r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`xxlorc r,a,b`
vector bool int	vector bool int	vector bool int	`xxlorc r,a,b`
vector signed int	vector signed int	vector signed int	`xxlorc r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`xxlorc r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxlorc r,a,b`
vector signed long long	vector signed long long	vector signed long long	`xxlorc r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxlorc r,a,b`
vector float	vector float	vector float	`xxlorc r,a,b`
vector double	vector double	vector double	`xxlorc r,a,b`

vec_pack

Vector Pack

```
r = vec_pack (a, b)
```

Purpose: Packs information from each element of two vectors into the result vector.

Result value: Let **v** represent the concatenation of vectors **a** and **b**. For integer types, the value of each element of **r** is taken from the low-order half of the corresponding element of **v**. For floating-point types, the value of each element of **r** is the corresponding element of **v**, rounded to the result type.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.117. Supported type signatures for vec_pack

r	a	b	Example LE Implementation	Example BE Implementation
vector bool char	vector bool short	vector bool short	`vpkuhum r,b,a`	`vpkuhum r,a,b`
vector signed char	vector signed short	vector signed short	`vpkuhum r,b,a`	`vpkuhum r,a,b`
vector unsigned char	vector unsigned short	vector unsigned short	`vpkuhum r,b,a`	`vpkuhum r,a,b`
vector bool short	vector bool int	vector bool int	`vpkuwum r,b,a`	`vpkuwum r,a,b`
vector signed short	vector signed int	vector signed int	`vpkuwum r,b,a`	`vpkuwum r,a,b`
vector unsigned short	vector unsigned int	vector unsigned int	`vpkuwum r,b,a`	`vpkuwum r,a,b`
vector bool int	vector bool long long	vector bool long long	`vpkudum r,b,a`	`vpkudum r,a,b`
vector signed int	vector signed long long	vector signed long long	`vpkudum r,b,a`	`vpkudum r,a,b`
vector unsigned int	vector unsigned long long	vector unsigned long long	`vpkudum r,b,a`	`vpkudum r,a,b`

vector float	vector double	vector double	`xxpermdi t,b,a,0`
			`xxpermdi u,b,a,3`
			`xvcvdpsp t,t`
			`xvcvdpsp u,u`
			`vmrgow r,t,u`
				`xxpermdi t,a,b,0`
				`xxpermdi u,a,b,3`
				`xvcvdpsp t,t`
				`xvcvdpsp u,u`
				`vmrgew r,t,u`

vec_pack_to_short_fp32

Vector Pack 32-bit Float to Short

```
r = vec_pack_to_short_fp32 (a, b)
```

Purpose: Packs eight single-precision 32-bit floating-point numbers from two source vectors into a vector of eight 16-bit floating-point numbers.

Result value: Let **v** represent the 16-element concatenation of **a** and **b**. Each value of **r** contains the result of converting the corresponding single-precision element of **v** to half-precision.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.118. Supported type signatures for vec_pack_to_short_fp32

r	a	b	Example LE Implementation	Example BE Implementation	Restrictions

vector unsigned short	vector float	vector float	`xvcvsphp t,a`
			`xvcvsphp u,b`
			`vpkuwum r,t,u`
				`xvcvsphp t,a`
				`xvcvsphp u,b`
				`vpkuwum r,u,t`
					ISA 3.0 or later

vec_packpx

Vector Pack Pixel

```
r = vec_packpx (a, b)
```

Purpose: Packs information from each element of two vectors into the result vector.

Result value: Let **v** be the concatenation of **a** and **b**. The value of each element of **r** is taken from the corresponding element of **v** as follows:

- 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.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.119. Supported type signatures for vec_packpx

r	a	b	Example LE Implementation	Example BE Implementation
vector pixel	vector unsigned int	vector unsigned int	`vpkpx r,b,a`	`vpkpx r,a,b`

vec_packs

Vector Pack Saturated

```
r = vec_packs (a, b)
```

Purpose: Packs information from each element of two vectors into the result vector, using saturated values.

Result value: Let **v** be the concatenation of **a** and **b**. The value of each element of **r** is the saturated value of the corresponding element of **v**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.120. Supported type signatures for vec_packs

r	a	b	Example LE Implementation	Example BE Implementation
vector signed char	vector signed short	vector signed short	`vpkshss r,b,a`	`vpkshss r,a,b`
vector unsigned char	vector unsigned short	vector unsigned short	`vpkuhus r,b,a`	`vpkuhus r,a,b`
vector signed short	vector signed int	vector signed int	`vpkswss r,b,a`	`vpkswss r,a,b`
vector unsigned short	vector unsigned int	vector unsigned int	`vpkuwus r,b,a`	`vpkuwus r,a,b`
vector signed int	vector signed long long	vector signed long long	`vpksdss r,b,a`	`vpksdss r,a,b`
vector unsigned int	vector unsigned long long	vector unsigned long long	`vpkudus r,b,a`	`vpkudus r,a,b`

vec_packsu

Vector Pack Saturated Unsigned

```
r = vec_packsu (a, b)
```

Purpose: Packs information from each element of two vectors into the result vector, using unsigned saturated values.

Result value: Let **v** be the concatenation of **a** and **b**. The value of each element of **r** is the saturated value of the corresponding element of **v**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.121. Supported type signatures for vec_packsu

r	a	b	Example LE Implementation	Example BE Implementation
vector unsigned char	vector signed short	vector signed short	`vpkshus r,b,a`	`vpkshus r,a,b`
vector unsigned char	vector unsigned short	vector unsigned short	`vpkuhus r,b,a`	`vpkuhus r,a,b`
vector unsigned short	vector signed int	vector signed int	`vpkswus r,b,a`	`vpkswus r,a,b`
vector unsigned short	vector unsigned int	vector unsigned int	`vpkuwus r,b,a`	`vpkuwus r,a,b`
vector unsigned int	vector signed long long	vector signed long long	`vpksdus r,b,a`	`vpksdus r,a,b`
vector unsigned int	vector unsigned long long	vector unsigned long long	`vpkudus r,b,a`	`vpkudus r,a,b`

vec_parity_lsbb

Vector Parity over Least-Significant Bits of Bytes

```
r = vec_parity_lsbb (a)
```

Purpose: Compute parity on the least-significant bit of each byte.

Result value: Each element of **r** contains the parity computed over the low-order bit of each of the bytes in the corresponding element of **a**.

An example for input **a** of type vector unsigned int follows:

word index	***0***				***1***				***2***				***3***			
***byte index* n**	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	*01010203*				*05080D15*				*22375990*				*E97962E1*			
bytes of **a**	01	01	02	03	05	08	0D	15	22	37	59	90	E9	79	62	E1
least-significant bit of byte n of **a**	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1
r	00000001				00000001				00000000				00000001			

Endian considerations: None.

Table 4.122. Supported type signatures for vec_parity_lsbb

r	a	Example Implementation	Restrictions
vector unsigned int	vector signed int	`vprtybw r,a`	ISA 3.0 or later
vector unsigned int	vector unsigned int	`vprtybw r,a`	ISA 3.0 or later
vector unsigned long long	vector signed long long	`vprtybd r,a`	ISA 3.0 or later
vector unsigned long long	vector unsigned long long	`vprtybd r,a`	ISA 3.0 or later
vector unsigned __int128	vector signed __int128	`vprtybq r,a`	ISA 3.0 or later
vector unsigned __int128	vector unsigned __int128	`vprtybq r,a`	ISA 3.0 or later

vec_perm

Vector Permute

```
r = vec_perm (a, b, c)
```

Purpose: Returns a vector that contains elements selected from two vectors, in the order specified by a third vector.

Result value: Let **v** be the concatenation of **a** and **b**. Each byte of **r** selected by using the least-significant 5 bits of the corresponding byte of **c** as an index into **v**.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Notes:

- The example little-endian code generation uses the **vpermr** instruction from ISA 3.0. For earlier targets, the compiler must generate an extra instruction to adjust the permute control vector **c**.
- The `vec_perm` built-in should only use permutations that reorder vector elements of the specified type, not to reorder bytes within those elements. The results are not guaranteed to be consistent across big- and little-endian if you violate this rule. See Section 2.7.3, “Limitations on bi-endianness of vec_perm” [16].

Table 4.123. Supported type signatures for vec_perm

r	a	b	c	Example LE Implementation	Example BE Implementation
vector bool char	vector bool char	vector bool char	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector signed char	vector signed char	vector signed char	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector bool short	vector bool short	vector bool short	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector signed short	vector signed short	vector signed short	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector pixel	vector pixel	vector pixel	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector bool int	vector bool int	vector bool int	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector signed int	vector signed int	vector signed int	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`

r	a	b	c	Example LE Implementation	Example BE Implementation
vector bool long long	vector bool long long	vector bool long long	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector signed long long	vector signed long long	vector signed long long	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector float	vector float	vector float	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`
vector double	vector double	vector double	vector unsigned char	`vpermr r,b,a,c`	`vperm r,a,b,c`

vec_permxor

Vector Permute and Exclusive-OR

```
r = vec_permxor (a, b, c)
```

Purpose: Applies a permute and exclusive-OR operation on two vectors of byte elements, with the selected elements identified by a third vector.

Result value: For each *i* (0 ≤ *i* < 16), let *x* be bits 0–3 and *y* be bits 4–7 of byte element *i* of **c**. Byte element *i* of **r** is set to the exclusive-OR of byte elements *x* of **a** and *y* of **b**.

An example for input **a** of type vector unsigned char follows:

byte index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF
b	FF	EF	DF	CF	BF	AF	9F	8F	7F	6F	5F	4F	3F	2F	1F	0F
c	01	23	45	67	89	AB	CD	EF	F0	E1	D2	C3	B4	A5	96	87
x *y*	0 1	2 3	4 5	6 7	8 9	A B	C D	E F	F 0	E 1	D 2	C 3	B 4	A 5	9 6	8 7
a_x b_y	F0 EF	F2 CF	F4 AF	F6 8F	F8 6F	FA 4F	FC 2F	FE 0F	FF FF	FE EF	FD DF	FC CF	FB BF	FA AF	F9 9F	F8 8F
r	1F	3D	5B	79	97	B5	D3	F1	00	11	22	33	44	55	66	77

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.124. Supported type signatures for vec_permxor

r	a	b	c	Example LE Implementation	Example BE Implementation
vector bool char	vector bool char	vector bool char	vector bool char	`xxlnor t,c,c` `vpermxor r,a,b,t`	`vpermxor r,a,b,c`
vector signed char	vector signed char	vector signed char	vector signed char	`xxlnor t,c,c` `vpermxor r,a,b,t`	`vpermxor r,a,b,c`
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	`xxlnor t,c,c` `vpermxor r,a,b,t`	`vpermxor r,a,b,c`

vec_pmsum_be

Vector Polynomial Multiply-Sum Big-Endian

```
r = vec_pmsum_be (a, b)
```

Purpose: Performs the exclusive-OR operation (implementing polynomial addition) on each even-odd pair of the polynomial-multiplication result of the corresponding elements of **a** and **b**.

Result value: Each element *i* of **r** is computed by an exclusive-OR operation of the polynomial multiplication of input elements 2 × *i* of **a** and **b** and input elements 2 × *i* + 1 of **a** and **b**.

An example follows for inputs of type vector unsigned int:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
a	A3000000	00A20000	0000A100	000000A0
b	00B30000	0000B200	000000B1	B00000B0
binary polynomial multiplicands	A3000000 00B30000	00A20000 0000B200	0000A100 000000B1	000000A0 B00000B0
intermediate results *XOR operands*	004E350000000000	0000004E24000000	00000000004E1100	0000004E00004E00
r	004E354E24000000		0000004E004E5F00	

Endian considerations: All element numberings in the above description denote big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_pmsum_be` does not follow the bi-endian programming model.

Table 4.125. Supported type signatures for vec_pmsum_be

r	a	b	Example Implementation
vector unsigned short	vector unsigned char	vector unsigned char	`vpmsumb r,a,b`
vector unsigned int	vector unsigned short	vector unsigned short	`vpmsumh r,a,b`
vector unsigned long long	vector unsigned int	vector unsigned int	`vpmsumw r,a,b`
vector unsigned __int128	vector unsigned long long	vector unsigned long long	`vpmsumd r,a,b`

vec_popcnt

Vector Population Count

```
r = vec_popcnt (a)
```

Purpose: Returns a vector containing the number of bits set in each element of the source vector.

Result value: The value of each element of **r** is the number of bits set in the corresponding element of **a**.

Endian considerations: None.

