ALTIVECPIM/D
6/1999
Rev. 0

AltiVec™ Technology Programming Interface Manual

Digital DNA from Motorola

PowerPC™

MOTOROLA

Motorola Literature Distribution Centers:
USA/EUROPE: Motorola Literature Distribution; P.O. Box 5405; Denver, Colorado 80217; Tel.: 1-800-441-2447 or 1-303-675-2140/
JAPAN: Nippon Motorola Ltd SPD, Strategic Planning Office 4-32-1, Nishi-Gotanda Shinagawa-ku, Tokyo 141, Japan Tel.: 81-3-5487-8488
ASIA/PACIFC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park, 51 Ting Kok Road, Tai Po, N.T., Hong Kong; Tel.: 852-26629298

Mfax™: RMFAX0@email.sps.mot.com; TOUCHTONE 1-602-244-6609; US & Canada ONLY (800) 774-1848;
World Wide Web Address: http://sps.motorola.com/mfax
INTERNET: http://motorola.com/sps

Technical Information: Motorola Inc. SPS Customer Support Center 1-800-521-6274; electronic mail address: crc@wmkmail.sps.mot.com.
Document Comments: FAX (512) 895-2638, Attn: RISC Applications Engineering.
World Wide Web Addresses: http://www.mot.com/PowerPC
http://www.mot.com/netcomm
http://www.mot.com/HPESD

CONTENTS

Paragraph Number	Title	Page Number

Chapter 1
Overview

Chapter 2
High-Level Language Interface

Chapter 3
Application Binary Interface (ABI)

CONTENTS

Paragraph Number	Title	Page Number

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

Figure Number	Title	Page Number

TABLES

TABLES

Table Number	Title	Page Number

About This Book

The primary objective of this manual is to help programmers to provide software that is compatible across the family of PowerPC™ processors using AltiVec™ technology.

To locate any published errata or updates for this document, refer to the website at http://www.mot.com/SPS/PowerPC/.

This book is one of two that discuss the AltiVec architecture, the two books are:

- *AltiVec: The Programming Interface Manual (AltiVec PIM)* is used as a reference guide for high-level programmers. The AltiVec PIM provides a mechanism for programmers to access AltiVec functionality from programming languages such as C and C++. The AltiVec PIM defines a programming model for use with the AltiVec instruction set extension to the PowerPC architecture.
- *AltiVec: The Programming Environments Manual (AltiVec PEM)* is used as a reference guide for assembler programmers. The AltiVec PEM provides a description for each instruction that includes the instruction format, an individualized legend that provides such information as the level(s) of the PowerPC architecture in which the instruction may be found, the privilege level of the instruction, and figures to help in understanding how the instruction works.

It is beyond the scope of this manual to describe individual AltiVec technology implementations on PowerPC processors. It must be kept in mind that each PowerPC processor is unique in its implementation of the AltiVec technology.

The information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation. For more information, contact your sales representative or visit our website at: http://www.mot.com/SPS/PowerPC/.

Audience

This manual is intended for system software and application programmers who want to develop products using the AltiVec technology extension to the PowerPC processors in general. It is assumed that the reader understands operating systems, microprocessor system design, the basic principles of RISC processing, and the AltiVec Instruction Set.

Organization

Following is a summary and a brief description of the major sections of this manual:

- Chapter 1, "Overview," is useful for those who want a general understanding of what the programming model defines in the AltiVec technology.
- Chapter 2, "High-Level Language Interface," is useful for software engineers who need to understand how to access AltiVec functionality from high level languages such as C and C++.
- Chapter 3, "Application Binary Interface (ABI)," describes AltiVec extensions for System V Application Binary Interface PowerPC Processor Supplement (SVR4 ABI), the PowerPC Embedded Application Binary Interface (EABI), Appendix A of The PowerPC Compiler Writer's Guide (AIX ABI), and the Apple Macintosh ABI.
- Chapter 4, "AltiVec Operations and Predicates," alphabetically defines the AltiVec operations and predicates. Each AltiVec operation and predicate description includes a pseudocode functional description and figures illustrating that function, a valid set of argument types for that AltiVec operation or predicate, the result type for that set of argument types, and the specific AltiVec instruction generated for that set of arguments.
- Appendix A, "AltiVec Instruction Set/Operation/Predicate Cross-Reference," cross-references the AltiVec instruction set, operations, and predicates by functionality.
- This manual also includes a glossary and an index.

Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the AltiVec technology and PowerPC architecture.

PowerPC Documentation

The PowerPC documentation is organized in the following types of documents:

- User's manuals—These books provide details about individual PowerPC implementations and are intended to be used in conjunction with *PowerPC Microprocessor Family: The Programming Environments Manual*.
- *PowerPC Microprocessor Family: The Programming Environments*, Rev. 1 provides information about resources defined by the PowerPC architecture that are common to PowerPC processors. This document describes both the 64- and 32-bit portions of the architecture.
  MPCFPE/AD (Motorola order #)
- *Implementation Variances Relative to Rev. 1 of The Programming Environments Manual* is available via the world-wide web at http://www.mot.com/SPS/PowerPC/.
- Addenda/errata to user's manuals—Because some processors have follow-on parts an addendum is provided that describes the additional features and changes to functionality of the follow-on part. These addenda are intended for use with the corresponding user's manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations for each PowerPC implementation.
- Technical Summaries—Each PowerPC implementation has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's user's manual.
- *PowerPC Microprocessor Family: The Programmer's Reference Guide*: MPCPRG/D (Motorola order #) is a concise reference that includes the register summary, memory control model, exception vectors, and the PowerPC instruction set.
- *PowerPC Microprocessor Family: The Programmer's Pocket Reference Guide*: MPCPRGREF/D (Motorola order #): This foldout card provides an overview of the PowerPC registers, instructions, and exceptions for 32-bit implementations.
- Application notes—These short documents contain useful information about specific design issues useful to programmers and engineers working with PowerPC processors (available via the worldwide web at http://www.mot.com/SPS/PowerPC/).
- Documentation for support chips

Additional literature on AltiVec technology and PowerPC implementations is being released as new processors become available. For a current list of AltiVec technology and PowerPC documentation, refer to the website at http://www.mot.com/SPS/PowerPC/.

General Information

The following documentation provides useful information about the PowerPC architecture and computer architecture in general:

- The following books are available from the Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104; Tel. (800) 745-7323 (U.S.A.), (415) 392-2665 (International); internet address: mkp@mkp.com.
  - *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.

    Updates to the architecture specification are accessible via the world-wide web at *http://www.austin.ibm.com/tech/ppc-chg.html*.
  - *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*, by Apple Computer, Inc., International Business Machines, Inc., and Motorola, Inc.
  - *Macintosh Technology in the Common Hardware Reference Platform*, by Apple Computer, Inc.
  - *Computer Organization and Design*, by David A. Patterson and John L. Hennessy.
  - *Computer Architecture: A Quantitative Approach*, Second Edition, by John L. Hennessy and David A. Patterson.
- *PowerPC Programming for Intel Programmers*, by Kip McClanahan; IDG Books Worldwide, Inc., 919 East Hillsdale Boulevard, Suite 400, Foster City, CA, 94404; Tel. (800) 434-3422 (U.S.A.), (415) 655-3022 (International).

Chapter 1
Overview

This document defines a programming model for use with the AltiVec instruction set extension to the PowerPC architecture. There are three types of programming interfaces described in this document:

- A high-level language interface, intended for use within programming languages such as C or C++
- An application binary interface (ABI) defining low-level coding conventions
- An assembly language interface

Although a higher-level application programming interface (API) such as `mediaLib` is intended for use with AltiVec, such a specification is not addressed by this document. For further details on mediaLib see the AltiVec website at: http://www.mot.com/SPS/PowerPC/AltiVec.

An AltiVec-enabled compiler implementing the model described in this document predefines the value `__VEC__` as the decimal integer 10205.

1.1 High-Level Language Interface

The high-level language interface for AltiVec is a way for programmer to be able to use the AltiVec technology from programming languages such as C and C++. It describes fundamental data type for the AltiVec programming model. Details of this interface are described in Chapter 2, “High-Level Language Interface.”

1.2 Application Binary Interface (ABI)

The AltiVec Programming Model extends the existing PowerPC ABIs and the extension is independent of the endian mode. The ABI reviews what the data types are and what the register usage conventions are for vector register files. The ABI also discusses how to set up the stack frame. The vector register save and restore functions are included in the ABI section to advocate uniformity among compilers on the method used in saving and restoring vector registers.

The Programming Interface Manual provides the valid set of argument types for specific AltiVec operations and predicates as well as the specific AltiVec instruction(s) generated for that set of arguments. The AltiVec operations and predicates are organized alphabetically in Chapter 4, “AltiVec Operations and Predicates.”

Chapter 2
High-Level Language Interface

The AltiVec high-level language interface:

- Provides an efficient and expressive mechanism for programmers to access AltiVec functionality from programming languages such as C and C++.

  Note: Access to AltiVec functionality from Java applications is not currently addressed by this specification, but will likely be addressed through a higher level API such as `mediaLib`.
- Defines a minimal set of language extensions that clearly describes the intent of the programmer while minimizing the impact on existing PowerPC compilers and development tools.
- Defines a minimal set of library extensions needed to support AltiVec functionality.

2.1 Data Types

The AltiVec programming model introduces a set of fundamental data types, as described in Table 2-1.

Table 2-1. AltiVec Data Types

New C/C++ Type	Interpretation of Contents	Components Represent Values
vector unsigned char	16 unsigned char	0...255
vector signed char	16 signed char	-128...127
vector bool char	16 unsigned char	0(F), 255 (T)
vector unsigned short	8 unsigned short	0...65536
vector unsigned short int		
vector signed short	8 signed short	-32768...32767
vector signed short int		
vector bool short	8 unsigned short	0 (F), 65535 (T)
vector bool short int		
vector unsigned int	4 unsigned int	$0...2^{32} - 1$
vector unsigned long*		
vector unsigned long int*		

Table 2-1. AltiVec Data Types (Continued)

New C/C++ Type	Interpretation of Contents	Components Represent Values
vector signed int	4 signed int	$-2^{31}...2^{31}-1$
vector signed long*		
vector signed long int*		
vector bool int	4 unsigned int	0 (F), 2^{32} - 1 (T)
vector bool long*		
vector bool long int*		
vector float	4 float	IEEE-754 values
vector pixel	8 unsigned short	1/5/5/5 pixel

*The vector types with the long keyword are deprecated and will be eliminated in a future version of this document.

In illustrations where an algorithm could apply to multiple types, `vec_data` represents any one of these types. Introducing fundamental types permits the compiler to provide stronger type checking and supports overloaded operations on vector types.

2.2 New Keywords

The model introduces new uses for the following five identifiers:

- `vector`
- `__vector`
- `pixel`
- `__pixel`
- `bool`

as simple type specifier keywords. Among the type specifiers used in a declaration, the `vector` type specifier must occur first. As in C and C++, the remaining type specifiers may be freely intermixed in any order, possibly with other declaration specifiers. The syntax does not allow the use of a `typedef` name as a type specifier. For example, the following is not allowed:

```
typedef signed short int16;
vector int16 data;
```

These new uses may conflict with their existing use in C and C++. There are two methods that may be used to deal with this conflict. An implementation of the AltiVec programming model may choose either method.

2.2.1 The Keyword and Predefine Method

In this method, `__vector`, `__pixel`, and `bool` are added as keywords while `vector` and `pixel` are predefined macros. `bool` is already a keyword in C++. To allow its use in C as a keyword, it is treated the same as it is in C++. This means that the C language is extended to allow `bool` alone as a set of type specifiers. Typically, this type will map to `int`. To accommodate a conflict with other uses of the identifiers `vector` and `pixel`, the user can either `#undef` or use a command line option to remove the predefines.

2.2.2 The Context Sensitive Keyword Method

In this method, `__vector` and `__pixel` are added as keywords without regard to context while the new uses of `vector`, `pixel`, and `bool` are keywords only in the context of a type. Since `vector` must be first among the type specifiers, it can be recognized as a type specifier when a type identifier is being scanned. The new uses of `pixel` and `bool` occur after vector has been recognized. In all other contexts, `vector`, `pixel`, and `bool` are not reserved. This avoids conflicts such as `class vector`, `typedef int bool`, and allows the use of `vector`, `pixel`, and `bool` as identifiers for other uses.

2.3 Alignment

The following paragraphs described AltiVec alignment requirements. When working with vector data, the programmer must be aware of these alignment issues. Because the AltiVec technology does not generate exceptions, the programmer must determine whether and when vector data becomes unaligned.

2.3.1 Alignment of Vector Types

A defined data item of any vector data type in memory is always aligned on a 16-byte boundary. A pointer to any vector data type always points to a 16-byte boundary. The compiler is responsible for aligning vector data types on 16-byte boundaries. Given that vector data is correctly aligned, a program is incorrect if it attempts to dereference a pointer to a vector type if the pointer does not contain a 16-byte aligned address. In the AltiVec architecture, an unaligned load/store does not cause an alignment exception that might lead to (slow) loading of the bytes at the given address. Instead, the low-order bits of the address are quietly ignored.