Table 4.126. Supported type signatures for vec_popcnt

r	a	Example Implementation
vector unsigned char	vector signed char	`vpopcntb r,a`
vector unsigned char	vector unsigned char	`vpopcntb r,a`
vector unsigned short	vector signed short	`vpopcnth r,a`
vector unsigned short	vector unsigned short	`vpopcnth r,a`
vector unsigned int	vector signed int	`vpopcntw r,a`
vector unsigned int	vector unsigned int	`vpopcntw r,a`
vector unsigned long long	vector signed long long	`vpopcntd r,a`
vector unsigned long long	vector unsigned long long	`vpopcntd r,a`

vec_re

Vector Reciprocal Estimate

```
r = vec_re (a)
```

Purpose: Returns a vector containing estimates of the reciprocals of the corresponding elements of the source vector.

Result value: Each element of **r** contains the estimated value of the reciprocal of the corresponding element of **a**.

An example for input **a** of type vector double follows:

doubleword index	*0*	*1*
a	-1.00000000000000000	4.00000000000000000
r	-0.99996948242187500	0.249992370605468750

Endian considerations: None.

Notes: For finite reciprocals, this intrinsic guarantees at least 14 bits of accuracy.

Table 4.127. Supported type signatures for vec_re

r	a	Example Implementation
vector float	vector float	`xvresp r,a`
vector double	vector double	`xvredp r,a`

vec_recipdiv

Vector Reciprocal Divide

```
r = vec_recipdiv (a, b)
```

Purpose: Returns a vector containing refined approximations of the division of the corresponding elements of **a** by the corresponding elements of **b**.

Result value: Each element of **r** contains a refined approximation of the division of the corresponding element of **a** by the corresponding element of **b**.

Endian considerations: None.

Notes:

- The example implementation for vector double assumes that a register *z* initially contains the double-precision floating-point value 1.0 in each doubleword.
- For finite reciprocals, this intrinsic guarantees at least 23 bits of accuracy for single-precision floating point, and at least 52 bits of accuracy for double-precision floating point.
- 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.

Table 4.128. Supported type signatures for vec_recipdiv

r	a	b	Example Implementation

vector float	vector float	vector float	`xvresp t,b`
			`xvmulsp u,a,t`
			`xvnmsubasp r/a,b,u`
			`xvmaddmsp r/a,t,u`

vector double	vector double	vector double	`xvredp t,b`
			`xvnmsubadp z,b,t`
			`xvmaddadp u,z,t`
			`xvmuldp v,a,u`
			`xvnmsubadp r/a,b,v`
			`xvmaddmdp r/a,u,v`

vec_revb

Vector Reverse Bytes

```
r = vec_revb (a)
```

Purpose: Reverse the bytes of each vector element of a vector.

Result value: Each element of **r** contains the byte-reversed value of the corresponding element of **a**.

Endian considerations: None.

Notes:

- The examples shown are for ISA 3.0. More complex sequences are required for earlier ISA levels.
- Interfaces that make no change to the data are deprecated.

Table 4.129. Supported type signatures for vec_revb

r	a	Example ISA 3.0 Implementation	Restrictions
vector bool char	vector bool char	`[none]`	Deprecated
vector signed char	vector signed char	`[none]`	Deprecated
vector unsigned char	vector unsigned char	`[none]`	Deprecated
vector bool short	vector bool short	`xxbrh r,a`	Deprecated
vector signed short	vector signed short	`xxbrh r,a`	
vector unsigned short	vector unsigned short	`xxbrh r,a`	
vector bool int	vector bool int	`xxbrw r,a`	Deprecated
vector signed int	vector signed int	`xxbrw r,a`	
vector unsigned int	vector unsigned int	`xxbrw r,a`	
vector bool long long	vector bool long long	`xxbrd r,a`	Deprecated
vector signed long long	vector signed long long	`xxbrd r,a`	
vector unsigned long long	vector unsigned long long	`xxbrd r,a`	
vector signed __int128	vector signed __int128	`xxbrq r,a`	

r	a	Example ISA 3.0 Implementation	Restrictions
vector unsigned __int128	vector unsigned __int128	`xxbrq r,a`	
vector float	vector float	`xxbrw r,a`	
vector double	vector double	`xxbrd r,a`	

vec_reve

Vector Reverse Elements

```
r = vec_reve (a)
```

Purpose: Reverse the elements of a vector.

Result value: Returns a vector with the elements of the source vector in reversed order.

Endian considerations: The vpermr instruction is most naturally used to implement this built-in function for a little-endian target, and the vperm instruction for a big-endian target. This is not technically necessary, however, provided the correct permute control vector is used. Note that use of vpermr requires ISA 3.0.

Notes: The example implementations assume that the permute control vector for the vperm or vpermr instruction is in a register identified by pcv. The value of pcv differs based on the element size, and is the same (in natural element order) for big- and little-endian, assuming the use of vperm for big-endian and vpermr for little-endian.

Vector types	Permute control vector
vector char	{ 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 }
vector short	{ 14, 15, 12, 13, 10, 11, 8, 9, 6, 7, 4, 5, 2, 3, 0, 1 }
vector int, vector float	{ 12, 13, 14, 15, 8, 9, 10, 11, 4, 5, 6, 7, 0, 1, 2, 3 }
vector long long, vector double	{ 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7 }

Table 4.130. Supported type signatures for vec_reve

r	a	Example Implementation
vector bool char	vector bool char	`vperm[r] r,a,a,pcv`
vector signed char	vector signed char	`vperm[r] r,a,a,pcv`
vector unsigned char	vector unsigned char	`vperm[r] r,a,a,pcv`
vector bool short	vector bool short	`vperm[r] r,a,a,pcv`
vector signed short	vector signed short	`vperm[r] r,a,a,pcv`
vector unsigned short	vector unsigned short	`vperm[r] r,a,a,pcv`
vector bool int	vector bool int	`vperm[r] r,a,a,pcv`
vector signed int	vector signed int	`vperm[r] r,a,a,pcv`
vector unsigned int	vector unsigned int	`vperm[r] r,a,a,pcv`

r	a	Example Implementation
vector bool long long	vector bool long long	`vperm[r] r,a,a,pcv`
vector signed long long	vector signed long long	`vperm[r] r,a,a,pcv`
vector unsigned long long	vector unsigned long long	`vperm[r] r,a,a,pcv`
vector float	vector float	`vperm[r] r,a,a,pcv`
vector double	vector double	`vperm[r] r,a,a,pcv`

vec_rint

Vector Round to Nearest Integer

```
r = vec_rint (a)
```

Purpose: Returns a vector containing the floating-point integral values nearest to the values of the corresponding elements of the source vector.

Result value: Each element of **r** contains the nearest representable floating-point integral value to the value of the corresponding element of **a**. When an input element value is exactly between two integer values, the result value is selected based on the rounding mode specified by the Floating-Point Rounding Control field (RN) of the FPSCR register.

Endian considerations: None.

Table 4.131. Supported type signatures for vec_rint

r	a	Example Implementation
vector float	vector float	`xvrspic r,a`
vector double	vector double	`xvrdpic r,a`

vec_rl

Vector Rotate Left

```
r = vec_rl (a, b)
```

Purpose: Rotates each element of a vector left by a given number of bits.

Result value: Each element of **r** is obtained by rotating the corresponding element of **a** left by the number of bits specified by the corresponding element of **b**.

Endian considerations: None.

Table 4.132. Supported type signatures for vec_rl

r	a	b	Example Implementation
vector signed char	vector signed char	vector unsigned char	`vrlb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vrlb r,a,b`
vector signed short	vector signed short	vector unsigned short	`vrlh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vrlh r,a,b`
vector signed int	vector signed int	vector unsigned int	`vrlw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vrlw r,a,b`
vector signed long long	vector signed long long	vector unsigned long long	`vrld r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vrld r,a,b`

vec_rlmi

Vector Rotate Left then Mask Insert

```
r = vec_rlmi (a, b, c)
```

Purpose: Rotates each element of a vector left and inserts each element under a mask.

Result value: Each element of **r** is obtained by rotating the corresponding element of vector **b** left and inserting it under mask into the corresponding element of **a**. Bits 11:15 of the corresponding element of **c** contain the mask beginning, bits 19:23 contain the mask end, and bits 27:31 contain the shift count.

Endian considerations: The referenced bit numbers within the elements of **c** are in left-to-right order.

Table 4.133. Supported type signatures for vec_rlmi

r	a	b	c	Example Implementation	Restrictions
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	`vrlwmi r/a,b,c`	ISA 3.0 or later
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned long long	`vrldmi r/a,b,c`	ISA 3.0 or later

vec_rlnm

Vector Rotate Left then AND with Mask

```
r = vec_rlnm (a, b, c)
```

Purpose: Rotates each element of a vector left, then logically ANDs it with a mask.

Result value: Each element of **a** is rotated left, then logically ANDed with a mask specified by **b** and **c**.

b contains the shift count for each element in the low-order byte, with other bytes zero. **c** contains the mask begin and mask end for each element, with the mask end in the low-order byte, the mask begin in the next higher byte, and other bytes zero.

Endian considerations: None.

Table 4.134. Supported type signatures for vec_rlnm

r	a	b	c	Example Implementation	Restrictions

vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	`vspltisw t,8`
				`vslw u,b,t`
				`xxlor v,u,c`
				`vrlwnm r,a,v`
					ISA 3.0 or later

vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxspltib t,8`
				`vextsb2d u,t`
				`vsld v,b,u`
				`xxlor w,v,c`
				`vrldnm r,a,w`
					ISA 3.0 or later

vec_round

Vector Round

```
r = vec_round (a)
```

Purpose: Returns a vector containing the rounded values of the corresponding elements of the source vector.

Result value: Each element of **r** contains the value of the corresponding element of **a**, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding. The current floating-point rounding mode is ignored.

Notes: 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 to the XLC compiler.

Endian considerations: None.

Table 4.135. Supported type signatures for vec_round

r	a	Example Implementation
vector float	vector float	`vrfin r,a`
vector double	vector double	`xvrdpi r,a`

vec_rsqrt

Vector Reciprocal Square Root

```
r = vec_rsqrt (a)
```

Purpose: Returns a vector containing a refined approximation of the reciprocal square roots of the corresponding elements of the source vector. This function provides an implementation-dependent greater precision than **vec_rsqrte**.

Result value: Each element of **r** contains a refined approximation of the reciprocal square root of the corresponding element of **a**.

Endian considerations: None.

Notes:

- The example implementations assume that a register **h** initially contains the floating-point value 0.5 in each element (single- or double-precision as appropriate).
- For finite square roots, this intrinsic guarantees at least 23 bits of accuracy for single-precision floating point, and at least 52 bits of accuracy for double-precision floating point.

Table 4.136. Supported type signatures for vec_rsqrt

r	a	Example Implementation

vector float	vector float	`xvrsqrtesp t,a`
		`xvmulsp u,t,a`
		`xvmulsp v,t,h`
		`xvnmsubmsp v,u,h`
		`xvmaddmsp r/v,t,t`

vector double	vector double	`xvrsqrtedp t,a`
		`xvmuldp u,t,a`
		`xvmuldp v,t,h`
		`xxlor w,h,h`
		`xvnmsubadp w,u,v`
		`xvmaddadp v,v,w`
		`xvmaddadp u,u,w`
		`xvnmsubmdp u,v,h`
		`xvmaddadp v,v,u`
		`xvadddp r,v,v`

vec_rsqrte

Vector Reciprocal Square Root Estimate

```
r = vec_rsqrte (a)
```

Purpose: Returns a vector containing estimates of the reciprocal square roots of the corresponding elements of the source vector.

Result value: Each element of **r** contains the estimated value of the reciprocal square root of the corresponding element of **a**.

Endian considerations: None.

Notes: For finite square roots, this intrinsic guarantees at least 14 bits of accuracy.

Table 4.137. Supported type signatures for vec_rsqrte

r	a	Example Implementation
vector float	vector float	`xvrsqrtesp r,a`
vector double	vector double	`xvrsqrtedp r,a`

vec_sbox_be

Vector AES SubBytes Big-Endian

```
r = vec_sbox_be (a)
```

Purpose: Performs the SubBytes operation, as defined in Federal Information Processing Standards FIPS-197, on a state_array contained in **a**.

Result value: r contains the result of the SubBytes operation, as defined in Federal Information Processing Standard FIPS-197, on the state array represented by **a**.

Endian considerations: All element numberings of the SubBytes operation use big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_sbox_be` does not follow the bi-endian programming model.