2.3.2 Alignment of Non-Vector Types

An array of components to be loaded into vector registers need not be aligned, but will have to be accessed with attention to its alignment. Typically, this is accomplished using either the Load Vector for Shift Right, `vec_lvsr(),` or Load Vector for Shift Left, `vec_lvsl(),` operation and the Vector Permute, `vec_perm(),` operation.

2.3.3 Alignment of Aggregates and Unions Containing Vector Types

Aggregates (structures and arrays) and unions containing vector types must be aligned on 16-byte boundaries and their internal organization padded, if necessary, so that each internal vector type is aligned on a 16-byte boundary. This is an extension to all ABIs (AIX, Apple, SVR4, and EABI).

2.4 Extensions of C/C++ Operators for the New Types

Most C/C++ operators do not permit any of their arguments to be one of the new types. Let `a` and `b` be vector types and `p` be a pointer to a vector type. The normal C/C++ operators are extended to include the following operations.

2.4.1 sizeof()

The operations `sizeof(a)` and `sizeof(*p)` return 16.

2.4.2 Assignment

If either the left hand side or right hand side of an expression has a vector type, then both sides of the expression must be of the same vector type. Thus, the expression `a = b` is valid and represents assignment if `a` and `b` are of the same vector type (or if neither is a vector type). Otherwise, the expression is invalid and must be signaled as an error by the compiler.

2.4.3 Address Operator

The operation `&a` is valid if `a` is a vector type. The result of the operation is a pointer to `a`.

2.4.4 Pointer Arithmetic

The usual pointer arithmetic can be performed on `p`. In particular, `p+1` is a pointer to the next vector after `p`.

2.4.5 Pointer Dereferencing

If `p` is a pointer to a vector type, `*p` implies either a 128-bit vector load from the address obtained by clearing the low order bits of `p`, equivalent to the instruction `vec_ld(0, p)` or a 128-bit vector store to that address equivalent to the instruction `vec_st(0, p)`. If it is desired to mark the data accessed as least-recently-used (LRU), the explicit instruction `vec_ldl(0,p)` or `vec_stl(0, p)` must be used.

Dereferencing a pointer to a non-vector type produces the standard behavior of either a load or a copy of the corresponding type.

Accessing of unaligned memory must be carried out explicitly by a `vec_ld(int, type *)` operation, a `vec_ldl(int, type *)` operation, a `vec_st(int, type *)` operation or a `vec_stl(int, type *)` operation.

2.4.6 Type Casting

Pointers to old and new types may be cast back and forth to each other. Casting a pointer to a new type represents an unchecked assertion that the address is 16-byte aligned. Some new operators are provided to provide the equivalence of casts and data initialization.

Casts from one vector type to another are provided by normal C casts. These should not be needed frequently if the overloaded forms of operators are used. None of the casts performs a conversion; the bit pattern of the result is the same as the bit pattern of the argument that is cast.

- `(vector signed char) vec_data`
- `(vector signed short) vec_data`
- `(vector signed int) vec_data`
- `(vector unsigned char) vec_data`
- `(vector unsigned short) vec_data`
- `(vector unsigned int) vec_data`
- `(vector bool char) vec_data`
- `(vector bool short) vec_data`
- `(vector bool int) vec_data`
- `(vector float) vec_data`
- `(vector pixel) vec_data`

Casts between vector types and scalar types are illegal. To copy data between these types, us the `vec_lde()` or `vec_ste()` operations. An alternative is to use a union consisting of a vector type and an equivalent array of the scalar type and copy the data using the union.

2.5 New Operators

New operators are introduced to construct vector literals, adjust pointers, and allow full access to the functionality provided by the AltiVec architecture.

2.5.1 Vector Literals

A vector literal is written as a parenthesized vector type followed by a parenthesized set of constant expressions. Vector literals may be used either in initialization statements or as constants in executable statements. Table 2-2 lists the formats and descriptions of the vector literals. For each, the compiler generates code that either computes or loads the values into the register.

Table 2-2. Vector Literal Format and Description

Notation	Represents
`(vector unsigned char) (unsigned int)`	A set of 16 unsigned 8-bit quantities which all have the value specified by the integer.
`(vector unsigned char) (unsigned int, ..., unsigned int)`	A set of 16 unsigned 8-bit quantities specified by the 16 integers.
`(vector signed char) (int)`	A set of 16 signed 8-bit quantities that all have the value specified by the integer.
`(vector signed char) (int, ..., int)`	A set of 16 signed 8-bit quantities specified by the 16 integers.
`(vector unsigned short) (unsigned int)`	A set of eight unsigned 16-bit quantities which all have the value specified by the unsigned integer.
`(vector unsigned short) (unsigned int, ..., unsigned int)`	A set of eight unsigned 16-bit quantities specified by the eight unsigned integers.
`(vector signed short) (int)`	A set of eight signed 16-bit quantities which all have the value specified by the integer.
`(vector signed short) (int, ..., int)`	A set of eight signed 16-bit quantities specified by the eight integers.
`(vector unsigned int) (unsigned int)`	A set of four unsigned 32-bit quantities which all have the value specified by the unsigned integer.
`(vector unsigned int) (unsigned int, ..., unsigned int)`	A set of four unsigned 32-bit quantities specified by the four unsigned integers.
`(vector signed int) (int)`	A set of four signed 32-bit quantities which all have the value specified by the integer.
`(vector signed int) (int, ..., int)`	A set of four signed 32-bit quantities specified by the 4 integers.
`(vector float) (float)`	A set of four floating-point quantities which all have the value specified by the floating-point value.
`(vector float) (float, ..., float)`	A set of four floating-point quantities which all have the value specified by the four floating-point values.

2.5.2 Vector Literals and Casts

The combination of vector casts and vector literals can complicate some parsers. An implementation is not required to support the cast to a vector type of a vector cast or vector literal when the operand of the cast is not a parenthesized expression. For example, the programmer may write the following:

```
(vector unsigned char)((vector unsigned int)(1, 2, 3, 4))
(vector signed char)((vector unsigned short) variable)
```

The similar expressions below without the parenthesized expression may not be used in a conforming application

```
(vector unsigned char)(vector unsigned int)(1, 2, 3, 4)
(vector signed char)(vector unsigned short) variable
```

2.5.3 Value for Adjusting Pointers

At compile time, the `vec_step(vec_data)` produces the integer value representing the amount by which a pointer to a component of an AltiVec data should increment to cause a pointer increment to increment by 16 bytes. For example, a `vector unsigned short` data type is considered to contain eight unsigned 2-byte values. A pointer to unsigned 2-byte values used to stream through an array of unsigned 2-byte values by a full vector at a time should increment by `vec_step(vector unsigned short) = 8`. Table 2-3 provides a summary of the values by data type.

Table 2-3. Increment Value for vec_step by Data Type

vec_step Expression	Value

`vec_step(vector unsigned char)`
`vec_step(vector signed char)`
`vec_step(vector bool char)`
	16

`vec_step(vector unsigned short)`
`vec_step(vector signed short)`
`vec_step(vector bool short)`
	8

`vec_step(vector unsigned int)`
`vec_step(vector signed int)`
`vec_step(vector bool int)`
	4

`vec_step(vector pixel)`	8
`vec_step(vector float)`	4

2.5.4 New Operators Representing AltiVec Operations

New operators are introduced to allow full access to the functionality provided by the AltiVec architecture. The new operators are represented in the programming language by language structures that parse like function calls. The names associated with these operations are all prefixed with `vec_`. The appearance of one of these forms can indicate the following:

- A generic AltiVec operation, like `vec_add()`
- A specific AltiVec operation, like `vec_addubm()`
- A predicate computed from a AltiVec operation like `vec_all_eq()`
- Loading of a vector of components, as discussed in Section 2.5.1, “Vector Literals”

Each AltiVec operator takes a list of arguments that represent the input operands. The order of the operands is prescribed in the architecture specification and includes a returned result (possibly `void`).

The programming model restricts the operand types permitted for each AltiVec operation, whether specific or generic. The programmer may override this constraint by explicitly casting arguments to permissible types.

For a specific operation, the operand types determine whether the operation is acceptable within the programming model and the type of the result. For example, `vec_vaddubm(vector signed char, vector signed char)` is acceptable in the programming model because it represents a reasonable way to do modular addition with signed bytes, while `vec_vaddubs(vector signed char, vector signed char)` and `vec_vaddubh(vector signed char, vector signed char)` are not acceptable. If permitted, the former operation would produce a result in which saturation treats the operands as unsigned; the latter operation would produce a result in which adjacent pairs of signed bytes are treated as signed halfwords.

For a generic operation, the operand types are used to determine whether the operation is acceptable, to select a particular operation according to the types of the arguments, and to determine the type of the result. For example, `vec_add(vector signed char, vector signed char)` will map onto `vec_vaddubm()` and return a result of type `vector signed char`, while `vec_add(vector unsigned short, vector unsigned short)` maps onto `vec_vadduhm()` and return a result of type `vector unsigned short`.

The AltiVec operations that set condition register CR6 (i.e., the compare dot instructions) are treated somewhat differently in the programming model. The programmer can not access specific register names. Instead of directly specifying a compare dot instruction, the programmer makes reference to a predicate that returns an integer value derived from the result of a compare dot instruction. As in C, this value may be used directly as a value (1 is true, 0 is false) or as a condition for branching. It is expected that the compiler will produce the minimum code needed to use the condition. Predicates begin with `vec_all_` or `vec_any_`. Either the true or false state of any bit that can be set by a compare dot instruction has a predicate. For example, `vec_all_gt(x,y)` tests the true value of bit 24 of the CR after executing some `vcmpgt.` instruction. To complete the coverage by predicates, additional predicates exercise compare dot instructions with reversed or duplicated arguments. As examples, `vec_all_lt(x,y)` performs a `vcmpgtx.(y,x)`, and `vec_all_nan(x)` is mapped onto `vcmpeqfp.(x,x)`. If the programmer wishes to have both the result of the compare dot instruction as returned in the vector register and the value of CR6, the programmer specifies two operations. The compiler's job is to determine that these can be merged. The AltiVec operations and predicates are listed in Chapter 4, "AltiVec Operations and Predicates".

2.6 Programming Interface

This document does not prohibit or require an implementation to provide any set of `include` files or `#pragma` preprocessor commands. If an implementation requires that an `include` file be used prior to the use of the syntax described in this document, it is suggested that the `include` file be named `<altivec.h>`. If an implementation supports `#pragma` preprocessor commands, it is suggested that it provide `__ALTIVEC__` as a predefined macro with a nonzero value. A suggested preprocessor command set includes the following:

```
#pragma altivec_codegen on | off
```

When this pragma is on, the compiler may use AltiVec instructions. When you set this pragma `off`, the `altivec_model` pragma is also set to `off`.

```
#pragma altivec_model on | off
```

When this pragma is `on`, the compiler accepts the syntax specified in this document, and the `altivec_codegen` pragma is also set to `on`.

```
#pragma altivec_vrsave on | off | allon
```

When this pragma is `on`, the compiler maintains the VRSAVE register. With `allon` selected, the compiler changes the VRSAVE register to have all bits set. It is combined with `#pragma altivec_vrsave off` by having a parent function do the work once of setting the value of the VRSAVE register with `#pragma altivec_vrsave allon` and the function it calls uses the setting `#pragma altivec_vrsave off`.

Chapter 3
Application Binary Interface (ABI)

Note: The ABI extensions described herein for embedded applications are still under review by the PowerPC EABI industry working group, and may be subject to change. Modifications, if any, will be highlighted in future revisions of this document.

The AltiVec programming model extends the existing PowerPC ABIs. This chapter specifies extensions to the System V Application Binary Interface PowerPC Processor Supplement (SVR4 ABI), the PowerPC Embedded Application Binary Interface (EABI), Appendix A of The PowerPC Compiler Writer's Guide (AIX ABI), and the Apple Macintosh ABI. The SVR4 ABI and EABI specifications define both a Big-Endian ABI and a Little-Endian ABI. This extension is independent of the endian mode.

3.1 Data Representation

The vector data types are 16-bytes long and 16-byte aligned. All ABIs are extended similarly. Aggregates (structures and arrays) and unions containing vector types must be aligned on 16-byte boundaries and their internal organization padded, if necessary, so that each internal vector type is aligned on a 16-byte boundary. The Apple ABI and AIX ABI specify a maximum alignment for aggregates and unions of 4-bytes; the EABI specifies a maximum alignment of 8-bytes. Increasing the alignment to 16-bytes creates the opportunity for padding or holes in the parameter lists involving these aggregates described in Section 3.4.2, "Apple Macintosh ABI and AIX ABI Parameter Passing without Varargs."

3.2 Register Usage Conventions

The register usage conventions for the vector register file are defined as follows:

Table 3-1. AltiVec Registers

Register	Intended use	Behavior across call sites
v0–v1	General use	Volatile (Caller save)
v2–v13	Parameters, general	Volatile (Caller save)
v14–v19	General	Volatile (Caller save)
v20-v31	General	Non-volatile (Callee save)

Table 3-1. AltiVec Registers

Register	Intended use	Behavior across call sites
VRSAVE	Special, see Section 3.3, "The Stack Frame	Non-volatile (Callee save)

The VRSAVE special purpose register (SPR256, named vrsave in assembly instructions) is used to inform the operating system which vector registers (VRs) need to be saved and reloaded across context switches. Bit *n* of this register is set to 1 if vector register vn needs to be saved and restored across a context switch. Otherwise, the operating system may return that register with any value that does not violate security after a context switch. The most significant bit in the 32-bit word is bit 0.

The EABI does not use VRSAVE for any special purpose, but VRSAVE is a non-volatile register.

3.3 The Stack Frame

The stack pointer maintains 16-byte alignment in the SVR4 ABI and the AIX ABI and 8-byte alignment in the EABI and the Apple Macintosh ABI and AIX ABI. It is not necessary to align the stack dynamically in either the SVR4 ABI or the AIX ABI, however, the alignment padding space is specified for both. The additions to the stack frame are the vector register save area, the vrsave word, and the alignment padding space to dynamically align the stack to a quadword boundary.

The following additional requirements apply to the stack frame:

- Before a function changes the value of vrsave, it shall save the value of VRSAVE at the time of entry to the function in the vrsave word.
- The alignment padding space shall be either 0, 4, 8, or 12 bytes long so that the address of the vector register save area (and subsequent stack locations) are quadword aligned.
- If the code establishing the stack frame dynamically aligns the stack pointer, it shall update the stack pointer atomically with an stwux instruction. The code may assume the stack pointer on entry is aligned on an 8-byte boundary.
- Before a function changes the value in any non-volatile vector register, vn, it shall save the value in vn in the word in the vector register save area 16*(32–*n*) bytes before the low-addressed end of the alignment padding space.
- Local variables of a vector data type which need to be saved to memory will be placed on the stack frame on a 16-byte alignment boundary in the same stack frame region used for local variables of other types.

SP in the figures denotes the stack pointer (general purpose register r1) of the called function after it has executed code establishing its stack frame.

3.3.1 SVR4 ABI and EABI Stack Frame

The size of the vector register save area and the presence of the VRSAVE word may vary within a function and are determined by a new registers valid tag. Note: In the SVR4 ABI, the registers valid tag is the most general way to describe a stack frame. It is associated with a frame or frame valid tag. Figure 3-1 shows an SVR4 and EABI stack frame.

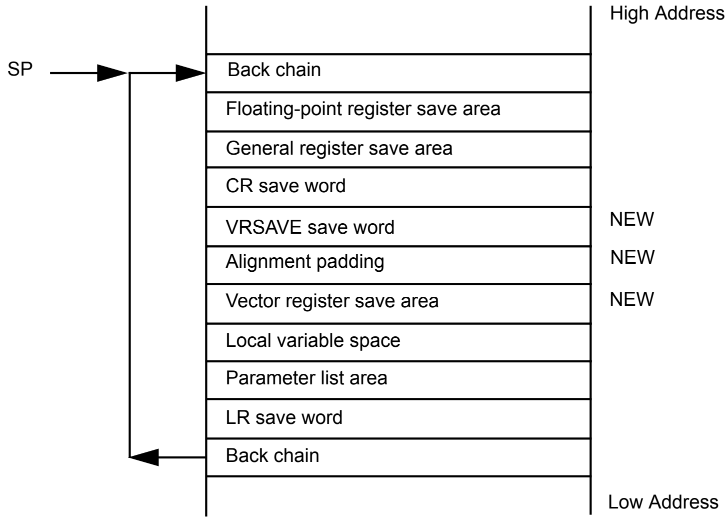


Figure 3-1. SVR4 ABI and EABI Stack Frame

Table 3-2. Vector Registers Valid Tag Format

Word	Bits	Name	Description
1	0–17	RESERVED	0
1	18–29	START_OFFSET	The number of words between the BASE of the nearest preceding Frame or Frame Valid tag and the first instruction to which this tag applies.
1	30–31	TYPE	2
2	0–11	VECTOR_REGS	One bit for each non-volatile vector register, bit 0 for v31,..., bit 11 for v20, with a 1 signifying that the register is saved in the vector register save area.
2	12	VRSAVE_AREA[1]	1 if and only if the VRSAVE word is allocated in the register save area.

1.If more than one Vector Registers Valid Tag applies to the same Frame or Frame Valid tag, they shall all have the same values for VRSAVE_AREA and VR.

Table 3-2. Vector Registers Valid Tag Format

Word	Bits	Name	Description
2	13-17	VR[1]	Size in quadwords of the vector register save area.
2	18-29	RANGE	The number of words between the first and the last instruction to which this tag applies.
2	30	VRSAVE_REG	1 if and only if VRSAVE is saved in the VRSAVE word.
2	31	SUBTYPE	1

1.If more than one Vector Registers Valid Tag applies to the same Frame or Frame Valid tag, they shall all have the same values for VRSAVE_AREA and VR.

The code example below shows sample prologue and epilogue code with full saves of all the non-volatile floating-point (FPRs), general (GPRs), and VRs for a stack frame of less than 32 Kbytes. The example aligns the stack pointer dynamically, addresses incoming arguments via r30, uses volatile VRs v0–v10, maintains VRSAVE, does not alter the non-volatile fields of the CR and does no dynamic stack allocation. Saving and restoring the VRs and updating vrsave can occur in either order. A function that does not need to address incoming arguments but does align the stack pointer dynamically can recover the address of the original stack pointer with an instruction such as lwz r11,0(sp). The computation of `len` in the example and whether to use subfic or addi to align the stack dynamically is based on the size of the components of the frame. Starting with the components at higher addresses, the value of `len` is computed by adding the size of the FPR save area, the GPR save area, the CR save word, and the VRSAVE word.

The size of the alignment padding space is then computed as the smallest number of bytes needed to make `len` a multiple of 16. In the example below, the alignment padding space is 4 bytes. Consequently, subfic is used to dynamically align the stack by increasing the size of the alignment padding space by either 0 or 8 bytes. Had the alignment padding space been 8 or 12 bytes, addi would be used to align the stack dynamically by decreasing the size of the alignment padding space by either 0 or 8 bytes. Continuing, the value of `len` is updated by adding the size of the vector register save area, the local variable space, the outgoing parameter list area, and the LR save word. The size of the local variable space is adjusted so that the overall value of `len` is a multiple of 16. The following is SVR4 ABI and EABI prologue and epilogue sample code.

```
function: mflr      r0                 # Save return address ...
          stw       r0,4(sp)           # ... in caller's frame.
          ori       r11,sp,0           # Save end of fpr save area
          rlwinm    r12,sp,0,28,28     # 0 or 8 based on SP alignment
          subfic    r12,r12,-len       # Add in stack length
          stwux     sp,sp,r12          # Establish new aligned frame
          bl        _savefpr_14        # Save floating-point registers
          addi      r11,r11,-144       # Compute end of gpr save area
          bl        _savegpr_14_g      # Save gprs and fetch GOT ptr
          mflr      r31                # Place GOT ptr in r31
                                       # Save CR here if necessary
          addi      r30,r11,144        # Save pointer to incoming
```

```
                                    #       arguments
mfspr      r0,vrsave                # Save VRSAVE ...
stw        r0,-220(r30)             # ... in caller's frame.
oris       r0,r0,0xff70             # Use v0-v10 and ...
ori        r0,r0,0x0fff             # v20-v31 (for example)
mtspr      vrsave,r0                # Update VRSAVE
addi       r0,sp,len-224            # Compute end of vr save area
bl         _savevr20                # Save VRs
                                    # Body of function
addi       r0,sp,len-224            # Address of vr save area to r0
bl         _restvr20                # Restore VRs
lwz        r0,-220(r30)             # Fetch prior value of VRSAVE
mtspr      vrsave,r0                # Restore VRSAVE
addi       r11,r30,-144             # Address of gpr save area to r11
bl         _restgpr_14              # Restore gprs
addi       r11,r11,144              # Address of fpr save area to r11
bl         _restfpr_14_x            # Restore fprs and return
```

3.3.2 Apple Macintosh ABI and AIX ABI Stack Frame

Figure 3-2 shows how the Apple Macintosh ABI and AIX ABI stack frame is set up.

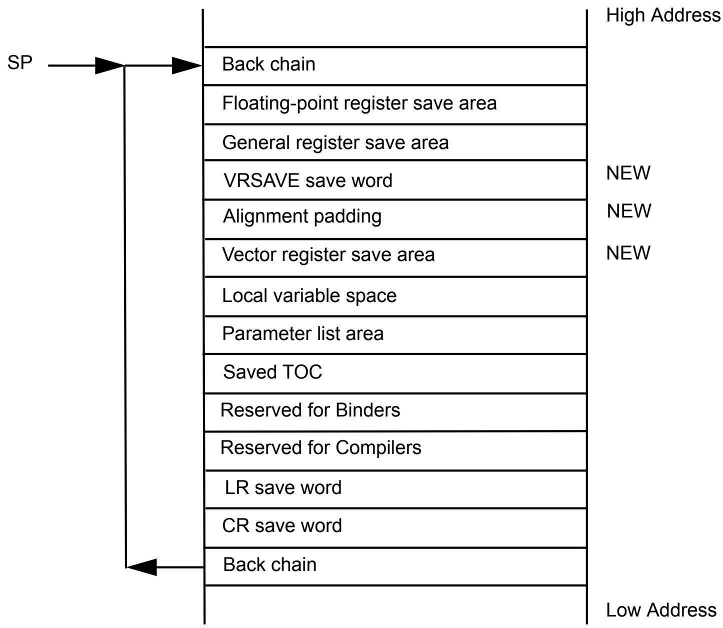


Figure 3-2. Apple Macintosh ABI and AIX ABI Stack Frame

The Apple Macintosh ABI and AIX ABI stack frame allow the use of a 220-byte area at a negative offset from the stack pointer. This area can be used to save non-volatile registers before the stack pointer has been updated. This size of this area is not changed. Depending on the number of non-volatile registers saved, it may be necessary to update the stack pointer before saving the VRs. However, it remains unnecessary to update the stack pointer before saving the GPRs or FPRs.

The size of the VR save area and the presence of the VRSAVE word are determined by a traceback table entry. The `spare3` 2-bit field in the fixed portion of the traceback table is changed to the following:

`has_vec_info`	This 1-bit field is set if the procedure saves non-volatile VRs in the vector register save area, saves `vrsave` in the VRSAVE word, specifies the number of vector parameters, or uses AltiVec instructions.
`spare4`	Reserved 1-bit field.

When the `has_vec_info` bit is set, all the following optional fields of the traceback table are present following the position of the `alloca_reg` field.

`vr_saved`	This 6-bit field represents the number of non-volatile VRs saved by this procedure. Because the last register saved is always v31, a value of 2 in `vr_saved` indicates that v30 and v31 are saved.
`saves_vrsave`	If this routine saves vrsave, this 1-bit field is set. If so, the VRSAVE word in the register save area must be used to restore the prior value before returning from this procedure.
`has_varargs`	If this function has a variable argument list, this 1-bit field is set. Otherwise, it is set to 0.
`vectorparms`	This 7-bit field records the number of vector parameters. The field may be set to a non-zero value for a procedure with vector parameters that does not have a variable argument list. Otherwise, `parmsonstk` must be set.
`vec_present`	This 1-bit field is set if AltiVec instructions are performed within the procedure.

The following code shows sample prologue and epilogue code with full saves of all the non-volatile floating-point, general, and VRs for a stack frame of less than 32 Kbytes. The code example dynamically aligns the stack pointer, addresses incoming arguments via r31, uses volatile VRs v0–v10, maintains VRSAVE, does not alter the non-volatile fields of the CR and does no dynamic stack allocation. Saving and restoring the VRs and updating the vrsave register can occur in either order. A function that does not need to address incoming arguments but does align the stack pointer dynamically can recover the address of the original stack pointer with an instruction such as lwz r11,0(sp).

The computation of `len` in the example and whether to use subfic or addi to align the stack dynamically are based on the size of the components of the frame. Starting with the components at higher addresses, the value of `len` is computed by adding the size of the floating-point register save area, the general register save area, and the VRSAVE word. The size of the alignment padding space is then computed as the smallest number of bytes needed to make `len` a multiple of 16. In the example below, the alignment padding space is 0 bytes. Consequently, subfic is used to align the stack dynamically by increasing the size of the alignment padding space by either 0 or 8 bytes. Had the alignment padding space been 8 or 12 bytes, addi is used to align the stack dynamically by decreasing the size of the alignment padding space by either 0 or 8 bytes. Continuing, the value of `len` is updated by adding the size of the vector register save area, the local variable space, the outgoing parameter list area, and 24 for the size of the link area. The size of the local variable space is adjusted so that the overall value of `len` is a multiple of 16.