Table 4.138. Supported type signatures for vec_sbox_be

r	a	Example Implementation
vector unsigned char	vector unsigned char	`vsbox r,a`

vec_sel

Vector Select

```
r = vec_sel (a, b, c)
```

Purpose: Returns a vector selecting bits from two source vectors depending on the corresponding bit values of a third source vector.

Result value: Each bit of **r** has the value of the corresponding bit of **a** if the corresponding bit of **c** is 0. Otherwise, the bit of **r** has the value of the corresponding bit of **b**.

Endian considerations: None.

Table 4.139. Supported type signatures for vec_sel

r	a	b	c	Example Implementation
vector bool char	vector bool char	vector bool char	vector bool char	`xxsel r,a,b,c`
vector bool char	vector bool char	vector bool char	vector unsigned char	`xxsel r,a,b,c`
vector signed char	vector signed char	vector signed char	vector bool char	`xxsel r,a,b,c`
vector signed char	vector signed char	vector signed char	vector unsigned char	`xxsel r,a,b,c`
vector unsigned char	vector unsigned char	vector unsigned char	vector bool char	`xxsel r,a,b,c`
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	`xxsel r,a,b,c`
vector bool short	vector bool short	vector bool short	vector bool short	`xxsel r,a,b,c`
vector bool short	vector bool short	vector bool short	vector unsigned short	`xxsel r,a,b,c`
vector signed short	vector signed short	vector signed short	vector bool short	`xxsel r,a,b,c`
vector signed short	vector signed short	vector signed short	vector unsigned short	`xxsel r,a,b,c`
vector unsigned short	vector unsigned short	vector unsigned short	vector bool short	`xxsel r,a,b,c`
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	`xxsel r,a,b,c`
vector bool int	vector bool int	vector bool int	vector bool int	`xxsel r,a,b,c`
vector bool int	vector bool int	vector bool int	vector unsigned int	`xxsel r,a,b,c`

r	a	b	c	Example Implementation
vector signed int	vector signed int	vector signed int	vector bool int	`xxsel r,a,b,c`
vector signed int	vector signed int	vector signed int	vector unsigned int	`xxsel r,a,b,c`
vector unsigned int	vector unsigned int	vector unsigned int	vector bool int	`xxsel r,a,b,c`
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	`xxsel r,a,b,c`
vector bool long long	vector bool long long	vector bool long long	vector bool long long	`xxsel r,a,b,c`
vector bool long long	vector bool long long	vector bool long long	vector unsigned long long	`xxsel r,a,b,c`
vector signed long long	vector signed long long	vector signed long long	vector bool long long	`xxsel r,a,b,c`
vector signed long long	vector signed long long	vector signed long long	vector unsigned long long	`xxsel r,a,b,c`
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector bool long long	`xxsel r,a,b,c`
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxsel r,a,b,c`
vector float	vector float	vector float	vector bool int	`xxsel r,a,b,c`
vector float	vector float	vector float	vector unsigned int	`xxsel r,a,b,c`
vector double	vector double	vector double	vector bool long long	`xxsel r,a,b,c`
vector double	vector double	vector double	vector unsigned long long	`xxsel r,a,b,c`

vec_shasigma_be

Vector SHA Sigma Big-Endian

```
r = vec_shasigma_be (a, b, c)
```

Purpose: Performs a Secure Hash computation in accordance with Federal Information Processing Standards FIPS-180-3.

Result value: Each element of **r** contains the SHA256 or SHA512 hash as follows.

The result of the SHA-256 function (**r**[*i*] for *i* = 0 to 3) is:

- σ0(**a**[*i*]), if **b** is 0 and bit *i* of the 4-bit **c** is 0.
- σ1(**a**[*i*]), if **b** is 0 and bit *i* of the 4-bit **c** is 1.
- Σ0(**a**[*i*]), if **b** is nonzero and bit *i* of the 4-bit **c** is 0.
- Σ1(**a**[*i*]), if **b** is nonzero and bit *i* of the 4-bit **c** is 1.

The result of the SHA-512 function (**r**[*i*] for *i* = 0 to 1) is:

- σ0(**a**[*i*]), if **b** is 0 and bit 2 × *i* of the 4-bit **c** is 0.
- σ1(**a**[*i*]), if **b** is 0 and bit 2 × *i* of the 4-bit **c** is 1.
- Σ0(**a**[*i*]), if **b** is nonzero and bit 2 × *i* of the 4-bit **c** is 0.
- Σ1(**a**[*i*]), if **b** is nonzero and bit 2 × *i* of the 4-bit **c** is 1.

Endian considerations: All element numberings in the above description denote big-endian (i.e., left-to-right) order, reflecting the underlying hardware instruction. Unlike most of the vector intrinsics in this chapter, `vec_pmsum_be` does not follow the bi-endian programming model.

Table 4.140. Supported type signatures for vec_shasigma_be

r	a	b	c	Example Implementation
vector unsigned int	vector unsigned int	const int	4-bit unsigned literal	`vshasigmaw r,a,b,c`
vector unsigned long long	vector unsigned long long	const int	4-bit unsigned literal	`vshasigmaw r,a,b,d`

vec_signed

Vector Convert Floating-Point to Signed Integer

```
r = vec_signed (a)
```

Purpose: Converts a vector of floating-point numbers to a vector of signed integers.

Result value: Each element of **r** is obtained by truncating the corresponding element of **a** to a signed integer. The current floating-point rounding mode is ignored.

Endian considerations: None.

Table 4.141. Supported type signatures for vec_signed

r	a	Example Implementation
vector signed int	vector float	`xvcvspsxws r,a`
vector signed long long	vector double	`xvcvdpsxds r,a`

vec_signed2

Vector Convert Double-Precision to Signed Word

```
r = vec_signed2 (a, b)
```

Purpose: Converts two vectors of double-precision floating-point numbers to a vector of signed 32-bit integers.

Result value: Let **v** be the concatenation of **a** and **b**. Each element of **r** is obtained by truncating the corresponding element of **v** to a signed 32-bit integer.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.142. Supported type signatures for vec_signed2

r	a	b	Example LE Implementation	Example BE Implementation

vector signed int	vector double	vector double	`xxpermdi t,b,a,3`
			`xxpermdi u,b,a,0`
			`xvcvdpsxws v,t`
			`xvcvdpsxws w,u`
			`vmrgow r,w,v`
				`xxpermdi t,a,b,0`
				`xxpermdi u,a,b,3`
				`xvcvdpsxws v,t`
				`xvcvdpsxws w,u`
				`vmrgew r,v,w`

vec_signede

Vector Convert Double-Precision to Signed Word Even

```
r = vec_signede (a)
```

Purpose: Converts elements of a source vector to signed integers and stores them in the even-numbered elements of the result vector.

Result value: Element 0 of **r** contains element 0 of **a**, truncated to a signed integer. Element 2 of **r** contains element 1 of **a**, truncated to a signed integer. Elements 1 and 3 of **r** are undefined.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.143. Supported type signatures for vec_signede

r	a	Example LE Implementation	Example BE Implementation
vector signed int	vector double	`xvcvdpsxws t,a` `vsldoi r,t,t,12`	`xvcvdpsxws t,a`

vec_signedo

Vector Convert Double-Precision to Signed Word Odd

```
r = vec_signedo (a)
```

Purpose: Converts elements of a source vector to signed integers and stores them in the odd-numbered elements of the result vector.

Result value: Element 1 of **r** contains element 0 of **a**, truncated to a signed integer. Element 3 of **r** contains element 1 of **a**, truncated to a signed integer. Elements 0 and 2 of **r** are undefined.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.144. Supported type signatures for vec_signedo

r	a	Example LE Implementation	Example BE Implementation
vector signed int	vector double	`xvcvdpsxws r,a`	`xvcvdpsxws t,a` `vsldoi r,t,t,12`

vec_sl

Vector Shift Left

```
r = vec_sl (a, b)
```

Purpose: Performs a left shift for each element of a vector.

Result value: Each element of **r** is the result of left-shifting the corresponding element of **a** by the number of bits specified by the corresponding element of **b**, modulo the number of bits in the element. Zeros are shifted in from the right.

Endian considerations: None.

Table 4.145. Supported type signatures for vec_sl

r	a	b	Example Implementation
vector signed char	vector signed char	vector unsigned char	`vslb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vslb r,a,b`
vector signed short	vector signed short	vector unsigned short	`vslh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vslh r,a,b`
vector signed int	vector signed int	vector unsigned int	`vslw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vslw r,a,b`
vector signed long long	vector signed long long	vector unsigned long long	`vsld r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vsld r,a,b`

vec_sld

Vector Shift Left Double

```
r = vec_sld (a, b, c)
```

Purpose: Left shifts a double vector (that is, two concatenated vectors) by a given number of bytes. For vec_sld being performed on the vector bool and floating-point types, the result is undefined when the specified shift count is not a multiple of the element size.

Result value: Vector **r** receives the most-significant 16 bytes obtained by concatenating **a** and **b** and shifting left by the number of bytes specified by **c**, which must be in the range 0–15.

Endian considerations: This intrinsic is *not* endian-neutral, so uses of vec_sld in big-endian code must be rewritten for little-endian targets. Historically, vec_sld could be used to shift by amounts not a multiple of the element size for most types, in which case the purpose of the shift is difficult to determine and difficult to automatically rewrite efficiently for little endian. So the concatenation of **a** and **b** is done in big-endian fashion (left to right), and the shift is always to the left. This will generally produce surprising results for little-endian targets. See also Section 2.7.2, “vec_sld and vec_sro are not bi-endian” [15].

Table 4.146. Supported type signatures for vec_sld

r	a	b	c	Example Implementation
vector bool char	vector bool char	vector bool char	4-bit unsigned literal	`vsldoi r,a,b,c`
vector signed char	vector signed char	vector signed char	4-bit unsigned literal	`vsldoi r,a,b,c`
vector unsigned char	vector unsigned char	vector unsigned char	4-bit unsigned literal	`vsldoi r,a,b,c`
vector bool short	vector bool short	vector bool short	4-bit unsigned literal	`vsldoi r,a,b,c`
vector signed short	vector signed short	vector signed short	4-bit unsigned literal	`vsldoi r,a,b,c`
vector unsigned short	vector unsigned short	vector unsigned short	4-bit unsigned literal	`vsldoi r,a,b,c`
vector pixel	vector pixel	vector pixel	4-bit unsigned literal	`vsldoi r,a,b,c`
vector bool int	vector bool int	vector bool int	4-bit unsigned literal	`vsldoi r,a,b,c`
vector signed int	vector signed int	vector signed int	4-bit unsigned literal	`vsldoi r,a,b,c`
vector unsigned int	vector unsigned int	vector unsigned int	4-bit unsigned literal	`vsldoi r,a,b,c`
vector bool long long	vector bool long long	vector bool long long	4-bit unsigned literal	`vsldoi r,a,b,c`

r	a	b	c	Example Implementation
vector signed long long	vector signed long long	vector signed long long	4-bit unsigned literal	`vsldoi r,a,b,c`
vector unsigned long long	vector unsigned long long	vector unsigned long long	4-bit unsigned literal	`vsldoi r,a,b,c`
vector float	vector float	vector float	4-bit unsigned literal	`vsldoi r,a,b,c`
vector double	vector double	vector double	4-bit unsigned literal	`vsldoi r,a,b,c`

vec_sldw

Vector Shift Left Double by Words

```
r = vec_sldw (a, b, c)
```

Purpose: Returns a vector obtained by shifting left the concatenated source vectors by the number of specified words.

Result value: Vector **r** receives the most-significant 16 bytes obtained by concatenating **a** and **b** and shifting left by the number of words specified by **c**, which must be in the range 0–3.

Endian considerations: This intrinsic is *not* endian-neutral, so uses of vec_sldw in big-endian code must be rewritten for little-endian targets. The concatenation of **a** and **b** is done in big-endian fashion (left to right), and the shift is always to the left. This will generally produce surprising results for little-endian targets.

Table 4.147. Supported type signatures for vec_sldw

r	a	b	c	Example Implementation
vector signed char	vector signed char	vector signed char	2-bit unsigned literal	`xxsldwi r,a,b,c`
vector unsigned char	vector unsigned char	vector unsigned char	2-bit unsigned literal	`xxsldwi r,a,b,c`
vector signed short	vector signed short	vector signed short	2-bit unsigned literal	`xxsldwi r,a,b,c`
vector unsigned short	vector unsigned short	vector unsigned short	2-bit unsigned literal	`xxsldwi r,a,b,c`
vector signed int	vector signed int	vector signed int	2-bit unsigned literal	`xxsldwi r,a,b,c`
vector unsigned int	vector unsigned int	vector unsigned int	2-bit unsigned literal	`xxsldwi r,a,b,c`
vector signed long long	vector signed long long	vector signed long long	2-bit unsigned literal	`xxsldwi r,a,b,c`
vector unsigned long long	vector unsigned long long	vector unsigned long long	2-bit unsigned literal	`xxsldwi r,a,b,c`

vec_sll

Vector Shift Left Long

```
r = vec_sll (a, b)
```

Purpose: Left shifts an entire vector by a given number of bits.

Result value: Vector **r** contains the contents of **a**, shifted left by the number of bits specified by the three least-significant bits of **b**. Zeros are supplied on the right. The shift count must have been replicated into all bytes of **b**; if not, the value of **r** is undefined.

Endian considerations: This intrinsic is *not* endian-neutral, so uses of vec_sll in big-endian code must be rewritten for little-endian targets.

Table 4.148. Supported type signatures for vec_sll

r	a	b	Example Implementation
vector signed char	vector signed char	vector unsigned char	`vsl r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vsl r,a,b`
vector signed short	vector signed short	vector unsigned char	`vsl r,a,b`
vector unsigned short	vector unsigned short	vector unsigned char	`vsl r,a,b`
vector pixel	vector pixel	vector unsigned char	`vsl r,a,b`
vector signed int	vector signed int	vector unsigned char	`vsl r,a,b`
vector unsigned int	vector unsigned int	vector unsigned char	`vsl r,a,b`
vector signed long long	vector signed long long	vector unsigned char	`vsl r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned char	`vsl r,a,b`

vec_slo

Vector Shift Left by Octets

```
r = vec_slo (a, b)
```

Purpose: Left shifts a vector by a given number of bytes (octets).

Result value: Vector **r** receives the contents of **a**, shifted left by the number of bytes specified by bits 1:4 of the least-significant byte of **b**.

Endian considerations: This intrinsic is *not* endian-neutral, so uses of vec_slo in big-endian code must be rewritten for little-endian targets. The shift count is in element 15 of **b** for big-endian, but in element 0 of **b** for little-endian.

Table 4.149. Supported type signatures for vec_slo

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vslo r,a,b`
vector signed char	vector signed char	vector unsigned char	`vslo r,a,b`
vector unsigned char	vector unsigned char	vector signed char	`vslo r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vslo r,a,b`
vector signed short	vector signed short	vector signed char	`vslo r,a,b`
vector signed short	vector signed short	vector unsigned char	`vslo r,a,b`
vector unsigned short	vector unsigned short	vector signed char	`vslo r,a,b`
vector unsigned short	vector unsigned short	vector unsigned char	`vslo r,a,b`
vector pixel	vector pixel	vector signed char	`vslo r,a,b`
vector pixel	vector pixel	vector unsigned char	`vslo r,a,b`
vector signed int	vector signed int	vector signed char	`vslo r,a,b`
vector signed int	vector signed int	vector unsigned char	`vslo r,a,b`
vector unsigned int	vector unsigned int	vector signed char	`vslo r,a,b`
vector unsigned int	vector unsigned int	vector unsigned char	`vslo r,a,b`

r	a	b	Example Implementation
vector signed long long	vector signed long long	vector signed char	`vslo r,a,b`
vector signed long long	vector signed long long	vector unsigned char	`vslo r,a,b`
vector unsigned long long	vector unsigned long long	vector signed char	`vslo r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned char	`vslo r,a,b`
vector float	vector float	vector signed char	`vslo r,a,b`
vector float	vector float	vector unsigned char	`vslo r,a,b`

vec_slv

Vector Shift Left Variable

```
r = vec_slv (a, b)
```

Purpose: Left-shifts a vector by a varying number of bits by element.

Result value: Let **v** be a 17-byte vector formed from **a** in bytes [0:15] and a zero byte in element 16. Then each byte element *i* of **r** is determined as follows. The start bit *sb* is obtained from bits 5:7 of byte element *i* of **b**. Then the contents of bits *sb*:*sb+7* of the halfword in byte elements *i*:*i+1* of **v** are placed into byte element *i* of **r**.

An example follows:

byte index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F
b	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
r	0F	1E	3C	78	F0	E1	C3	87	0F	1E	3C	78	F0	E1	C3	80

Endian considerations: All bit and byte element numbers are specified in big-endian order. This intrinsic is *not* endian-neutral.

Table 4.150. Supported type signatures for vec_slv

r	a	b	Example Implementation	Restrictions
vector unsigned char	vector unsigned char	vector unsigned char	`vslv r,a,b`	ISA 3.0 or later

vec_splat

Vector Splat

```
r = vec_splat (a, b)
```

Purpose: Returns a vector that has all of its elements set to a given value.

Result value: The value of each element of **r** is the value of the element of **a** specified by **b**, which must be an element number less than the number of elements supported for **a**'s type.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.151. Supported type signatures for vec_splat

r	a	b	Example LE Implementation	Example BE Implementation
vector bool char	vector bool char	const int	`vspltb r,a,15-b`	`vspltb r,a,b`
vector signed char	vector signed char	const int	`vspltb r,a,15-b`	`vspltb r,a,b`
vector unsigned char	vector unsigned char	const int	`vspltb r,a,15-b`	`vspltb r,a,b`
vector bool short	vector bool short	const int	`vsplth r,a,7-b`	`vsplth r,a,b`
vector signed short	vector signed short	const int	`vsplth r,a,7-b`	`vsplth r,a,b`
vector unsigned short	vector unsigned short	const int	`vsplth r,a,7-b`	`vsplth r,a,b`
vector pixel	vector pixel	const int	`vsplth r,a,7-b`	`vsplth r,a,b`
vector bool int	vector bool int	const int	`xxspltw r,a,3-b`	`xxspltw r,a,b`
vector signed int	vector signed int	const int	`xxspltw r,a,3-b`	`xxspltw r,a,b`
vector unsigned int	vector unsigned int	const int	`xxspltw r,a,3-b`	`xxspltw r,a,b`
vector bool long long	vector bool long long	const int	`xxpermdi r,a,a,(1-b)*3`	`xxpermdi r,a,a,b`
vector signed long long	vector signed long long	const int	`xxpermdi r,a,a,(1-b)*3`	`xxpermdi r,a,a,b`
vector unsigned long long	vector unsigned long long	const int	`xxpermdi r,a,a,(1-b)*3`	`xxpermdi r,a,a,b`
vector float	vector float	const int	`xxspltw r,a,3-b`	`xxspltw r,a,b`

r	a	b	Example LE Implementation	Example BE Implementation
vector double	vector double	const int	`xxpermdi r,a,a,(1-b)*3`	`xxpermdi r,a,a,b`

vec_splat_s8

Vector Splat to Signed Byte

```
r = vec_splat_s8 (a)
```

Purpose: Returns a vector with all elements equal to the given value.

Result value: Each element of **r** is given the sign-extended 5-bit value of **a**. The range of this value is [-16:15].

Endian considerations: None.

Table 4.152. Supported type signatures for vec_splat_s8

r	a	Example Implementation
vector signed char	5-bit signed literal	`vspltisb r,a`

vec_splat_s16

Vector Splat to Signed Halfword

```
r = vec_splat_s16 (a)
```

Purpose: Returns a vector with all elements equal to the given value.

Result value: Each element of **r** is given the sign-extended 5-bit value of **a**. The range of this value is [-16:15].

Endian considerations: None.

Table 4.153. Supported type signatures for vec_splat_s16

r	a	Example Implementation
vector signed short	5-bit signed literal	`vspltish r,a`

vec_splat_s32

Vector Splat to Signed Word

```
r = vec_splat_s32 (a)
```

Purpose: Returns a vector with all elements equal to the given value.

Result value: Each element of **r** is given the sign-extended 5-bit value of **a**. The range of this value is [-16:15].

Endian considerations: None.

Table 4.154. Supported type signatures for vec_splat_s32

r	a	Example Implementation
vector signed int	5-bit signed literal	`vspltisw r,a`

vec_splat_u8

Vector Splat to Unsigned Byte

```
r = vec_splat_u8 (a)
```

Purpose: Returns a vector with all elements equal to the given value.

Result value: The 5-bit signed value of **a** is sign-extended to a byte and the resulting value is cast to an unsigned char. This value is placed in each element of **r**. The range of the original value is [-16:15].

Endian considerations: None.

Table 4.155. Supported type signatures for vec_splat_u8

r	a	Example Implementation
vector unsigned char	5-bit signed literal	`vspltisb r,a`

vec_splat_u16

Vector Splat to Unsigned Halfword

```
r = vec_splat_u16 (a)
```

Purpose: Returns a vector with all elements equal to the given value.

Result value: The 5-bit signed value of **a** is sign-extended to a halfword and the resulting value is cast to an unsigned short. This value is placed in each element of **r**. The range of the original value is [-16:15].

Endian considerations: None.

Table 4.156. Supported type signatures for vec_splat_u16

r	a	Example Implementation
vector unsigned short	5-bit signed literal	`vspltish r,a`

vec_splat_u32

Vector Splat to Unsigned Word

```
r = vec_splat_u32 (a)
```

Purpose: Returns a vector with all elements equal to the given value.

Result value: The 5-bit signed value of **a** is sign-extended to a word and the resulting value is cast to an unsigned int. This value is placed in each element of **r**. The range of the original value is [-16:15].

Endian considerations: None.

Table 4.157. Supported type signatures for vec_splat_u32

r	a	Example Implementation
vector unsigned int	5-bit signed literal	`vspltisw r,a`

vec_splats

Vector Splat Scalar

```
r = vec_splats (a)
```

Purpose: Returns a vector with the value of each element set to the value of the scalar input parameter.

Result value: Each element of **r** is set to the value of **a**.

Endian considerations: None.

Table 4.158. Supported type signatures for vec_splats

r	a	Example Implementation

vector signed char	signed char	`rlwinm t,a,0,0xff`
		`mtvsrd u,t`
		`vspltb r,u,7`

vector unsigned char	unsigned char	`rlwinm t,a,0,0xff`
		`mtvsrd u,t`
		`vspltb r,u,7`

vector signed short	signed short	`rlwinm t,a,0,0xffff`
		`mtvsrd u,t`
		`vsplth r,u,3`

vector unsigned short	unsigned short	`rlwinm t,a,0,0xffff`
		`mtvsrd u,t`
		`vsplth r,u,3`

vector signed int	signed int	`mtvsrd t,a` `vspltb r,t,7`
vector unsigned int	unsigned int	`mtvsrd t,a` `vspltb r,t,7`
vector signed long long	signed long long	`mtvsrd t,a` `xxpermdi r,t,t,0`
vector unsigned long long	unsigned long long	`mtvsrd t,a` `xxpermdi r,t,t,0`
vector signed __int128	signed __int128	`mtvsrwz t,a` `xxspltw r,t,1`
vector unsigned __int128	unsigned __int128	`mtvsrwz t,a` `xxspltw r,t,1`
vector float	float	`xxscvdpspn t,a` `xxspltw r,t,0`

r	a	Example Implementation
vector double	double	`xxpermdi r,a,a,0`

vec_sqrt

Vector Square Root

```
r = vec_sqrt (a)
```

Purpose: Returns a vector containing the square root of each element in the source vector.

Result value: Each element of **r** is the square root of the corresponding element of **a**.

Endian considerations: None.

Table 4.159. Supported type signatures for vec_sqrt

r	a	Example Implementation
vector float	vector float	`xvsqrtsp r,a`
vector double	vector double	`xvsqrtdp r,a`

vec_sr

Vector Shift Right

```
r = vec_sr (a, b)
```

Purpose: Performs a logical right shift for each element of a vector.

Result value: Each element of **r** is the result of logically right-shifting the corresponding element of **a** by the number of bits specified by the corresponding element of **b**, modulo the number of bits in the element. Zeros are shifted in from the left.

Endian considerations: None.

Table 4.160. Supported type signatures for vec_sr

r	a	b	Example Implementation
vector signed char	vector signed char	vector unsigned char	`vsrb r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vsrb r,a,b`
vector signed short	vector signed short	vector unsigned short	`vsrh r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vsrh r,a,b`
vector signed int	vector signed int	vector unsigned int	`vsrw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vsrw r,a,b`
vector signed long long	vector signed long long	vector unsigned long long	`vsrd r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vsrd r,a,b`

vec_sra

Vector Shift Right Algebraic

```
r = vec_sra (a, b)
```

Purpose: Performs an algebraic right shift for each element of a vector.