The following is Apple Macintosh ABI and AIX ABI prologue and epilogue sample code.

```
function: mflr      r0                  # Save return address ...
          stw       r0,8(sp)            # ... in the caller's frame.
          bl        _savef14            # Save floating-point registers.
          stmw      r13,-220(sp)        # Save gprs in gpr save area
                                        # Save CR here if necessary
          ori       r31,sp,0            # Save pointer to incoming
                                        #       arguments
          rlwinm    r12,sp,0,28,28      # 0 or 8 based on SP alignment
          subfic    r12,r12,-len        # Add in stack length
          stwux     sp,sp,r12           # Establish new aligned frame
          mfspr     r0,vrsave           # Save VRSAVE ...
          stw       r0,-224(r31)        # ... in caller's frame.
          oris      r0,r0,0xff70        # Use v0-v10 v20-v31 and ...
          ori       r0,r0,0x0fff        # v20-v31 (for example)
          mtspr     vrsave,r0           # Update VRSAVE
          addi      r0,sp,len-224       # Compute end of VRSAVE area
          bl        _savev20            # Save VRs
                                        # Body of function
          addi      r0,sp,len-224       # Address of VRSAVE area to r0
          bl        _restv20            # Restore VRs
          lwz       r0,-224(r31)        # Fetch prior value of VRSAVE
          mtspr     vrsave,r0           # Restore Vrsave
          ori       sp,r31              # Restore SP
          lmw       r13,-220(sp)        # Restore gprs
          lwz       r0,8(sp)            # Restore return address ...
          mtlr      r0                  # ... and return from _restf14
          b         _restf14            # Restore fprs and return
```

3.3.3 Vector Register Saving and Restoring Functions

The vector register saving and restoring functions described in this section are not part of the ABI. They are defined here only to encourage uniformity among compilers in the code used to save and restore VRs.

On entry to the functions described in this section, r0 contains the address of the word just beyond the end of the vector register save area, and they leave r0 undisturbed. They modify the value of r12. The following code is an example of saving a vector register.

```
_savev20: addi          r12,r0,-192
          stvx          v20,r12,r0          # save v20
_savev21: addi          r12,r0,-176
```

```
            stvx         v21,r12,r0          # save v21
_savev22:   addi         r12,r0,-160
            stvx         v22,r12,r0          # save v22
_savev23:   addi         r12,r0,-144
            stvx         v23,r12,r0          # save v23
_savev24:   addi         r12,r0,-128
            stvx         v24,r12,r0          # save v24
_savev25:   addi         r12,r0,-112
            stvx         v25,r12,r0          # save v25
_savev26:   addi         r12,r0,-96
            stvx         v26,r12,r0          # save v26
_savev27:   addi         r12,r0,-80
            stvx         v27,r12,r0          # save v27
_savev28:   addi         r12,r0,-64
            stvx         v28,r12,r0          # save v28
_savev29:   addi         r12,r0,-48
            stvx         v29,r12,r0          # save v29
_savev30:   addi         r12,r0,-32
            stvx         v30,r12,r0          # save v30
_savev31:   addi         r12,r0,-16
            stvx         v31,r12,r0          # save v31
            blr                              # return to prologue
```

The following code shows how to restore a vector register.

```
_restv20:   addi         r12,r0,-192
            lvx          v20,r12,r0          # restore v20
_restv21:   addi         r12,r0,-176
            lvx          v21,r12,r0          # restore v21
_restv22:   addi         r12,r0,-160
            lvx          v22,r12,r0          # restore v22
_restv23:   addi         r12,r0,-144
            lvx          v23,r12,r0          # restore v23
_restv24:   addi         r12,r0,-128
            lvx          v24,r12,r0          # restore v24
_restv25:   addi         r12,r0,-112
            lvx          v25,r12,r0          # restore v25
_restv26:   addi         r12,r0,-96
            lvx          v26,r12,r0          # restore v26
_restv27:   addi         r12,r0,-80
            lvx          v27,r12,r0          # restore v27
_restv28:   addi         r12,r0,-64
            lvx          v28,r12,r0          # restore v28
_restv29:   addi         r12,r0,-48
            lvx          v29,r12,r0          # restore v29
_restv30:   addi         r12,r0,-32
            lvx          v30,r12,r0          # restore v30
_restv31:   addi         r12,r0,-16
            lvx          v31,r12,r0          # restore v31
            blr                              # return to prologue
```

3.4 Function Calls

This section applies to all user functions. Note that the intrinsic AltiVec operations are not treated as function calls, so these comments don't apply to those operations.

The first twelve vector parameters are placed in VRs v2–v13. If fewer (or no) vector type arguments are passed, the unneeded registers are not loaded and contain undefined values upon entry to the called function.

Functions that declare a vector data type as a return value place that return value in v2.

Any function that returns a vector type or has a vector parameter requires a prototype. This requirement enables the compiler to avoid shadowing VRs in GPRs.

3.4.1 SVR4 ABI and EABI Parameter Passing and Varargs

The SVR4 ABI algorithm for passing parameters considers the arguments as ordered from left (first argument) to right, although the order of evaluation of the arguments is unspecified. The vector arguments maintain their ordering. The algorithm is modified to add `vr` to contain the number of the next available vector register. In the INITIALIZE step, set `vr=2`. In the SCAN loop, add a case for the next argument `VECTOR_ARG` as follows:

- If the next argument is in the variable portion of a parameter list, set `vr=14`. This leaves the fixed portion of a variable argument list in VRs and places the variable portion in memory.
- If `vr>13` (that is, there are no more available VRs), go to OTHER. Otherwise, load the argument value into vector register `vr`, set `vr` to `vr+1`, and go to SCAN.

The OTHER case is modified only to understand that vector arguments have 16-byte size and alignment.

Aggregates are passed by reference (i.e., converted to a pointer to the object), so no change is needed to deal with 16-byte aligned aggregates.

The `va_list` type is unchanged, but an additional `va_arg_type` value of 4 named `arg_VECTOR` is defined for the `__va_arg()` interface. Since vector parameters in the variable portion of a parameter list are passed in memory, the `__va_arg()` routine can access the vector value from the `overflow_arg_area` value in the `va_list` type.

3.4.2 Apple Macintosh ABI and AIX ABI Parameter Passing without Varargs

If the function does not take a variable argument list, the non-vector parameters are passed in the same registers and stack locations as they would be if the vector parameters were not present. The only change is that aggregates and unions may be 16-byte aligned instead of 4-byte aligned. This can result in words in the parameter list being skipped for alignment (padding) and left with undefined value.

The first twelve vector parameters are placed in v2–v13. These parameters are not shadowed in GPRs. They are not allocated space in the memory argument list. Any additional vector parameters are passed through memory on the program stack. They appear together, 16-byte aligned, and after any non-vector parameters.

3.4.3 Apple Macintosh ABI and AIX ABI Parameter Passing with Varargs

The `va_list` type continues to be a pointer to the memory location of the next parameter. If `va_arg()` accesses a vector type, the `va_list` value must first be aligned to a 16-byte boundary.

A function that takes a variable argument list has all parameters, including vector parameters, mapped in the argument area as ordered and aligned according to their type. The first 8 words of the argument area are shadowed in the GPRs only if they correspond to the variable portion of the parameter list. The first parameter word is named PW0 and is at stack offset 0x24. A vector parameter must be aligned on a 16-byte boundary. This means there are two cases where vector parameters are passed in GPRs. If a vector parameter is passed in PW2:PW5 (stack offset 0x32), its value is placed in GPR5–GPR8. If a vector parameter is passed in PW6:PW9 (stack offset 0x48), its value PW6:PW7 is placed in GPR9 and GPR10 and the value PW8:PW9 is placed on the stack. All parameters after the first 8 words of the argument area that correspond to the variable portion of the parameter list are passed in memory.

In the fixed portion of the parameter list, vector parameters are placed in v2–v13, but are provided a stack location corresponding to their position in the parameter list.

3.5 malloc(), vec_malloc(), and new

In the interest of saving space, `malloc()`, `calloc()`, and `realloc()` are not required to return a 16-byte aligned address. Instead, a new set of memory management functions is introduced that return a 16-byte aligned address. The new functions are named `vec_malloc()`, `vec_calloc()`, `vec_realloc()`, and `vec_free()`. The two sets of memory management functions may not be interchanged: memory allocated with `malloc()`, `calloc()`, or `realloc()` may only be freed with `free()` and reallocated with `realloc()`; memory allocated with `vec_alloc()`, `vec_calloc()`, or `vec_realloc()` may only be freed with `vec_free()` and reallocated with `vec_realloc()`.

The user must use the appropriate set of functions based on the alignment requirement of the type involved. In the case of the C++ `operator new`, the implementation of `new` is required to use the appropriate set of functions based on the alignment requirement of the type.

3.6 setjmp() and longjmp()

The context required to be saved and restored by `setjmp()`, `longjmp()`, and related functions now includes the 12 non-volatile VRs and vrsave. The user types `sigjmp_buf` and `jmp_buf` are extended by 48 words. An unused word in the existing `jmp_buf` is used to save VRSAVE.

Table 3-3. ABI Specifications for setjmp() and longjmp()

ABI	jmp_buf Size	VRSAVE Offset	v20–v31 Offset
AIX ABI	448	100	256
Apple Macintosh ABI	448	16	256
SVR4 ABI and EABI	448	248	256

There are complications in implementing `setjmp()` and `longjmp()`:

- The user types must be enlarged. Existing applications that use these interfaces will have to be recompiled even though they make no use of the AltiVec instruction set.
- The implementation that saves and restores the VRs can only assume that the v20–v31 offset is aligned on a 4-byte boundary. A method where the VRs are saved at the first aligned location in the `jmp_buf` was rejected because the user types are only 4-byte aligned and may be copied by value to a location with different alignment.
- The implementation that saves and restores the VRs and vrsave uses instructions that do not exist on a non-AltiVec enabled PowerPC implementation. The method for testing whether the AltiVec instructions operate is privileged. One solution is to define an O/S interface that saves and restores the VRs and vrsave if and only if the AltiVec instructions exist and are enabled.

A simple solution to these complications is to define `setjmp()`, `longjmp()` and the user types `sigjmp_buf` and `jmp_buf` differently when compiled with an AltiVec-enabled compiler (i.e., when `__VEC__` is defined). These bindings result in a larger `jmp_buf` with 16-byte alignment. The bindings for `setjmp()` and `longjmp()` unconditionally save and restore the vector state. Such an implementation does not save and restore the vector state when these interfaces are compiled without an AltiVec-enabled compiler. The application must ensure that these two sets of bindings are not mixed.

3.7 Debugging Information

Extensions to the debugging information format are required to describe vector types and vector register locations. While vector types can be described as fixed length arrays of existing C types, the implementation should describe these as new fundamental types. Doing so allows a debugger to provide mechanisms to display vector values, assign vector values, and create vector literals.

This section is subject to change. It is intended to describe the extensions to the standard debugging formats: xcoff stabstrings, DWARF version 1.1.0, and DWARF version 2.0.0.

Xcoff stabstrings used in the AIX ABI and adopted by the Apple Macintosh ABI support the location of objects in GPRs and FPRs. The stabstring code “R” describes a parameter passed by value in the given GPR; “r” describes a local variable residing in the given GPR. The stabstring code “X” describes a parameter passed by value in the given vector register; “x” describes a local variable residing in the given vector register.

DWARF 2.0 debugging DIEs support the location of objects in any machine register. The SVR4 ABI specifies the DWARF register number mapping. The VRs `v0–v31` are assigned register numbers 1124–1155. The VRSAVE SPR is SPR256 and is assigned the register number 356.

3.8 printf() and scanf() Control Strings

The conversion specifications in control strings for input functions (`fscanf`, `scanf`, `sscanf`) and output functions (`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `vsprintf`) are extended to support vector types.

3.8.1 Output Conversion Specifications

The output conversion specifications have the following general form:

```
%[<flags>][<width>][<precision>][<size>]<conversion>
```

where,

```
<flags>            ::=<flag-char> | <flags><flag-char>
<flag-char>        ::=<std-flag-char> | <c-sep>
<std-flag-char>    ::=   '-' | '+' | '0' | '#' | ' '
<c-sep>            ::=   ',' | ';' | ':' | '_'
<width>            ::=   <decimal-integer> | '*'
<precision>        ::=   '.' <width>
<size>             ::=   'll' | 'L' | 'l' | 'h' | <vector-size>
<vector-size>      ::=   'vl' | 'vh' | 'lv' | 'hv' | 'v'
<conversion>       ::=   <char-conv> | <str-conv> | <fp-conv> |
                         <int-conv> | <misc-conv>
<char-conv>        ::=   'c'
<str-conv>         ::=   's' | 'P'
<fp-conv>          ::=   'e' | 'E' | 'f' | 'g' | 'G'
<int-conv>         ::=   'd' | 'i' | 'u' | 'o' | 'p' | 'x' | 'X'
<misc-conv>        ::=   'n' | '%'
```

The extensions to the output conversion specification for vector types are shown in bold.

The `<vector-size>` indicates that a single vector value is to be converted. The vector value is displayed in the following general form:

$$value_1\ C\ value_2\ C\ \ldots\ C\ value_n$$

where C is a separator character defined by `<c-sep>` and there are 4, 8, or 16 output values depending on the `<vector-size>` each formatted according to the `<conversion>`, as follows:

- A `<vector-size>` of ‘vl’ or ‘lv’ consumes one argument and modifies the `<int-conv>` conversion; it should be of type `vector signed int`, `vector unsigned int`, or `vector bool int`; it is treated as a series of four 4-byte components.
- A `<vector-size>` of ‘vh’ or ‘hv’ consumes one argument and modifies the `<int-conv>` conversion; it should be of type `vector signed short`, `vector unsigned short`, `vector bool short`, or `vector pixel`; it is treated as a series of eight 2-byte components.
- A `<vector-size>` of ‘v’ with `<int-conv>` or `<char-conv>` consumes one argument; it should be of type `vector signed char`, `vector unsigned char`, or `vector bool char`; it is treated as a series of sixteen 1-byte components.
- A `<vector-size>` of ‘v’ with `<fp-conv>` consumes one argument; it should be of type `vector float`; it is treated as a series of four 4-byte floating-point components.
- All other combinations of `<vector-size>` and `<conversion>` are undefined.

The default value for the separator character is a space unless the `‘c’` conversion is being used. For the `‘c’` conversion the default separator character is null. Only one separator character may be specified in `<flags>`.

Examples:

```
vector signed char s8 = vector signed char(‘a’,‘b’,‘ ’,‘d’,‘e’,‘f’,
                                           ‘g’,‘h’,‘i’,‘j’,‘k’,‘l’,
                                           ‘m,’,‘,’,’o’,’p’);
vector unsigned short u16 = vector unsigned short(1,2,3,4,5,6,7,8);
vector signed int s32 = vector signed int(1, 2, 3, 99);
vector float f32 = vector float(1.1, 2.2, 3.3, 4.39501);
printf(“s8  = %vc\n”, s8);
printf(“s8  = %,vc\n”, s8);
printf(“u16 = %vhu\n”, u16);
printf(“s32 = %,2lvd\n”, s32);
printf(“f32 = %,5.2vf\n”, f32);
```

This code produces the following output:

```
s8  = ab defghijklm,op
s8  = a,b, ,d,e,f,g,h,i,j,k,l,m,,,o,p
u16 = 1 2 3 4 5 6 7 8
s32 =  1, 2, 3,99
f32 =  1.10 ,2.20 ,3.30 ,4.40
```

3.8.2 Input Conversion Specifications

The input conversion specifications have the following general form:

```
%[<flags>][<width>][<size>]<conversion>
```

where,

```
<flags>        ::=     '*' | <c-sep> ['*'] | ['*'] <c-sep>
<c-sep>        ::=     ',' | ';' | ':' | '_'
<width>        ::=     <decimal-integer>
<size>         ::=     'll' | 'L' | 'l' | 'h' | <vector-size>
<vector-size>  ::=     'vl' | 'vh' | 'lv' | 'hv' | 'v'
<conversion>   ::=     <char-conv> | <str-conv> | <fp-conv> |
                       <int-conv> | <misc-conv>
<char-conv>    ::=     'c'
<str-conv>     ::=     's' | 'P'
<fp-conv>      ::=     'e' | 'E' | 'f' | 'g' | 'G'
<int-conv>     ::=     'd' | 'i' | 'u' | 'o' | 'p' | 'x' | 'X'
<misc-conv>    ::=     'n' | '%' | '['
```

The extensions to the input conversion specification for vector types are shown in bold.

The <vector-size> indicates that a single vector value is to be scanned and converted. The vector value to be scanned is in the following general form:

value$_1$ C value$_2$ C ... C value$_n$

where C is a separator sequence defined by <c-sep> (the separator character optionally preceded by whitespace characters) and 4, 8, or 16 values are scanned depending on the <vector-size> each value scanned according to the <conversion>, as follows:

- A <vector-size> of 'vl' or 'lv' consumes one argument and modifies the <int-conv> conversion; it should be of type `vector signed int *` or `vector unsigned int *` depending on the <int-conv> specification; four values are scanned.
- A <vector-size> of 'vh' or 'hv' consumes one argument and modifies the <int-conv> conversion; it should be of type `vector signed *` or `vector unsigned short *` depending on the <int-conv> specification; 8 values are scanned.
- A <vector-size> of 'v' with <int-conv> or <char-conv> consumes one argument; it should be of type `vector signed char *` or `vector unsigned char *` depending on the <int-conv> or <char-conv> specification; 16 values are scanned.
- A <vector-size> of 'v' with <fp-conv> consumes one argument; it should be of type `vector float *`; four floating-point values are scanned.
- All other combinations of <vector-size> and <conversion> are undefined.

For the `'c'` conversion the default separator character is `null`, and the separator sequence does not include whitespace characters preceding the separator character. For other than the `'c'` conversions, the default separator character is a space, and the separator sequence does include whitespace characters preceding the separator character.

If the input stream reaches end-of-file or there is a conflict between the control string and a character read from the input stream, the input functions return `EOF` and do not assign to their vector argument.

When a conflict occurs, the character causing the conflict remains unread and is processed by the next input operation.

Examples:

```
sscanf("ab defghijklm,op", "%vc", &s8);
sscanf("a,b, ,d,e,f,g,h,i,j,k,l,m,,,o,p", "%,vc", &s8);
sscanf("1 2 3 4 5 6 7 8", "%vhu", &u16);
sscanf("1, 2, 3,99", "%,2lvd", &s32);
sscanf("1.10 ,2.20 ,3.30 ,4.40" ,"%,5vf" ,&f32);
```

This is equivalent to:

```
vector signed char s8 = vector signed char('a','b',' ','d','e','f',
                                            'g','h','i','j','k','l',
                                            'm',',','o','p');
vector unsigned short u16 = vector unsigned short(1,2,3,4,5,6,7,8);
vector signed int s32 = vector signed int(1, 2, 3, 99);
vector float f32 = vector float(1.1, 2.2, 3.3, 4.4);
```

Chapter 4
AltiVec Operations and Predicates

The following three subsections provide some background information that is helpful in understanding the descriptions provided for each operation and predicate. This is followed by a detailed listing of AltiVec operations followed by a separate section describing the AltiVec predicates. The final subsection contains compiler notes for handling predicates.

4.1 Vector Status and Control Register

The vector status and control register (VSCR) is a special 32-bit vector register shown in Figure 4-1.

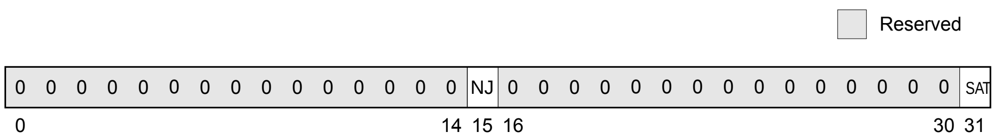


Figure 4-1. Vector Status and Control Register (VSCR)

The VSCR has two defined bits, the AltiVec non-Java mode (NJ) bit (VSCR[15]) and the AltiVec saturation (SAT) bit (VSCR[31]); the remaining bits are reserved. The vec_mfvscr operation moves the VSCR to a vector register. When moved, the 32-bit VSCR is right-justified in the 128-bit vector register, and the upper 96 bits VR*x*[0–95] of the vector register are cleared, so the VSCR in a vector register looks as shown in Figure 4-2.

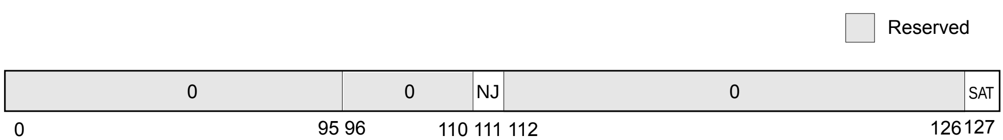


Figure 4-2. VSCR Moved to a Vector Register

VSCR bit settings are shown in Table 4-1.

Table 4-1. VSCR Field Descriptions

Bits	Name	Description
0–14	—	Reserved. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.
15	NJ	Non-Java. A mode control bit that determines whether AltiVec floating-point operations will be performed in a Java-IEEE-C9X–compliant mode or a possibly faster non-Java/non-IEEE mode. 0 The Java-IEEE-C9X–compliant mode is selected. Denormalized values are handled as specified by Java, IEEE, and C9X standard. 1 The non-Java/non-IEEE–compliant mode is selected. If an element in a source vector register contains a denormalized value, the value 0 is used instead. If an instruction causes an underflow exception, the corresponding element in the target VR is cleared to 0. In both cases the 0 has the same sign as the denormalized or underflowing value. This mode is described in detail in the AltiVec Programming Environments Manual.
16–30	—	Reserved. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.
31	SAT	Saturation. A sticky status bit indicating that some field in a saturating instruction saturated since the last time SAT was cleared. In other words, when SAT = 1 it remains set until it is cleared by an explicit instruction. 0 Indicates no saturation occurred, an instruction can explicitly clear this bit. 1 The AltiVec saturate instruction implicitly sets the SAT field when saturation has occurred on the results one of the AltiVec instructions or vector operations having saturate in its name.

After `vec_mfvscr` executes, the result in the target vector register is architecturally precise. That is, it reflects all updates to the SAT bit that could have been made by vector instructions logically preceding it in the program flow, and further, it does not reflect any SAT updates that may be made to it by vector instructions logically following it in the program flow. Reading the VSCR can be much slower than typical AltiVec instructions, and therefore care must be taken in reading it to avoid performance problems.

The first six 16-bit elements of the result are 0. The seventh element of the result contains the high-order 16 bits of the VSCR (including NJ). The eighth element of the result contains the low-order 16 bits of the VSCR (including SAT).

The setting of the Non-Java mode (NJ) bit (VSCR[15]) affects some vector floating-point operations. The other special bit (VSCR[31]) is the AltiVec Saturation (SAT) bit that is set when an operation generates a saturated result. Saturation is defined with respect to the type of resulting element The result d of saturating a value x with respect to a type t means:

```
d = max (minimum(t), min(maximum(t), x))
```

```
where minimum(t) is the algebraically smallest value representable by a number of
type t and maximum(t) is the algebraically largest value by a number of type t.
```

For each operation, where applicable, the effects of the NJ bit setting and/or the effects on the SAT bit are described in the operation description.

4.2 Byte Ordering

The default mapping for AltiVec ISA is PowerPC big-endian. The endian support of the PowerPC architecture does not address any data element larger than a double word; the basic memory unit for vectors is a quad word. Big-endian byte ordering is shown in Figure 4-3.

<table>
<tr><td colspan="16">Quad Word</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="4">High-Order Word 0</td><td colspan="4">Word 1</td><td colspan="4">Word 2</td><td colspan="4">Low-Order Word 3</td></tr>
<tr><td colspan="2">High-Order Half Word for Word 0</td><td colspan="2">Low-Order Half Word for Word 0</td><td colspan="4"></td><td colspan="4"></td><td colspan="4"></td></tr>
<tr><td colspan="2">High-Order Half Word</td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2">Low-Order Half Word</td></tr>
<tr><td colspan="2">Half Word 0</td><td colspan="2">Half Word 1</td><td colspan="2">Half Word 2</td><td colspan="2">Half Word 3</td><td colspan="2">Half Word 4</td><td colspan="2">Half Word 5</td><td colspan="2">Half Word 6</td><td colspan="2">Half Word 7</td></tr>
<tr><td>High-Order Byte</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>Low-Order Byte</td></tr>
<tr><td>Byte 0</td><td>Byte 1</td><td>Byte 2</td><td>Byte 3</td><td>Byte 4</td><td>Byte 5</td><td>Byte 6</td><td>Byte 7</td><td>Byte 8</td><td>Byte 9</td><td>Byte 10</td><td>Byte 11</td><td>Byte 12</td><td>Byte 13</td><td>Byte 14</td><td>Byte 15</td></tr>
<tr><td>0</td><td>8</td><td>16</td><td>24</td><td>32</td><td>40</td><td>48</td><td>56</td><td>64</td><td>72</td><td>80</td><td>88</td><td>96</td><td>104</td><td>112</td><td>120 127</td></tr>
<tr><td>↑
MSB
(High-Order)</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>↑
LSB
(Low-Order)</td></tr>
</table>

Figure 4-3. Big-Endian Byte Ordering for a Vector Register

As shown in Figure 4-3, the vector register elements are numbered using big-endian byte ordering. For example, the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15.

When defining high-order and low-order for elements in a vector register, be careful not to confuse its meaning based on the bit numbering. For example, in Figure 4-3 the high-order half word for word 0 would be half word 0 (bits 0–7), and the low-order half word for word 0 would be half word 1 (bits 8–15).

4.3 Notation and Conventions

Operation and predicate functionality is described in this section by a semiformal pseudocode language. Table 4-2 lists the pseudocode notation and conventions used throughout the section.