Result value: Each element of **r** is the result of algebraically right-shifting the corresponding element of **a** by the number of bits specified by the corresponding element of **b**, modulo the number of bits in the element. Copies of the sign bit are shifted in from the left.

Endian considerations: None.

Table 4.161. Supported type signatures for vec_sra

r	a	b	Example Implementation
vector signed char	vector signed char	vector unsigned char	`vsrab r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vsrab r,a,b`
vector signed short	vector signed short	vector unsigned short	`vsrah r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vsrah r,a,b`
vector signed int	vector signed int	vector unsigned int	`vsraw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vsraw r,a,b`
vector signed long long	vector signed long long	vector unsigned long long	`vsrad r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vsrad r,a,b`

vec_srl

Vector Shift Right Long

```
r = vec_srl (a, b)
```

Purpose: Right shifts a vector by a given number of bits.

Result value: Vector **r** contains the contents of **a**, shifted right by the number of bits specified by the 3 least-significant bits of **b**. Zeros are supplied on the left. The shift count must have been replicated into all bytes of **b**; if not, the value of **r** is undefined.

Endian considerations: This intrinsic is *not* endian-neutral, so uses of vec_srl in big-endian code must be rewritten for little-endian targets.

Table 4.162. Supported type signatures for vec_srl

r	a	b	Example Implementation
vector signed char	vector signed char	vector unsigned char	`vsr r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vsr r,a,b`
vector signed short	vector signed short	vector unsigned char	`vsr r,a,b`
vector unsigned short	vector unsigned short	vector unsigned char	`vsr r,a,b`
vector pixel	vector pixel	vector unsigned char	`vsr r,a,b`
vector signed int	vector signed int	vector unsigned char	`vsr r,a,b`
vector unsigned int	vector unsigned int	vector unsigned char	`vsr r,a,b`
vector signed long long	vector signed long long	vector unsigned char	`vsr r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned char	`vsr r,a,b`

vec_sro

Vector Shift Right by Octets

```
r = vec_sro (a, b)
```

Purpose: Right shifts a vector by a given number of bytes (octets).

Result value: Vector **r** receives the contents of **a**, shifted right by the number of bytes specified by bits 1–4 of the least-significant byte of **b**. Zeros are supplied from the left.

Endian considerations: This intrinsic is *not* endian-neutral, so uses of vec_sro in big-endian code must be rewritten for little-endian targets. The shift count is in element 15 of **b** for big-endian, but in element 0 of **b** for little-endian. See also Section 2.7.2, “vec_sld and vec_sro are not bi-endian” [15].

Table 4.163. Supported type signatures for vec_sro

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vsro r,a,b`
vector signed char	vector signed char	vector unsigned char	`vsro r,a,b`
vector unsigned char	vector unsigned char	vector signed char	`vsro r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vsro r,a,b`
vector signed short	vector signed short	vector signed char	`vsro r,a,b`
vector signed short	vector signed short	vector unsigned char	`vsro r,a,b`
vector unsigned short	vector unsigned short	vector signed char	`vsro r,a,b`
vector unsigned short	vector unsigned short	vector unsigned char	`vsro r,a,b`
vector pixel	vector pixel	vector signed char	`vsro r,a,b`
vector pixel	vector pixel	vector unsigned char	`vsro r,a,b`
vector signed int	vector signed int	vector signed char	`vsro r,a,b`
vector signed int	vector signed int	vector unsigned char	`vsro r,a,b`
vector unsigned int	vector unsigned int	vector signed char	`vsro r,a,b`
vector unsigned int	vector unsigned int	vector unsigned char	`vsro r,a,b`

r	a	b	Example Implementation
vector signed long long	vector signed long long	vector signed char	`vsro r,a,b`
vector signed long long	vector signed long long	vector unsigned char	`vsro r,a,b`
vector unsigned long long	vector unsigned long long	vector signed char	`vsro r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned char	`vsro r,a,b`
vector float	vector float	vector signed char	`vsro r,a,b`
vector float	vector float	vector unsigned char	`vsro r,a,b`

vec_srv

Vector Shift Right Variable

```
r = vec_srv (a, b)
```

Purpose: Right-shifts a vector by a varying number of bits by element.

Result value: Let **v** be a 17-byte vector formed from a zero byte in element 0 and the elements of **a** in bytes [1:16]. Then each byte element *i* of **r** is determined as follows. The start bit *sb* is obtained from bits 5:7 of byte element *i* of **b**. Then the contents of bits (8 – *sb*):(15 – *sb*) of the halfword in byte elements *i*:*i*+1 of **v** are placed into byte element *i* of **r**.

An example follows:

byte index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F	0F
b	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
r	00	3C	78	F0	E1	C3	87	0F	1E	3C	78	F0	E1	C3	87	0F

Endian considerations: All bit and byte element numbers are specified in big-endian order. This intrinsic is *not* endian-neutral.

Table 4.164. Supported type signatures for vec_srv

r	a	b	Example Implementation	Restrictions
vector unsigned char	vector unsigned char	vector unsigned char	`vsrv r,a,b`	ISA 3.0 or later

vec_st

Vector Store Indexed

```
vec_st (a, b, c)
```

Purpose: Stores a 16-byte vector into memory at the address specified by a displacement and a pointer, ignoring the four low-order bits of the calculated address.

Operation: A memory address is obtained by adding **b** and **c**, and masking off the four low-order bits of the result. The 16-byte vector in **a** is stored to the resultant memory address.

Endian considerations: None.

Table 4.165. Supported type signatures for vec_st

a	b	c	Example ISA 3.0 Implementation
vector bool char	signed long long	vector bool char *	`stvx r,b,a`
vector bool char	signed long long	signed char *	`stvx r,b,a`
vector bool char	signed long long	unsigned char *	`stvx r,b,a`
vector signed char	signed long long	signed char *	`stvx r,b,a`
vector signed char	signed long long	vector signed char *	`stvx r,b,a`
vector unsigned char	signed long long	unsigned char *	`stvx r,b,a`
vector unsigned char	signed long long	vector unsigned char *	`stvx r,b,a`
vector bool short	signed long long	vector bool short *	`stvx r,b,a`
vector bool short	signed long long	signed short *	`stvx r,b,a`
vector bool short	signed long long	unsigned short *	`stvx r,b,a`
vector signed short	signed long long	signed short *	`stvx r,b,a`
vector signed short	signed long long	vector signed short *	`stvx r,b,a`
vector unsigned short	signed long long	unsigned short *	`stvx r,b,a`
vector unsigned short	signed long long	vector unsigned short *	`stvx r,b,a`

a	b	c	Example ISA 3.0 Implementation
vector pixel	signed long long	vector pixel *	`stvx r,b,a`
vector bool int	signed long long	vector bool int *	`stvx r,b,a`
vector bool int	signed long long	signed int *	`stvx r,b,a`
vector bool int	signed long long	unsigned int *	`stvx r,b,a`
vector signed int	signed long long	signed int *	`stvx r,b,a`
vector signed int	signed long long	vector signed int *	`stvx r,b,a`
vector unsigned int	signed long long	unsigned int *	`stvx r,b,a`
vector unsigned int	signed long long	vector unsigned int *	`stvx r,b,a`
vector bool long long	signed long long	vector bool long long *	`stvx r,b,a`
vector signed long long	signed long long	signed long long *	`stvx r,b,a`
vector signed long long	signed long long	vector signed long long *	`stvx r,b,a`
vector unsigned long long	signed long long	unsigned long long *	`stvx r,b,a`
vector unsigned long long	signed long long	vector unsigned long long *	`stvx r,b,a`
vector float	signed long long	float *	`stvx r,b,a`
vector float	signed long long	vector float *	`stvx r,b,a`
vector double	signed long long	double *	`stvx r,b,a`
vector double	signed long long	vector double *	`stvx r,b,a`

vec_ste

Vector Store Element Indexed

```
vec_ste (a, b, c)
```

Purpose: Stores a single element from a 16-byte vector into memory at the address specified by a displacement and a pointer, aligned to the element size.

Operation: The integer value **b** is added to the pointer value **c**. The resulting address is rounded down to the nearest address that is a multiple of *es*, where *es* is 1 for char pointers, 2 for short pointers, and 4 for float or int pointers. An element offset *eo* is calculated by taking the resultant address modulo 16. The vector element of **a** at offset *eo* is stored to the resultant address.

Endian considerations: None.

Notes: Be careful to note that the address (b+c) is aligned to an element boundary. Do not attempt to store unaligned data with this intrinsic.

Table 4.166. Supported type signatures for vec_ste

a	b	c	Example ISA 3.0 Implementation
vector bool char	signed long long	signed char *	`stvebx r,b,a`
vector bool char	signed long long	unsigned char *	`stvebx r,b,a`
vector signed char	signed long long	signed char *	`stvebx r,b,a`
vector unsigned char	signed long long	unsigned char *	`stvebx r,b,a`
vector bool short	signed long long	signed short *	`stvehx r,b,a`
vector bool short	signed long long	unsigned short *	`stvehx r,b,a`
vector signed short	signed long long	signed short *	`stvehx r,b,a`
vector unsigned short	signed long long	unsigned short *	`stvehx r,b,a`
vector pixel	signed long long	unsigned short *	`stvehx r,b,a`
vector bool int	signed long long	signed int *	`stvewx r,b,a`
vector bool int	signed long long	unsigned int *	`stvewx r,b,a`
vector signed int	signed long long	signed int *	`stvewx r,b,a`

a	b	c	Example ISA 3.0 Implementation
vector unsigned int	signed long long	unsigned int *	`stvewx r,b,a`
vector float	signed long long	float *	`stvewx r,b,a`

vec_stl

Vector Store Indexed Least Recently Used

```
vec_stl (a, b, c)
```

Purpose: Stores a 16-byte vector into memory at the address specified by a displacement and a pointer, ignoring the four low-order bits of the calculated address, and marking the cache line containing the address as least frequently used.

Operation: A memory address is obtained by adding **b** and **c**, and masking off the four low-order bits of the result. The 16-byte vector in **a** is stored to the resultant memory address, and the containing cache line is marked as least frequently used.

Endian considerations: None.

Notes: This intrinsic can be used to indicate the last access to a portion of memory, as a hint to the data cache controller that the associated cache line can be replaced without performance loss.

Table 4.167. Supported type signatures for vec_stl

a	b	c	Example ISA 3.0 Implementation
vector bool char	signed long long	vector bool char *	`stvxl r,b,a`
vector bool char	signed long long	signed char *	`stvxl r,b,a`
vector bool char	signed long long	unsigned char *	`stvxl r,b,a`
vector signed char	signed long long	signed char *	`stvxl r,b,a`
vector signed char	signed long long	vector signed char *	`stvxl r,b,a`
vector unsigned char	signed long long	unsigned char *	`stvxl r,b,a`
vector unsigned char	signed long long	vector unsigned char *	`stvxl r,b,a`
vector bool short	signed long long	vector bool short *	`stvxl r,b,a`
vector bool short	signed long long	signed short *	`stvxl r,b,a`
vector bool short	signed long long	unsigned short *	`stvxl r,b,a`
vector signed short	signed long long	signed short *	`stvxl r,b,a`
vector signed short	signed long long	vector signed short *	`stvxl r,b,a`

a	b	c	Example ISA 3.0 Implementation
vector unsigned short	signed long long	unsigned short *	`stvxl r,b,a`
vector unsigned short	signed long long	vector unsigned short *	`stvxl r,b,a`
vector pixel	signed long long	vector pixel *	`stvxl r,b,a`
vector bool int	signed long long	vector bool int *	`stvxl r,b,a`
vector bool int	signed long long	signed int *	`stvxl r,b,a`
vector bool int	signed long long	unsigned int *	`stvxl r,b,a`
vector signed int	signed long long	signed int *	`stvxl r,b,a`
vector signed int	signed long long	vector signed int *	`stvxl r,b,a`
vector unsigned int	signed long long	unsigned int *	`stvxl r,b,a`
vector unsigned int	signed long long	vector unsigned int *	`stvxl r,b,a`
vector bool long long	signed long long	vector bool long long *	`stvxl r,b,a`
vector signed long long	signed long long	signed long long *	`stvxl r,b,a`
vector signed long long	signed long long	vector signed long long *	`stvxl r,b,a`
vector unsigned long long	signed long long	unsigned long long *	`stvxl r,b,a`
vector unsigned long long	signed long long	vector unsigned long long *	`stvxl r,b,a`
vector float	signed long long	float *	`stvxl r,b,a`
vector float	signed long long	vector float *	`stvxl r,b,a`
vector double	signed long long	double *	`stvxl r,b,a`
vector double	signed long long	vector double *	`stvxl r,b,a`

vec_sub

Vector Subtract

```
r = vec_sub (a, b)
```

Purpose: Returns a vector containing the result of subtracting each element of one source vector from the corresponding element of another source vector.