Table 4-2. Notation and Conventions

Notation/Convention	Meaning
$\leftarrow$	Assignment
$+, +_{fp}$	Add, single-precision floating-point add
$-, -_{fp}$	Subtract, single-precision floating-point subtract
$*, *_{fp}$	Multiply, single-precision floating-point multiply
/	Integer division with non-negative remainder
$<, <_{fp}$	Less than, single-precision floating-point less than
$\leq, \leq_{fp}$	Less than or equal, single-precision floating-point less than or equal
$>, >_{fp}$	Greater than, single-precision floating-point greater than
$\geq, \geq_{fp}$	Greater than or equal, single-precision floating-point greater than or equal
$!=, !=_{fp}$	Not equal, floating-point not equal
$=, =_{fp}$	Equal, floating-point equal
$+\infty, -\infty$	Positive infinity, negative infinity
$\|$	Concatenation of two bit strings (e.g., $010 \| 111$ is the same as 010111)
&	AND bit-wise operator
$\vert$	OR bit-wise operator
⊕	Exclusive-OR bit-wise operator
¬	NOT logical operator (one’s complement)
0bnnnn	A number expressed in binary format
0xnnnn	A number expressed in hexadecimal format
a,b,c,d	These symbols represent whole operands in an AltiVec operation or predicate. This is typically a vector, but in some operations it can represent a specific length literal value.
a_i,b_i,c_i,d_i	These symbols represent the i^{th} component elements of a vector a, b, c, or d, respectively.
ABS(x)	Absolute value of x
BorrowOut(x - y)	Borrow out of the difference of x and y
BoundAlign(x,y)	Align x to a y-byte boundary.
CarryOut(x + y)	Carry out of the sum of x and y
Ceil(x)	The smallest single-precision floating-point integer that is greater than or equal to x
do i=x to y	Do loop. • Do the following starting at x and iterating to y • Indenting shows range. • “To” and/or “by” clauses specify incrementing an iteration variable. • “While” clauses give termination conditions.
end	Indicates the end of a do loop

Table 4-2. Notation and Conventions (Continued)

Notation/Convention	Meaning
Floor(x)	The largest single-precision floating-point integer that is less than or equal to x
FP2^xEst(x)	3-bit-accurate floating-point estimate of 2**x
FPLog$_2$Est(x)	3-bit-accurate floating-point estimate of log2(x)
FPRecipEst(x)	12-bit-accurate floating-point estimate of 1/x
if...then...else...	Conditional execution, indenting shows range, else is optional.
ISNaN(x)	Result is 1 if x is a not a number (NaN) and 0 is x is a number
ISNUM(x)	Result is 1 if x is a number and 0 is x is not a number (NaN)
MAX(x,y)	Returns the larger of x or y. For floating-point values, the following applies: • the maximum of +0.0 and –0.0 is +0.0 • the maximum of any value and a NaN is a QNaN
MEM(x,y)	Value at memory location x of size y bytes
MIN(x,y)	Returns the smaller of x or y. For floating-point values, the following applies: • the minimum of +0.0 and –0.0 is –0.0 • the minimum of any value and a NaN is a QNaN
mod(x,y)	Remainder of x/y
NaN	Not a Number, non-numeric
NEG(x)	Result is -x
NGE(x,y)	Result is 1 if x or y is a NaN or if x < y, and 0 otherwise
NGT(x,y)	Result is 1 if x or y is a NaN or x ≤ y, and 0 oherwise
NLE(x,y)	Result is 1 if x or y is a NaN or x > y, and 0 otherwise
NLT(x,y)	Result is 1 if x or y is a NaN or x ≥ y, and 0 otherwise
QNaN	NaN that propagates through most arithmetic operations without signalling an exception
RecipSQRTEst(x)	Result is a 12-bit accurate single-precision floating-point estimate of the reciprocal of the square root of x
RndToFPINear(x)	The single-precision floating-point integer that is nearest in value to x (in case of a tie, the even single-precision floating-point value is used).
RndToFPITrunc(x)	The largest single-precision floating-point integer that is less than or equal to x if x≥0, or the smallest single-precision floating-point integer that is greater than or equal to x if x<0
RndToFPNearest(x)	IEEE rounding to nearest floating-point number
ROTL(x,y)	Result of rotating x left by y bits
S	Represents a propagated sign bit in a figure
Saturate(x)	y ← Saturate(x) means saturate x to the type of y
ShiftRight(x,y) ShiftLeft(x,y)	Shift the contents of x right or left y bits, clearing vacated bits (logical shift). This operation is used for shift instructions.
ShiftRightA(x,y)	Shift the contents of x right y bits, copying the sign bit to the vacated bits (algebraic shift)
SignExtend(x,y)	Sign-extend x on the left with sign bits (that is, with copies of bit 0 of x) to produce y-bit value; represented in figures by a single S
SIToFP(x,y)	Result of converting the signed integer x to a y-bit floating-point value using Round-to-Nearest mode

Table 4-2. Notation and Conventions (Continued)

Notation/Convention	Meaning
UIToUImod(x,y)	Truncate an unsigned integer x to y-bit unsigned integer
Undefined	An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.
x_i	The i^{th} element of vector x where the size and type of the element are determined by the type of x
x{i}	The i^{th} byte of vector x
x[y:x]	Bits i through j of vector x, where i can equal j if referring to a single bit
${}^{x}0$	A bit string of x zeros
${}^{x}1$	A bit string of x ones
${}^{x}y$	A bit string of x copies of y, for example, ${}^{3}1$ = 111
x^n	x raised to the nth power

Precedence rules for pseudocode operators are summarized in Table 4-3.

Table 4-3. Precedence Rules

Operators	Associativity
x{i}, x[y], x[y:z] function evaluation	Left to right
${}^{x}y$ or replication, x^y or exponentiation	Right to left
unary –, ¬	Right to left
*, $*_{fp,}$ /	Left to right
+, $+_{fp}$, –, $-_{fp}$	Left to right
||	Left to right
=, $=_{fp,}$!=,$!=_{fp}$, <, $<_{fp}$, ≤, $\leq_{fp}$, >, $>_{fp}$, ≥, $\geq_{fp}$	Left to right
&, ⊕	Left to right
|	Left to right
←	None

Operators higher in Table 4-3 are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. For example, '–' (unary minus) associates from left to right, so a – b – c = (a – b) – c. Parentheses are used to override the evaluation order implied by Table 4-3, or to increase clarity; parenthesized expressions are evaluated before serving as operands.

4.4 Generic and Specific AltiVec Operations

The AltiVec operations are organized alphabetically by generic operation name with a definition of the permitted generic and specific AltiVec operations. The operations are listed in alphabetical order by mnemonic. Figure 4-4 shows the format for each operation description page.

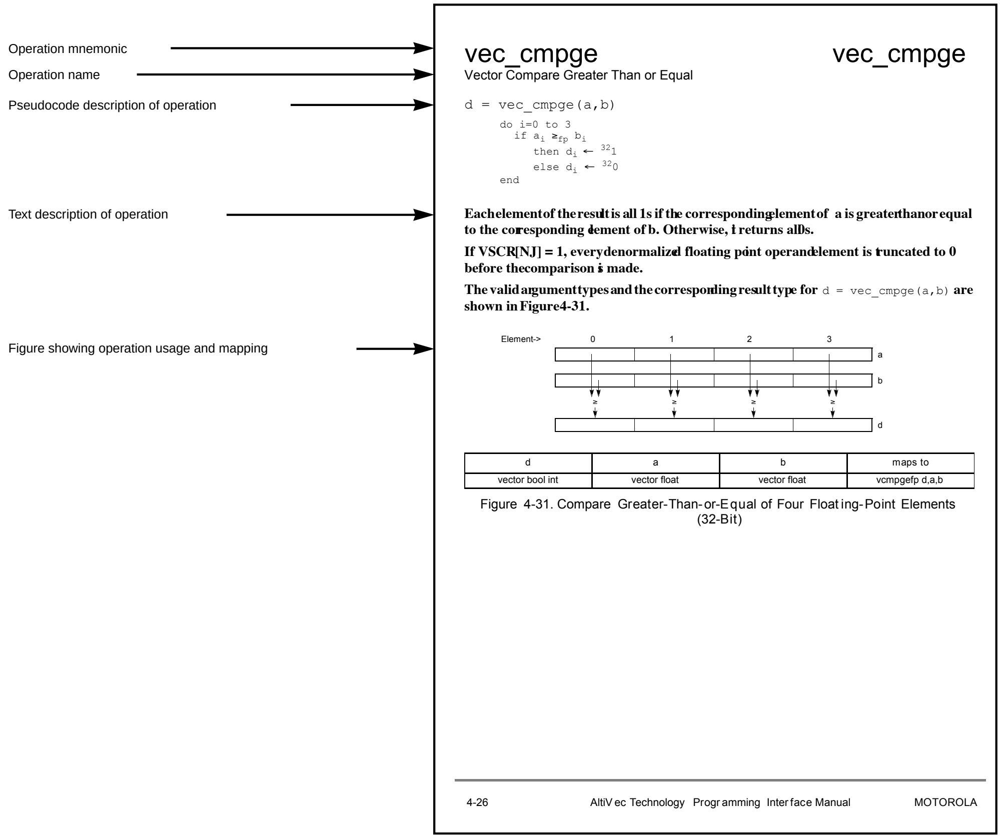


Figure 4-4. Operation Description Format

Where possible, each description is supported by reference figures indicating data modifications and including a table that lists:

- the valid set of argument types for that generic AltiVec operation,
- the result type for each set of argument types, and
- the specific AltiVec instruction(s) generated for that set of arguments.

Any operation not explicitly permitted in this section is prohibited.

vec_abs

vec_abs

Vector Absolute Value

d = vec_abs(**a**)

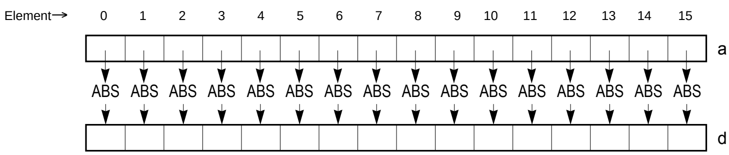
```
n ← number of elements
do i=0 to n-1
d_i ← ABS(a_i)
end
```

Each element of the result is the absolute value of the corresponding element of a. The arithmetic is modular for integer types.

For `vector float` argument types, the operation is independent of VSCR[NJ].

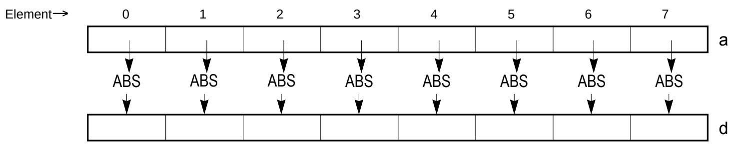
Programming note: Unlike other operations, `vec_abs` maps to multiple instructions. The programmer should consider alternatives. For example, to compute the absolute difference of two vectors a and b, the expression `vec_abs(vec_sub(a,b))` expands to four instructions. A simpler method uses the expression `vec_sub(vec_max(a,b), vec_min(a,b))` that expands to three instructions.

The valid combinations of argument types and the corresponding result types for `d = vec_abs(a)` are shown in Figure 4-5, Figure 4-6, Figure 4-7, and Figure 4-8. It is necessary to use the generic name since there is no specific operation for `vec_abs`.



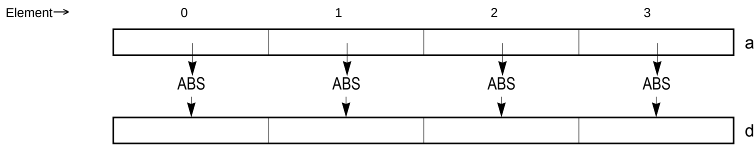
d	a	maps to
vector signed char	vector signed char	vspltisb z,0 vsububm t,z,a vmaxsb d,a,t

Figure 4-5. Absolute Value of Sixteen Integer Elements (8-bit)



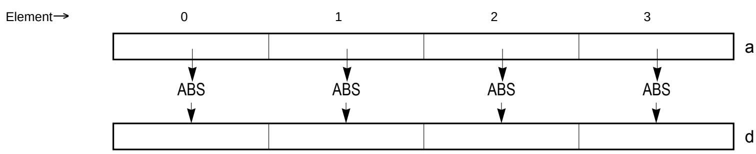
d	a	maps to
vector signed short	vector signed short	vspltisb z,0 vsubuhm t,z,a vmaxsh d,a,t

Figure 4-6. Absolute Value of Eight Integer Elements (16-bit)



d	a	maps to
vector signed int	vector signed int	vsplisb z,0 vsubuwm t,z,a vmaxsw d,a,t

Figure 4-7. Absolute Value of Four Integer Elements (32-bit)



d	a	maps to
vector float	vector float	vspltisw m,-1 vslw t,m,m vandc d,a,t

Figure 4-8. Absolute Value of Four Floating-Point Elements (32-bit)

vec_abss vec_abss

Vector Absolute Value Saturated

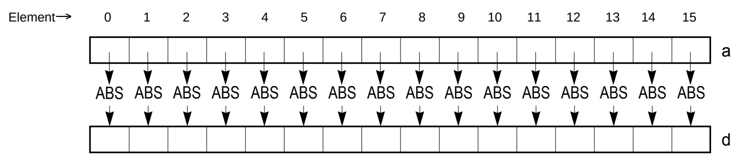
d = vec_abss(**a**)

```
n ← number of elements
do i=0 to n-1
d_i ← Saturate(ABS(a_i))
end
```

Each element of the result is the absolute value of the corresponding element of a. The arithmetic is saturated for integer types. If saturation occurs, VSCR[SAT] is set (see Table 4-1).

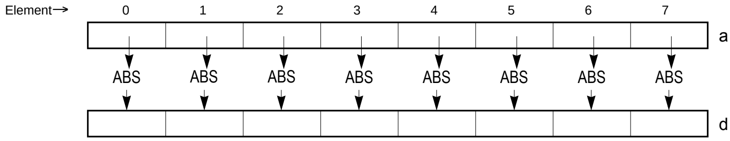
Programming note: Unlike other operations, vec_abss maps to multiple instructions. The programmer should consider alternatives. For example, to compute the absolute difference of two vectors a and b, the expression `vec_abss(vec_subs(a,b))` expands to four instructions. A simpler method uses the expression `vec_subs(vec_max(a,b),vec_min(a,b))` that expands to three instructions.

The valid combinations of argument types and the corresponding result types for `d = vec_abss(a)` are shown in Figure 4-9, Figure 4-10, and Figure 4-11. It is necessary to use the generic name since there is no specific operation for `vec_abss`.