Result value: The value of each element of **r** is the result of subtracting the value of the corresponding element of **b** from the value of the corresponding element of **a**. The arithmetic is modular for integer vectors.

Endian considerations: None.

Table 4.168. Supported type signatures for vec_sub

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vsububm r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vsububm r,a,b`
vector signed short	vector signed short	vector signed short	`vsubuhm r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vsubuhm r,a,b`
vector signed int	vector signed int	vector signed int	`vsubuwm r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vsubuwm r,a,b`
vector signed long long	vector signed long long	vector signed long long	`vsubudm r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`vsubudm r,a,b`
vector signed __int128	vector signed __int128	vector signed __int128	`vsubuqm r,a,b`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vsubuqm r,a,b`
vector float	vector float	vector float	`xvsubsp r,a,b`
vector double	vector double	vector double	`xvsubdp r,a,b`

vec_subc

Vector Subtract Carryout

```
r = vec_subc (a, b)
```

Purpose: Returns a vector wherein each element contains the carry produced by subtracting the corresponding elements of the two source vectors.

Result value: The value of each element of **r** is the complement of the carry produced by subtracting the value of the corresponding element of **b** from the value of the corresponding element of **a**. The value is 0 if a borrow occurred, or 1 if no borrow occurred.

Endian considerations: None.

Table 4.169. Supported type signatures for vec_subc

r	a	b	Example Implementation
vector signed int	vector signed int	vector signed int	`vsubcuw r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vsubcuw r,a,b`
vector signed __int128	vector signed __int128	vector signed __int128	`vsubcuq r,a,b`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vsubcuq r,a,b`

vec_sube

Vector Subtract Extended

```
r = vec_sube (a, b, c)
```

Purpose: Returns a vector containing the result of first elementwise subtracting one vector from another vector, and then elementwise adding a third carry vector. Elements of the carry vector have a value of 0 or 1.

Result value: Let **c'** be a vector for which each element is 0 if the rightmost bit of the corresponding element of **c** is 0, and 1 otherwise. Then the value of each element of **r** is produced by subtracting the corresponding element of **b** from the corresponding element of **a**, and then adding the corresponding element of **c'**.

Endian considerations: None.

Notes: Code generated for this intrinsic should ensure only the low-order bit of **c** participates in the sum.

Table 4.170. Supported type signatures for vec_sube

r	a	b	c	Example Implementation

vector signed int	vector signed int	vector signed int	vector signed int	`vspltisw t,1`
				`vsubuwm u,a,b`
				`xxland v,c,t`
				`vsubuwm r,u,v`

vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	`vspltisw t,1`
				`vsubuwm u,a,b`
				`xxland v,c,t`
				`vsubuwm r,u,v`

vector signed __int128	vector signed __int128	vector signed __int128	vector signed __int128	`vsubeuqm r,a,b,c`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vsubeuqm r,a,b,c`

vec_subec

Vector Subtract Extended Carryout

```
r = vec_subec (a, b, c)
```

Purpose: Returns a vector containing the carries produced by subtracting one vector from another, then adding a third vector to the difference. The third vector is a carry vector, with each element having a value of 0 or 1.

Result value: The value of each element of **r** is the carry produced by subtracting the corresponding element of **b** from the corresponding element of **a**, and then adding the carry specified in the corresponding element of **c** (1 if there is a carry, 0 otherwise).

Endian considerations: None.

Notes: Code generated for this intrinsic should ensure only the low-order bit of **c** participates in the sum.

Table 4.171. Supported type signatures for vec_subec

r	a	b	c	Example Implementation

vector signed int	vector signed int	vector signed int	vector signed int	`vspltisw t,1`
				`xxland u,c,t`
				`vsubuwm v,a,b`
				`vsubcuw w,a,b`
				`vsubcuw x,v,u`
				`xxlor r,w,x`

vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	`vspltisw t,1`
				`xxland u,c,t`
				`vsubuwm v,a,b`
				`vsubcuw w,a,b`
				`vsubcuw x,v,u`
				`xxlor r,w,x`

vector signed __int128	vector signed __int128	vector signed __int128	vector signed __int128	`vsubecuq r,a,b,c`
vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	vector unsigned __int128	`vsubecuq r,a,b,c`

vec_subs

Vector Subtract Saturated

```
r = vec_subs (a, b)
```

Purpose: Returns a vector containing the saturated differences of each set of corresponding elements of the source vectors.

Result value: The value of each element of **r** is the saturated result of subtracting the value of the corresponding element of **b** from the value of the corresponding element of **a**.

Endian considerations: None.

Table 4.172. Supported type signatures for vec_subs

r	a	b	Example Implementation
vector signed char	vector signed char	vector signed char	`vsubsbs r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`vsububs r,a,b`
vector signed short	vector signed short	vector signed short	`vsubshs r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`vsubuhs r,a,b`
vector signed int	vector signed int	vector signed int	`vsubsws r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`vsubuws r,a,b`

vec_sum2s

Vector Sum Across Half

```
r = vec_sum2s (a, b)
```

Purpose: Returns a vector containing the results of performing a sum-across operation within each doubleword of the first source vector together with accumulated results in the second source vector.

Result value: Elements 0 and 2 of **r** are 0. Element 1 of **r** contains the saturated sum of elements 0 and 1 of **a** and element 1 of **b**. Element 3 of **r** contains the saturated sum of elements 2 and 3 of **a** and element 3 of **b**.

An example follows:

word index	***0***	***1***	***2***	***3***
a	-2 (FFFFFFFE)	-3 (FFFFFFFD)	7 (00000007)	15 (0000000F)
b	31 (0000001F)	-61 (FFFFFFC3)	121 (000000F0)	2147483647 (7FFFFFFF) *(MAXINT)*
calculation	0	-2 + -3 + -61	0	7 + 15 + 2147483647
r	00000000	-66 (FFFFFFBE)	00000000	2147483647 (7FFFFFFF) *(saturated)*

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.173. Supported type signatures for vec_sum2s

r	**a**	**b**	**Example LE Implementation**	**Example BE Implementation**

vector signed int	vector signed int	vector signed int	`vsldoi t,b,b,12`
			`vsum2sws u,a,t`
			`vsldoi r,u,u,4`
				`vsum2sws r,a,b`

vec_sum4s

Vector Sum Across Quarter

```
r = vec_sum4s (a, b)
```

Purpose: Returns a vector containing the results of performing a sum-across operation within each word of the first source vector together with accumulated results in the second source vector.

Result value: There are two cases:

- **a** is a vector of signed or unsigned char. For each element *n* of the result vector, the value is obtained by adding elements 4*n* through 4*n* + 3 of **a** and element *n* of **b** using saturated addition.
- **a** is a vector of signed short. For each element *n* of the result vector, the value is obtained by adding elements 2*n* and 2*n* + 1 of **a** and element *n* of **b** using saturated addition.

An example for input **a** of type vector unsigned char follows:

word index	***0***				***1***				***2***				***3***			
byte index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
a	01	03	07	0F	FF	EE	BB	66	11	22	44	88	00	00	00	01
sum across	0000001A				0000030E				000000FF				00000001			
b	0000FF00				10203040				0000FFFF				FFFFFFFF			
r	0000FF1A				1020334E				000100FE				FFFFFFFF *(saturated)*			

An example for input **a** of type vector signed short follows:

word index	***0***		***1***		***2***		***3***	
halfword index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***
a	FFFF	FFFE	7FFF	7FFE	0124	4210	FFFE	0001
sum across	FFFFFFFD		0000FFFD		00004334		FFFFFFFF	
b	00000003		12340000		7FFFFF00		FFFFFFFF	
r	00000000		1234FFFD		7FFFFFFF *(saturated)*		FFFFFFFE	

Endian considerations: None.

Table 4.174. Supported type signatures for vec_sum4s

r	a	b	Example Implementation
vector signed int	vector signed char	vector signed int	`vsum4sbs r,a,b`
vector unsigned int	vector unsigned char	vector unsigned int	`vsum4ubs r,a,b`
vector signed int	vector signed short	vector signed int	`vsum4shs r,a,b`

vec_sums

Vector Sum Across

```
r = vec_sums (a, b)
```

Purpose: Returns a vector containing the results of performing a sum-across operation on the first source vector together with accumulated results in the second source vector.

Result value: Elements 0, 1, and 2 of **r** are 0. Element 3 is the saturated sum of all the elements of **a** and element 3 of **b**.

An example follows:

word index	*0*	*1*	*2*	*3*
a	00000001	00000003	00000007	0000000F
sum across	00000000	00000000	00000000	0000001A
b	???????? *(ignored)*	???????? *(ignored)*	???????? *(ignored)*	87654321
r	00000000	00000000	00000000	8765433B

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.175. Supported type signatures for vec_sums

r	a	b	Example LE Implementation	Example BE Implementation

vector signed int	vector signed int	vector signed int	`vspltw t,b,0`
			`vsumsws u,a,t`
			`vsldoi r,u,u,12`
				`vsumsws r,a,b`

vec_test_data_class

Vector Test Data Class

```
    r = vec_test_data_class (a, b)
```

Purpose: Determines the data class for each floating-point element.

Result value: Each element of **r** is set to all ones if the corresponding element of **a** matches one of the possible data types selected by **b**. If not, the element is set to all zeros. **b** can select one of the following data classes, or more than one of them by ORing the constants together.

```
Not a number (NaN)       64
Positive infinity        32
Negative infinity        16
Positive zero             8
Negative zero             4
Positive subnormal        2
Negative subnormal        1
```

For clarity of code, the following named constants are suggested. Preferably, compilers will provide these constants in a header file, but this is not required for compliance.

```
#define __VEC_CLASS_FP_NAN          (1<<6)
#define __VEC_CLASS_FP_INFINITY_P   (1<<5)
#define __VEC_CLASS_FP_INFINITY_N   (1<<4)
#define __VEC_CLASS_FP_ZERO_P       (1<<3)
#define __VEC_CLASS_FP_ZERO_N       (1<<2)
#define __VEC_CLASS_FP_SUBNORMAL_P  (1<<1)
#define __VEC_CLASS_FP_SUBNORMAL_N  (1<<0)

#define __VEC_CLASS_FP_INFINITY   (__VEC_CLASS_FP_INFINITY_P
                                   | __VEC_CLASS_FP_INFINITY_N)
#define __VEC_CLASS_FP_ZERO       (__VEC_CLASS_FP_ZERO_P | __VEC_CLASS_FP_ZERO_N)
#define __VEC_CLASS_FP_SUBNORMAL  (__VEC_CLASS_FP_SUBNORMAL_P
                                   | __VEC_CLASS_FP_SUBNORMAL_N)
#define __VEC_CLASS_FP_NOT_NORMAL (__VEC_CLASS_FP_NAN | __VEC_CLASS_FP_SUBNORMAL
                                   | __VEC_CLASS_FP_ZERO | __VEC_CLASS_FP_INFINITY)
```

Endian considerations: None.

Table 4.176. Supported type signatures for vec_test_data_class

r	a	b	Example Implementation	Restrictions
vector bool int	vector float	7-bit unsigned literal	`xvtstdcsp r,a,b`	ISA 3.0 or later
vector bool long long	vector double	7-bit unsigned literal	`xvtstdcdp r,a,b`	ISA 3.0 or later

vec_trunc

Vector Truncate

```
r = vec_trunc (a)
```

Purpose: Returns a vector containing the truncated values of the corresponding elements of the source vector.

Result value: Each element of **r** contains the value of the corresponding element of **a**, truncated to an integral value.

Endian considerations: None.

Table 4.177. Supported type signatures for vec_trunc

r	a	Example Implementation
vector float	vector float	`xvrspiz r,a`
vector double	vector double	`xvrdpiz r,a`

vec_unpackh

Vector Unpack High

```
r = vec_unpackh (a)
```

Purpose: Unpacks the most-significant (“high”) half of a vector into a vector with larger elements.

Result value: If **a** is an integer vector, the value of each element of **r** is the value of the corresponding element of the most-significant half of **a**.

An example for input **a** of type vector signed int follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
r	10111213	24252627	????????	????????
a	0000000010111213		0000000024252627	

If **a** is a floating-point vector, the value of each element of **r** is the value of the corresponding element of the most-significant half of **a**, widened to the result precision.

An example for input **a** of type vector float follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
r	-2.71828182	3.14159265	????????	????????
a	-2.71828182		3.14159265	

If **a** is a pixel vector, the value of each element of **r** is taken from the corresponding element of the most-significant half of **a** as follows:

- All bits in the first byte of the element of **r** are set to the value of the first bit of the element of **a**.
- The least-significant 5 bits of the second byte of the element of **r** are set to the value of the next 5 bits in the element of **a**.
- The least-significant 5 bits of the third byte of the element of **r** are set to the value of the next 5 bits in the element of **a**.
- The least-significant 5 bits of the fourth byte of the element of **r** are set to the value of the next 5 bits in the element of **a**.

An example follows:

word index	***0***		***1***		***2***		***3***	
halfword index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***
a	1234	2567	489A	8BCD	????	????	????	????
unpack halfwords to words	1234		2567		489A		8BCD	
as bits	0001001000110100		0010010101100111		0100100010011010		1000101111001101	
as *1-5-5-5*	0 00100 10001 10100		0 01001 01011 00111		0 10010 00100 11010		1 00010 11110 01101	
r	00041114		00090B07		0012041A		FF021E0D	

Endian considerations: The "high" half of a vector with *n* elements is the first *n*/2 elements of the vector. For little endian, these elements are in the rightmost half of the vector. For big endian, these elements are in the leftmost half of the vector.

Table 4.178. Supported type signatures for vec_unpackh

r	a	Example LE Implementation	Example BE Implementation
vector bool short	vector bool char	`vupklsb r,a`	`vupkhsb r,a`
vector signed short	vector signed char	`vupklsb r,a`	`vupkhsb r,a`
vector bool int	vector bool short	`vupklsh r,a`	`vupkhsh r,a`
vector signed int	vector signed short	`vupklsh r,a`	`vupkhsh r,a`
vector unsigned int	vector pixel	`vupklpx r,a`	`vupkhpx r,a`
vector bool long long	vector bool int	`vupklsw r,a`	`vupkhsw r,a`
vector signed long long	vector signed int	`vupklsw r,a`	`vupkhsw r,a`

vector double	vector float	`xxsldwi t,a,a,3`
		`xxsldwi u,a,t,2`
		`xvcvspdp r,u`
			`xxsldwi t,a,a,1`
			`xxsldwi u,t,a,3`
			`xvcvspdp r,u`

vec_unpackl

Vector Unpack Low

```
r = vec_unpackl (a)
```

Purpose: Unpacks the least-significant (“low”) half of a vector into a vector with larger elements.

Result value: If **a** is an integer vector, the value of each element of **r** is the value of the corresponding element of the least-significant half of **a**.

An example for input **a** of type vector signed int follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
r	????????	????????	38393A3B	4C4D4E4F
a	0000000038393A3B		000000004C4D4E4F	

If **a** is a floating-point vector, the value of each element of **r** is the value of the corresponding element of the least-significant half of **a**, widened to the result precision.

An example for input **a** of type vector float follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
r	????????	????????	6.0221409e+23	1.61803398875
a	6.0221409e+23		1.61803398875	

If **a** is a pixel vector, the value of each element of **r** is taken from the corresponding element of the least-significant half of **a** as follows:

- All bits in the first byte of the element of **r** are set to the value of the first bit of the element of **a**.
- The least-significant 5 bits of the second byte of the element of **r** are set to the value of the next 5 bits in the element of **a**.
- The least-significant 5 bits of the third byte of the element of **r** are set to the value of the next 5 bits in the element of **a**.
- The least-significant 5 bits of the fourth byte of the element of **r** are set to the value of the next 5 bits in the element of **a**.

An example follows:

word index	***0***		***1***		***2***		***3***	
halfword index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***
a	????	????	????	????	DCB8	A984	7652	4321
unpack halfwords to words	DCB8		A984		7652		4321	
as bits	1101110010111000		1010100110000100		0111011001010010		0100001100100001	
as *1-5-5-5*	1	10111 00101 11000	1	01010 01100 00100	0	11101 10010 10010	0	10000 11001 00001
r	FF170518		FF0A0C04		001D1212		00101901	

Endian considerations: The "high" half of a vector with *n* elements is the first *n*/2 elements of the vector. For little endian, these elements are in the rightmost half of the vector. For big endian, these elements are in the leftmost half of the vector.

Table 4.179. Supported type signatures for vec_unpackl

r	ARG1	Example LE Implementation	Example BE Implementation
vector bool short	vector bool char	`vupkhsb r,a`	`vupklsb r,a`
vector signed short	vector signed char	`vupkhsb r,a`	`vupklsb r,a`
vector bool int	vector bool short	`vupkhsh r,a`	`vupklsh r,a`
vector signed int	vector signed short	`vupkhsh r,a`	`vupklsh r,a`
vector unsigned int	vector pixel	`vupkhpx r,a`	`vupklpx r,a`
vector bool long long	vector bool int	`vupkhsw r,a`	`vupklsw r,a`
vector signed long long	vector signed int	`vupkhsw r,a`	`vupklsw r,a`

vector double	vector float	`xxsldwi t,a,a,1`
		`xxsldwi u,t,a,3`
		`xvcvspdp r,u`
			`xxsldwi t,a,a,3`
			`xxsldwi u,a,t,2`
			`xvcvspdp r,u`

vec_unsigned

Vector Convert Floating-Point to Unsigned Integer

```
r = vec_unsigned (a)
```

Purpose: Converts a vector of floating-point numbers to a vector of unsigned integers.

Result value: Each element of **r** is obtained by truncating the corresponding element of **a** to an unsigned integer. The current floating-point rounding mode is ignored.

Endian considerations: None.

Table 4.180. Supported type signatures for vec_unsigned

r	a	Example Implementation
vector unsigned int	vector float	`xvcvspsxws r,a`
vector unsigned long long	vector double	`xvcvdpsxds r,a`

vec_unsigned2

Vector Convert Double-Precision to Unsigned Word

```
r = vec_unsigned2 (a, b)
```

Purpose: Converts two vectors of double-precision floating-point numbers to a vector of unsigned 32-bit integers.

Result value: Let **v** be the concatenation of **a** and **b**. Each element of **r** is obtained by truncating the corresponding element of **v** to an unsigned 32-bit integer. The current floating-point rounding mode is ignored.

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.181. Supported type signatures for vec_unsigned2

r	a	b	Example LE Implementation	Example BE Implementation

vector unsigned int	vector double	vector double	`xxpermdi t,b,a,3`
			`xxpermdi u,b,a,0`
			`xvcvdpuxws v,t`
			`xvcvdpuxws w,u`
			`vmrgow r,w,v`
				`xxpermdi t,a,b,3`
				`xxpermdi u,a,b,0`
				`xvcvdpuxws v,t`
				`xvcvdpuxws w,u`
				`vmrgew r,v,w`

vec_unsignede

Vector Convert Double-Precision to Unsigned Word Even

```
r = vec_unsignede (a)
```

Purpose: Converts elements of the source vector to unsigned integers and stores them in the even-numbered elements of the result vector.

Result value: Element 0 of **r** contains element 0 of **a**, truncated to an unsigned integer. Element 2 of **r** contains element 1 of **a**, truncated to a signed integer. Elements 1 and 3 of **r** are undefined. Truncation of a negative number to an unsigned integer results in a value of zero.

An example follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
a	1.7		-1.0	
r	00000001	???????? *(undefined)*	00000000 *(truncation of a negative number to unsigned is 0)*	???????? *(undefined)*

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.182. Supported type signatures for vec_unsignede

r	a	Example LE Implementation	Example BE Implementation
vector unsigned int	vector double	`xvcvdpuxws t,a` `vsldoi r,t,t,12`	`xvcvdpuxws r,a`

vec_unsignedo

Vector Convert Double-Precision to Unsigned Word Odd

```
r = vec_unsignedo (a)
```

Purpose: Converts elements of the source vector to unsigned integers and stores them in the odd-numbered elements of the result vector.

Result value: Element 1 of **r** contains element 0 of **a**, truncated to an unsigned integer. Element 3 of **r** contains element 1 of **a**, truncated to an unsigned integer. Elements 0 and 2 of **r** are undefined. Truncation of a negative number to an unsigned integer results in a value of zero.

An example follows:

doubleword index	***0***		***1***	
word index	***0***	***1***	***2***	***3***
a	1.7		-1.0	
r	???????? *(undefined)*	00000001	???????? *(undefined)*	00000000 *(truncation of a negative number to unsigned is 0)*

Endian considerations: The element numbering within a register is left-to-right for big-endian targets, and right-to-left for little-endian targets.

Table 4.183. Supported type signatures for vec_unsignedo

r	a	Example LE Implementation	Example BE Implementation
vector unsigned int	vector double	`xvcvdpuxws r,a`	`xvcvdpuxws t,a` `vsldoi r,t,t,12`

vec_xl

VSX Unaligned Load

```
r = vec_xl (a, b)
```

Purpose: Loads a 16-byte vector from the memory address specified by the displacement and the pointer.

Result value: The value of **r** is obtained by adding **a** and **b**, then loading the 16-byte vector from the resultant memory address.

Endian considerations: For ISA 2.07, there is no bi-endian unaligned load instruction. For little-endian targets, it is necessary to use the lxvd2x instruction and swap the doublewords with an xxswapd instruction. For big-endian targets, the lxvd2x instruction or lxvw4x instruction suffices. The examples below assume ISA 3.0, where the bi-endian lxv instruction is available.

Notes:

- For languages that support built-in methods for pointer dereferencing, such as the C/C++ * and [ ] operators, use of the native operators is encouraged when the memory to be accessed is aligned on a 32-bit boundary or aligned to the type of **b**, whichever is weaker.
- GCC provides a commonly used synonym for vec_xl called vec_vsx_ld. Although these have the same behavior, only vec_xl is guaranteed to be portable across compliant compilers. Therefore vec_xl is preferred.

Table 4.184. Supported type signatures for vec_xl

r	a	b	Example ISA 3.0 Implementation
vector signed char	signed long long	const signed char *	`lxv r,a(b)`
vector unsigned char	signed long long	const unsigned char *	`lxv r,a(b)`
vector signed short	signed long long	const signed short *	`lxv r,a(b)`
vector unsigned short	signed long long	const unsigned short *	`lxv r,a(b)`
vector signed int	signed long long	const signed int *	`lxv r,a(b)`
vector unsigned int	signed long long	const unsigned int *	`lxv r,a(b)`
vector signed signed long long	signed long long	const signed long long *	`lxv r,a(b)`
vector unsigned long long	signed long long	const unsigned long long *	`lxv r,a(b)`
vector signed __int128	signed long long	const signed __int128 *	`lxv r,a(b)`

r	a	b	Example ISA 3.0 Implementation
vector unsigned __int128	signed long long	const unsigned __int128 *	`lxv r,a(b)`
vector float	signed long long	const float *	`lxv r,a(b)`
vector double	signed long long	const double *	`lxv r,a(b)`

vec_xl_be

VSX Unaligned Load as Big Endian

```
r = vec_xl_be (a, b)
```

Purpose: Loads a vector from an address into a register in big-endian element order, regardless of the endianness of the target machine.

Result value: The value of **r** is obtained by adding **a** and **b**, then loading the vector elements from the resulting address in big-endian order.

Endian considerations: In big-endian mode, this acts just like the vec_xl intrinsic. In little-endian mode, the highest-numbered element of **r** is loaded from the lowest data address, and the lowest-numbered element of **r** from the highest data address.

Table 4.185. Supported type signatures for vec_xl_be

r	a	b	Example ISA 3.0 LE Implementation	Example ISA 3.0 BE Implementation
vector signed char	signed long long	const signed char *	`lxvb16x r,a,b`	`lxv r,a,b`
vector unsigned char	signed long long	const unsigned char *	`lxvb16x r,a,b`	`lxv r,a,b`
vector signed short	signed long long	const signed short *	`lxvh8x r,a,b`	`lxv r,a,b`
vector unsigned short	signed long long	const unsigned short *	`lxvh8x r,a,b`	`lxv r,a,b`
vector signed int	signed long long	const signed int *	`lxvw4x r,a,b`	`lxv r,a,b`
vector unsigned int	signed long long	const unsigned int *	`lxvw4x r,a,b`	`lxv r,a,b`
vector signed long long	signed long long	const signed long long *	`lxvd2x r,a,b`	`lxv r,a,b`
vector unsigned long long	signed long long	const unsigned long long *	`lxvd2x r,a,b`	`lxv r,a,b`
vector signed __int128	signed long long	const signed __int128 *	`lxv r,a,b`	`lxv r,a,b`
vector unsigned __int128	signed long long	const unsigned __int128 *	`lxv r,a,b`	`lxv r,a,b`
vector float	signed long long	const float *	`lxvw4x r,a,b`	`lxv r,a,b`
vector double	signed long long	const double *	`lxvd2x r,a,b`	`lxv r,a,b`

vec_xl_len

Vector Load with Length

```
r = vec_xl_len (a, b)
```

Purpose: Loads a vector of a specified byte length.