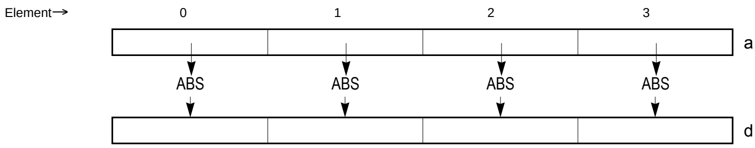
d	a	maps to
vector signed char	vector signed char	vspltisb z,0 vsubsbs t,z,a vmaxsb d,a,t

Figure 4-9. Saturated Absolute Value of Sixteen Integer Elements (8-bit)



d	a	maps to
vector signed short	vector signed short	vspltisb z,0 vsubshs t,z,a vmaxsh d,a,t

Figure 4-10. Saturated Absolute Value of Eight Integer Elements (16-bit)



d	a	maps to
vector signed int	vector signed int	vsplisb z,0 vsubsws t,z,a vmaxsw d,a,t

Figure 4-11. Saturated Absolute Value of Four Integer Elements (32-bit)

vec_add

vec_add

Vector Add

d = vec_add(**a**,**b**)

- Integer add:

```
n ← number of elements
do i=0 to n-1
d_i ← a_i + b_i
end
```

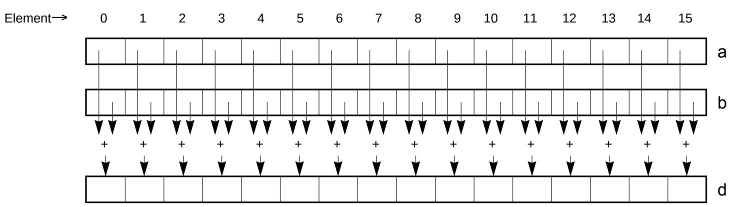
- Floating-point add:

```
do i=0 to 3
d_i ← a_i +_fp b_i
end
```

Each element of a is added to the corresponding element of b. Each sum is placed in the corresponding element of d.

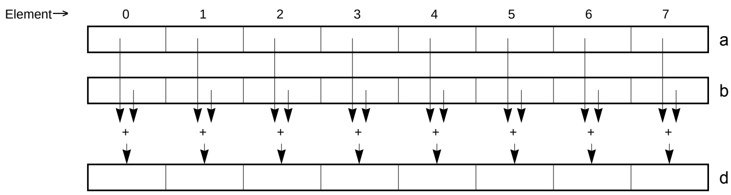
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element is truncated to a 0 of the same sign.

The valid combinations of argument types and the corresponding result types for **d** = vec_add(**a**,**b**) are shown in Figure 4-12, Figure 4-13, Figure 4-14, and Figure 4-15.



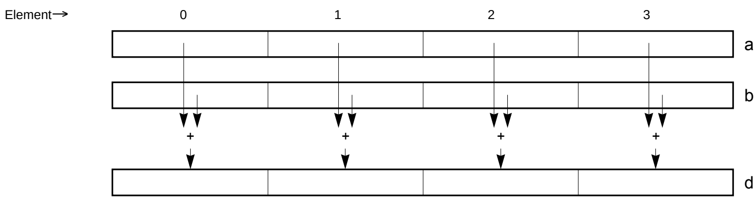
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vaddubm d,a,b
vector unsigned char	vector unsigned char	vector bool char	vaddubm d,a,b
vector unsigned char	vector bool char	vector unsigned char	vaddubm d,a,b
vector signed char	vector signed char	vector signed char	vaddubm d,a,b
vector signed char	vector signed char	vector bool char	vaddubm d,a,b
vector signed char	vector bool char	vector signed char	vaddubm d,a,b

Figure 4-12. Add Sixteen Integer Elements (8-bit)



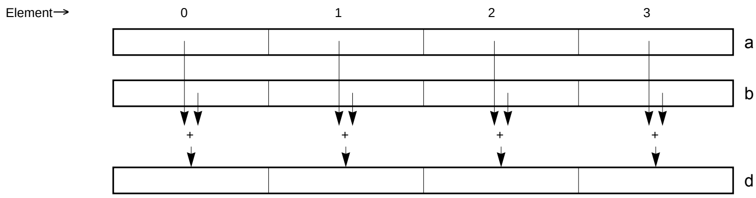
d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vadduhm d,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-13. Add Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vadduwm d,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-14. Add Four Integer Elements (32-bit)



d	a	b	maps to
vector float	vector float	vector float	vaddfp d,a,b

Figure 4-15. Add Four Floating-Point Elements (32-bit)

vec_addc

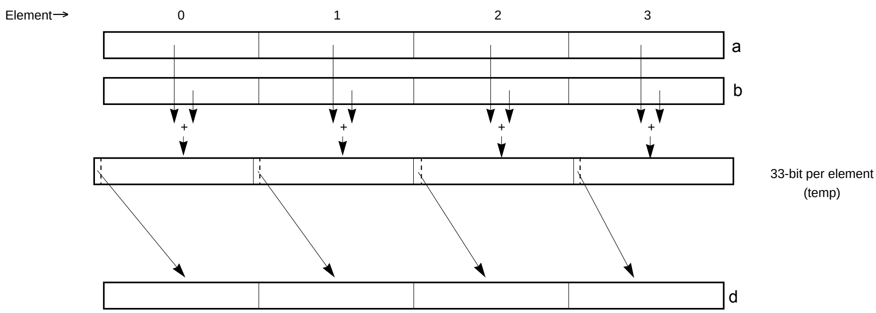
vec_addc

Vector Add Carryout Unsigned Word

d = vec_addc(**a**,**b**)

```
do i=0 to 3
di = CarryOut(ai + bi)
end
```

Each element of a is added to the corresponding element in b. The carry from each sum is zero-extended and placed into the corresponding element of d. CarryOut (a + b) is 1 if there is a carry, and otherwise 0. The valid argument types and the corresponding result type for `d = vec_addc(a,b)` are shown in Figure 4-16.



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vaddcuw d,a,b

Figure 4-16. Carryout of Four Unsigned Integer Adds (32-bit)

vec_adds

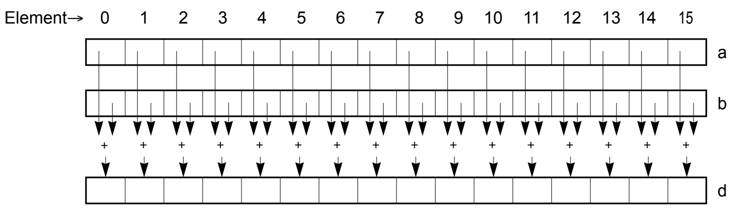
vec_adds

Vector Add Saturated

d = vec_adds(**a**,**b**)

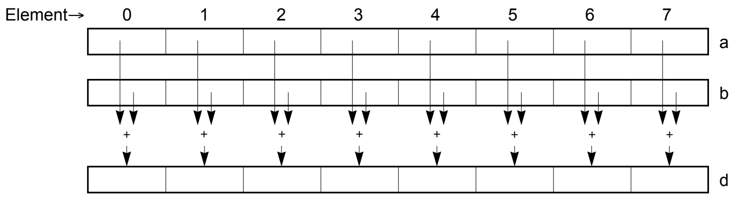
```
n ← number of elements
do i=0 to n-1
d_i ← Saturate(a_i + b_i)
end
```

Each element of a is added to the corresponding element of b. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The signed-integer result is placed into the corresponding element of d. The valid combinations of argument types and the corresponding result types for **`d`** `= vec_adds(`**`a,b`**`)` are shown in Figure 4-17, Figure 4-18, and Figure 4-19.



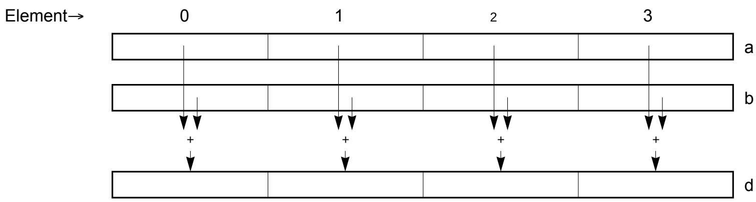
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vaddubs d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	vaddsbs d,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-17. Add Saturating Sixteen Integer Elements (8-bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vadduhs d,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	vaddshs d,a,b
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-18. Add Saturating Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vadduws d,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	vaddsws d,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-19. Add Saturating Four Integer Elements (32-bit)

vec_and

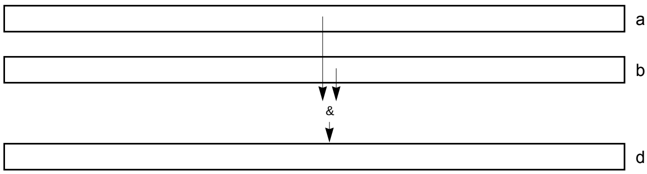
vec_and

Vector Logical AND

d = vec_and(**a**,**b**)

d ← **a** & **b**

Each bit of the result is the logical AND of the corresponding bits of a and b. The valid combinations of argument types and the corresponding result types for **d** = vec_and(**a**,**b**) are shown in Figure 4-20.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vand d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector signed char	
vector bool char	vector bool char	vector bool char	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector signed short	
vector bool short	vector bool short	vector bool short	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector signed int	
vector bool int	vector bool int	vector bool int	
vector float	vector bool int	vector float	
	vector float	vector bool int	
	vector float	vector float	

Figure 4-20. Logical Bit-Wise AND

vec_andc

vec_andc

Vector Logical AND with Complement

d = vec_andc(**a**,**b**)

d ← **a** & ¬**b**

Each bit of the result is the logical AND of the corresponding bit of a and the one's complement of the corresponding bit of b. the valid combinations of argument types and the corresponding result types for **d** = `vec_andc(`**a**`,`**b**`)` are shown in Figure 4-21.

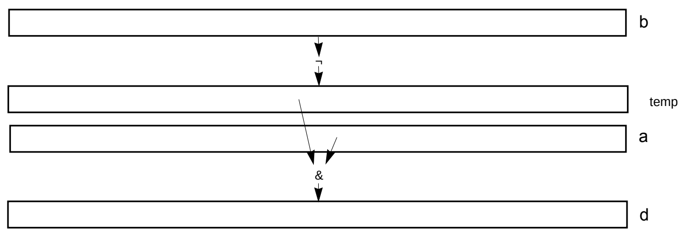


Figure 4-21. Logical Bit-Wise AND with Complement

d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vandc d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector signed char	
vector bool char	vector bool char	vector bool char	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector signed short	
vector bool short	vector bool short	vector bool short	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector signed int	
vector bool int	vector bool int	vector bool int	
vector float	vector bool int	vector float	
	vector float	vector bool int	
	vector float	vector float	

Figure 4-21. Logical Bit-Wise AND with Complement

vec_avg

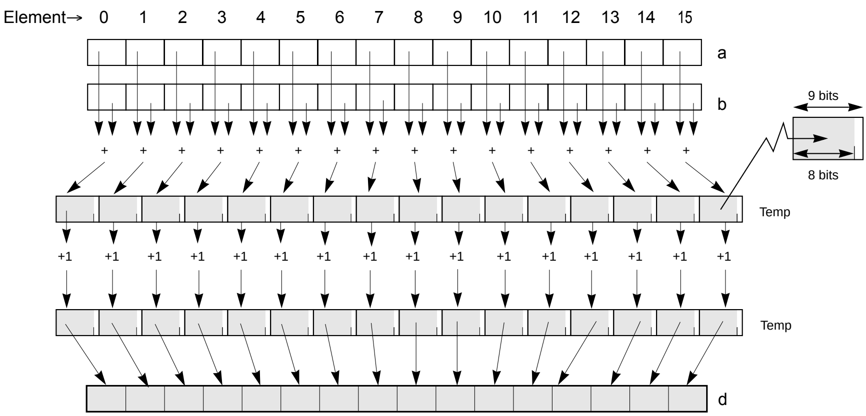
vec_avg

Vector Average

d = vec_avg(**a**,**b**)

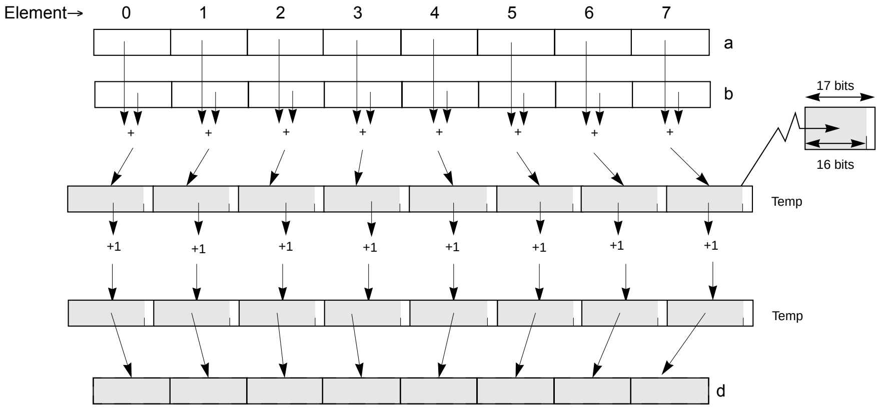
```
n ← number of elements
do i=0 to n-1
d_i ← (a_i + b_i + 1) / 2
end
```

Each element of the result is a rounded average of the corresponding elements of a and b. Intermediate calculations are not limited by the element size. The value 1 is added to the sum of elements in a and b to ensure the result is rounded up. The valid combinations of argument types and the corresponding result types for `d = vec_avg(a,b)` are shown in Figure 4-22, Figure 4-23, and Figure 4-24.