Result value: Loads the number of bytes specified by **b** from the address specified in **a**. Initializes elements in order from the byte stream (as defined by the endianness of the target). Any bytes of elements that cannot be initialized from the number of loaded bytes have a zero value.

Between 0 and 16 bytes, inclusive, will be loaded. The length is specified by the least-significant byte of **b**, as *min* (**b** *mod* 256, 16). The behavior is undefined if the length argument is outside of the range 0–255, or if it is not a multiple of the vector element size.

Endian considerations: None.

Notes: vec_xl_len should not be used to load from cache-inhibited memory.

Table 4.186. Supported type signatures for vec_xl_len

r	a	b	Example Implementation	Restrictions
vector signed char	const signed char *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector unsigned char	const unsigned char *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector signed short	const signed short *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector unsigned short	const unsigned short *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector signed int	const signed int *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector unsigned int	const unsigned int *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector signed long long	const signed long long *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector unsigned long long	const unsigned long long *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector signed __int128	const signed __int128 *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later

r	a	b	Example Implementation	Restrictions
vector unsigned __int128	const unsigned __int128 *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector float	const float *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later
vector double	const double *	size_t	`sldi t,b,56` `lxvl r,a,t`	ISA 3.0 or later

vec_xl_len_r

Vector Load with Length Right-Justified

```
r = vec_xl_len_r (a, b)
```

Purpose: Loads a vector of a specified byte length, right-justified.

Result value: Loads the number of bytes specified by **b** from the address specified in **a**, right justified in **r**. Initializes elements in order from the byte stream (as defined by the endianness of the target). Any bytes of elements that cannot be initialized from the number of loaded bytes have a zero value.

Between 0 and 16 bytes, inclusive, will be loaded. The length is specified by the least-significant byte of **b**, as *min* (**b** *mod* 256, 16). The behavior is undefined if the length argument is outside of the range 0–255.

Endian considerations: None.

Notes: vec_xl_len_r should not be used to load from cache-inhibited memory.

Table 4.187. Supported type signatures for vec_xl_len_r

r	a	b	Example Implementation	Restrictions

vector unsigned char	const unsigned char *	size_t	`sldi  t,b,56`
			`lvsl  u,0,b`
			`lxvll v,a,t`
			`vperm r,v,v,u`
				ISA 3.0 or later

vec_xor

Vector Exclusive OR

```
r = vec_xor (a, b)
```

Purpose: Performs a bitwise XOR of two vectors.

Result value: **v** is the bitwise exclusive OR of **a** and **b**.

Endian considerations: None.

Table 4.188. Supported type signatures for vec_xor

r	a	b	Example Implementation
vector bool char	vector bool char	vector bool char	`xxlxor r,a,b`
vector signed char	vector signed char	vector signed char	`xxlxor r,a,b`
vector unsigned char	vector unsigned char	vector unsigned char	`xxlxor r,a,b`
vector bool short	vector bool short	vector bool short	`xxlxor r,a,b`
vector signed short	vector signed short	vector signed short	`xxlxor r,a,b`
vector unsigned short	vector unsigned short	vector unsigned short	`xxlxor r,a,b`
vector bool int	vector bool int	vector bool int	`xxlxor r,a,b`
vector signed int	vector signed int	vector signed int	`xxlxor r,a,b`
vector unsigned int	vector unsigned int	vector unsigned int	`xxlxor r,a,b`
vector bool long long	vector bool long long	vector bool long long	`xxlxor r,a,b`
vector signed long long	vector signed long long	vector signed long long	`xxlxor r,a,b`
vector unsigned long long	vector unsigned long long	vector unsigned long long	`xxlxor r,a,b`
vector float	vector float	vector float	`xxlxor r,a,b`
vector double	vector double	vector double	`xxlxor r,a,b`

vec_xst

VSX Unaligned Store

```
vec_xst (a, b, c)
```

Purpose: Stores a 16-byte value into memory at the address specified by the displacement and pointer.

Operation: The values of **b** and **c** are added, and the value of **a** is stored to the resultant address.

Endian considerations: For ISA 2.07, there is no bi-endian unaligned store instruction. For little-endian targets, it is necessary to first swap the doublewords of the value to be stored using an xxswapd instruction, and then store the result using the stxvd2x instruction. For big-endian targets, the stxvd2x or stxvw4x instruction suffices. The examples below assume ISA 3.0, where the bi-endian stxv instruction is available.

Notes:

- For languages that support built-in methods for pointer dereferencing, such as the C/C++ * and [ ] operators, use of the native operators is encouraged when the memory to be accessed is aligned on a 32-bit boundary or aligned to the type of **b**, whichever is weaker.
- GCC provides a commonly used synonym for vec_xst called vec_vsx_st. Although these have the same behavior, only vec_xst is guaranteed to be portable across compliant compilers. Therefore vec_xst is preferred.

Table 4.189. Supported type signatures for vec_xst

a	b	c	Example ISA 3.0 Implementation
vector signed char	signed long long	signed char *	`stxv a,b(c)`
vector unsigned char	signed long long	unsigned char *	`stxv a,b(c)`
vector signed short	signed long long	signed short *	`stxv a,b(c)`
vector unsigned short	signed long long	unsigned short *	`stxv a,b(c)`
vector signed int	signed long long	signed int *	`stxv a,b(c)`
vector unsigned int	signed long long	unsigned int *	`stxv a,b(c)`
vector signed long long	signed long long	signed long long *	`stxv a,b(c)`
vector unsigned long long	signed long long	unsigned long long *	`stxv a,b(c)`
vector signed __int128	signed long long	signed __int128 *	`stxv a,b(c)`

a	b	c	Example ISA 3.0 Implementation
vector unsigned __int128	signed long long	unsigned __int128 *	`stxv a,b(c)`
vector float	signed long long	float *	`stxv a,b(c)`
vector double	signed long long	double *	`stxv a,b(c)`

vec_xst_be

VSX Unaligned Store as Big Endian

```
vec_xst_be (a, b, c)
```

Purpose: Stores a vector to an address using big-endian element order, regardless of the endianness of the target machine.

Result value: The values of **b** and **c** are added, and the value of **a** is stored to the resultant address using big-endian element order.

Endian considerations: In big-endian mode, this acts just like the vec_xst intrinsic. In little-endian mode, the lowest data address receives the highest-numbered element of **a**, and the highest data address receives the lowest-numbered element of **a**.

Table 4.190. Supported type signatures for vec_xst_be

a	b	c	Example ISA 3.0 LE Implementation	Example ISA 3.0 BE Implementation
vector signed char	signed long long	signed char *	`stxvb16x a,b,c`	`stxv a,b,c`
vector unsigned char	signed long long	unsigned char *	`stxvb16x a,b,c`	`stxv a,b,c`
vector signed short	signed long long	signed short *	`stxvh8x a,b,c`	`stxv a,b,c`
vector unsigned short	signed long long	unsigned short *	`stxvh8x a,b,c`	`stxv a,b,c`
vector signed int	signed long long	signed int *	`stxvw4x a,b,c`	`stxv a,b,c`
vector unsigned int	signed long long	unsigned int *	`stxvw4x a,b,c`	`stxv a,b,c`
vector signed long long	signed long long	signed long long *	`stxvd2x a,b,c`	`stxv a,b,c`
vector unsigned long long	signed long long	unsigned long long *	`stxvd2x a,b,c`	`stxv a,b,c`
vector signed __int128	signed long long	signed __int128 *	`stxv a,b,c`	`stxv a,b,c`
vector unsigned __int128	signed long long	unsigned __int128 *	`stxv a,b,c`	`stxv a,b,c`
vector float	signed long long	float *	`stxvw4x a,b,c`	`stxv a,b,c`
vector double	signed long long	double *	`stxvd2x a,b,c`	`stxv a,b,c`

vec_xst_len

Vector Store with Length

```
vec_xst_len (a, b, c)
```

Purpose: Stores a vector of a specified byte length.

Operation: Stores the number of bytes specified by **c** of the vector **a** to the address specified in **b**. The bytes are obtained starting from the lowest-numbered byte of the lowest-numbered element (as defined by the endianness of the target). All bytes of an element are accessed before proceeding to the next higher element.

Between 0 and 16 bytes, inclusive, will be stored. The length is specified by the least-significant byte of **c**, as *min* (**c** *mod* 256, 16). The behavior is undefined if the length argument is outside of the range 0–255, or if it is not a multiple of the vector element size.

Endian considerations: None.

Notes: vec_xst_len should not be used to store to cache-inhibited memory.

Table 4.191. Supported type signatures for vec_xst_len

a	b	c	Example Implementation	Restrictions
vector signed char	signed char *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector unsigned char	unsigned char *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector signed short	signed short *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector unsigned short	unsigned short *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector signed int	signed int *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector unsigned int	unsigned int *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector signed long long	signed long long *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector unsigned long long	unsigned long long *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector signed __int128	signed __int128 *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later

a	b	c	Example Implementation	Restrictions
vector unsigned __int128	unsigned __int128 *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector float	float *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later
vector double	double *	size_t	`sldi t,c,56` `stxvl a,b,t`	ISA 3.0 or later

vec_xst_len_r

Vector Store with Length Right-Justified

```
vec_xst_len_r (a, b, c)
```

Purpose: Stores a right-justified vector of a specified byte length.

Operation: Stores the number of bytes specified by **c** of the right-justified vector **a** to the address specified by **b**.

Between 0 and 16 bytes, inclusive, will be stored. The length is specified by the least-significant byte of **c**, as *min* (**b** *mod* 256, 16). The behavior is undefined if the length argument is outside of the range 0–255.

Endian considerations: None.

Notes: vec_xst_len_r should not be used to store to cache-inhibited memory.

Table 4.192. Supported type signatures for vec_xst_len_r

a	b	c	Example Implementation	Restrictions

vector unsigned char	unsigned char *	size_t	`lvsr t,0,c`
			`sldi u,c,56`
			`vperm v,a,a,t`
			`stxvll v,b,u`
				ISA 3.0 or later

5. Instruction/Intrinsic Cross-Reference

This chapter contains a cross-reference from Power hardware instructions to intrinsics that generate them.

C

D

E

L

M

R

S

V

X

Appendix A. OpenPOWER Foundation overview

The OpenPOWER Foundation was founded in 2013 as an open technical membership organization that will enable data centers to rethink their approach to technology. Member companies are enabled to customize POWER CPU processors and system platforms for optimization and innovation for their business needs. These innovations include custom systems for large or warehouse scale data centers, workload acceleration through GPU, FPGA or advanced I/O, platform optimization for SW appliances, or advanced hardware technology exploitation. OpenPOWER members are actively pursing all of these innovations and more and welcome all parties to join in moving the state of the art of OpenPOWER systems design forward.

To learn more about the OpenPOWER Foundation, visit the organization website at openpowerfoundation.org.

A.1. Foundation documentation

Key foundation documents include:

- *Bylaws of OpenPOWER Foundation*
- *OpenPOWER Foundation Intellectual Property Rights (IPR) Policy*
- *OpenPOWER Foundation Membership Agreement*
- *OpenPOWER Anti-Trust Guidelines*

More information about the foundation governance can be found at openpowerfoundation.org/about-us/governance.

A.2. Technical resources

Development resouces fall into the following general categories:

- Foundation work groups
- Remote development environments (VMs)
- Development systems
- Technical specifications
- Software
- Developer tools

The complete list of technical resources are maintained on the foundation Technical Resources web page.

A.3. Contact the foundation

To learn more about the OpenPOWER Foundation, please use the following contact points:

- General information -- <info@openpowerfoundation.org>
- Membership -- <membership@openpowerfoundation.org>
- Technical Work Groups and projects -- <tsc-chair@openpowerfoundation.org>
- Events and other activities -- <admin@openpowerfoundation.org>
- Press/Analysts -- <press@openpowerfoundation.org>

More contact information can be found at openpowerfoundation.org/get-involved/contact-us.