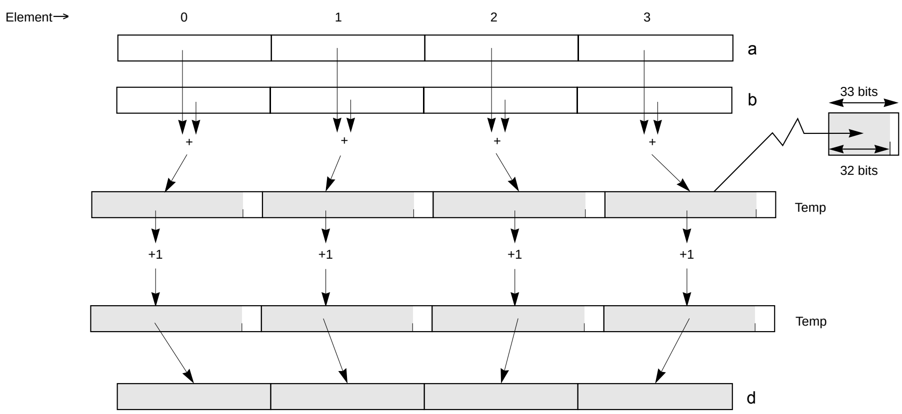
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vavgub d,a,b
vector signed char	vector signed char	vector signed char	vavgsb d,a,b

Figure 4-22. Average Sixteen Integer Elements (8-bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vavguh d,a,b
vector signed short	vector signed short	vector signed short	vavgsh d,a,b

Figure 4-23. Average Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vavguw d,a,b
vector signed int	vector signed int	vector signed int	vavgsw d,a,b

Figure 4-24. Average Four Integer Elements (32-bit)

vec_ceil

vec_ceil

Vector Ceiling

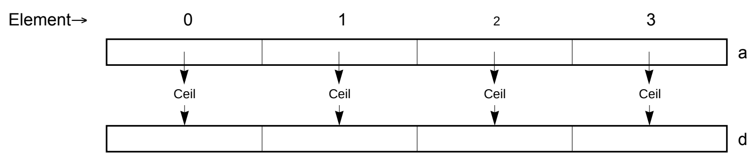
d = vec_ceil(**a**)

```
do i=0 to 3
di ← Ceil(ai)
end
```

Each single-precision floating-point element in a is rounded to a single-precision floating-point integer using the rounding mode Round toward +Infinity, and placed into the corresponding word element of d. If an element a_i is infinite, the corresponding element d_i equals a_i. If an element a_i is finite, the corresponding element d_i is the smallest represented floating-point value ≥ a_i. For example, if the floating-point element was 123.45, the resulting integer would be 124.

If VSCR[NJ] = 1, every denormalized operand element is truncated to 0 before the operation.

The valid argument types and the corresponding result type for `d = vec_ceil(a,b)` are shown in Figure 4-25.



d	a	maps to
vector float	vector float	vrfip d,a

Figure 4-25. Round to Plus Infinity of Four Floating-Point Integer Elements (32-Bit)

vec_cmpb

vec_cmpb

Vector Compare Bounds Floating-Point

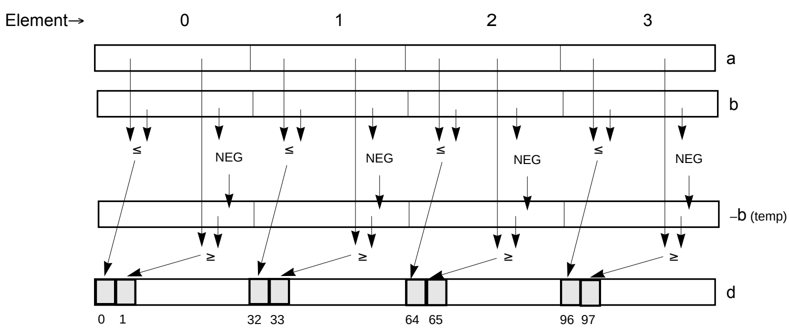
d = vec_cmpb(**a**,**b**)

```
do i=0 to 3
d_i ← 0
if a_i ≤fp b_i
        then d_i[0] ← 0
        else d_i[0] ← 1
if a_i ≥fp -b_i
        then d_i[1] ← 0
        else d_i[1] ← 1
end
```

Each element in a is compared to the corresponding element in b. The 2-bit result indicates whether the element in a is within the bounds specified by the element in b. Bit 0 of each result is 0 if the element in a is less than or equal to the element in b (i.e., in bounds high), and is 1 otherwise (i.e., out of bounds high). Bit 1 of the 2-bit value is 0 if the element in a is greater than or equal to the negative of the element in b (i.e., in bounds low), and is 1 otherwise (i.e., out of bounds low). The 2-bit result is placed into the high-order two bits (bit 0 and 1) of the corresponding element in d (which correspond to bits 0–1, 32–33, 64–65, and 96–97 of d, respectively) and the remaining bits are cleared. If any single-precision floating-point word element in b is negative; the corresponding element in a is out of bounds. If an element in a or b element is a NaN, the two high-order bits of the corresponding result are both 1.

If VSCR[NJ] = 1, every denormalized operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for **`d`** `= vec_cmpb(`**`a,b`**`)` are shown in Figure 4-26.



d	a	b	maps to
vector signed int	vector float	vector float	vcmpbfp d,a,b

Figure 4-26. Compare Bounds of Four Floating-Point Elements (32-Bit)

vec_cmpeq vec_cmpeq

Vector Compare Equal

d = vec_cmpeq(**a**,**b**)

- Integer compare equal:

```
n ← number of elements
m ← number of bits in an element (128/n)
do i=0 to n-1
if a_i = b_i
        then d_i ← ^m1
        else d_i ← ^m0
end
```

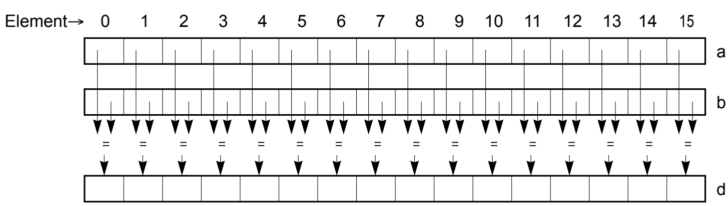
- Floating-point compare equal:

```
do i=0 to 3
if a_i =_fp b_i
        then d_i ← ^32 1
        else d_i ← ^32 0
end
```

Each element of the result is all ones if the corresponding element of a is equal to the corresponding element of b. Otherwise, it returns all zeros.

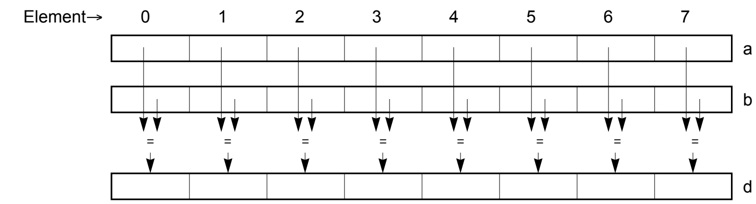
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result types for **`d = vec_cmpeq(a,b)`** are shown in Figure 4-27, Figure 4-28, Figure 4-29, and Figure 4-30.



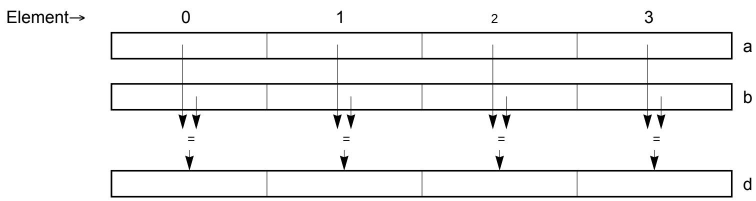
d	a	b	maps to
vector bool char	vector unsigned char	vector unsigned char	vcmpequb d,a,b
	vector signed char	vector signed char	

Figure 4-27. Compare Equal of Sixteen Integer Elements (8-bits)



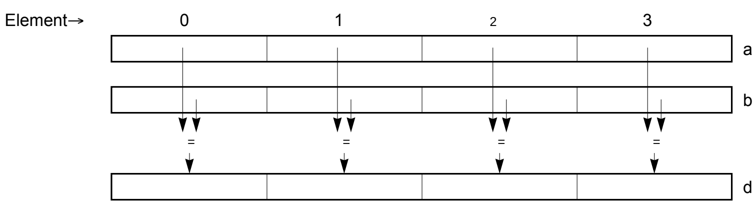
d	a	b	maps to
vector bool short	vector unsigned short	vector unsigned short	vcmpequh d,a,b
	vector signed short	vector signed short	

Figure 4-28. Compare Equal of Eight Integer Elements (16-Bit)



d	a	b	maps to
vector bool int	vector unsigned int	vector unsigned int	vcmpequw d,a,b
	vector signed int	vector signed int	

Figure 4-29. Compare Equal of Four Integer Elements (32-Bit)



d	a	b	maps to
vector bool int	vector float	vector float	vcmpeqfp d,a,b

Figure 4-30. Compare Equal of Four Floating-Point Elements (32-Bit)

vec_cmpge

vec_cmpge

Vector Compare Greater Than or Equal

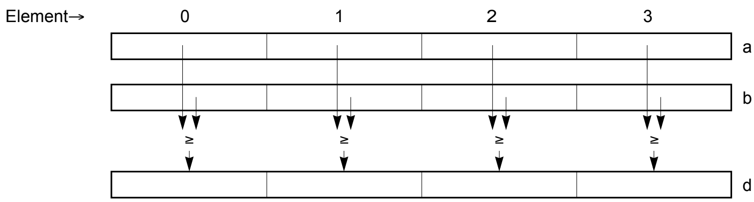
d = vec_cmpge(**a**,**b**)

```
do i=0 to 3
if aᵢ ≥fp bᵢ
        then dᵢ ← ³²1
        else dᵢ ← ³²0
end
```

Each element of the result is all ones if the corresponding element of a is greater than or equal to the corresponding element of b. Otherwise, it returns all zeros.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for **d** = vec_cmpge(**a**,**b**) are shown in Figure 4-31.



d	a	b	maps to
vector bool int	vector float	vector float	vcmpgefp d,a,b

Figure 4-31. Compare Greater-Than-or-Equal of Four Floating-Point Elements (32-Bit)

vec_cmpgt

vec_cmpgt

Vector Compare Greater Than

d = vec_cmpgt(**a**,**b**)

- Integer compare greater than:

```
n ← number of elements
m ← number of bits in an element (128/n)
do i=0 to n-1
if a_i > b_i
        then d_i ← ^m1
        else d_i ← ^m0
end
```

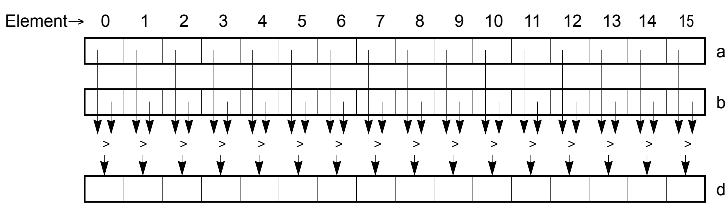
- Floating-point compare greater than:

```
do i=0 to 3
if a_i >_fp b_i
        then d_i ← ^32 1
        else d_i ← ^32 0
end
```

Each element of the result is all ones if the corresponding element of a is greater than the corresponding element of b. Otherwise, it returns all zeros.

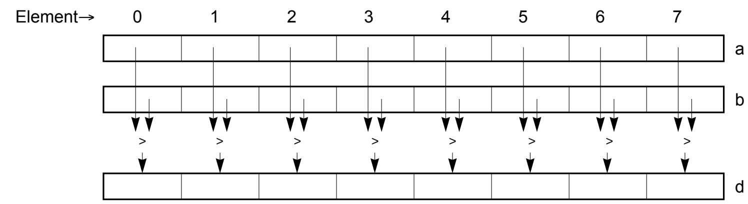
For `vector float` types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result types for **`d`** `= vec_cmpgt(`**`a,b`**`)` are shown in Figure 4-32, Figure 4-33, Figure 4-34, and Figure 4-35.



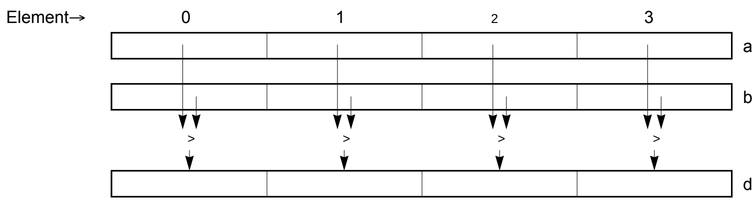
d	a	b	maps to
vector bool char	vector unsigned char	vector unsigned char	vcmpgtub d,a,b
	vector signed char	vector signed char	vcmpgtsb d,a,b

Figure 4-32. Compare Greater-Than of Sixteen Integer Elements (8-bits)



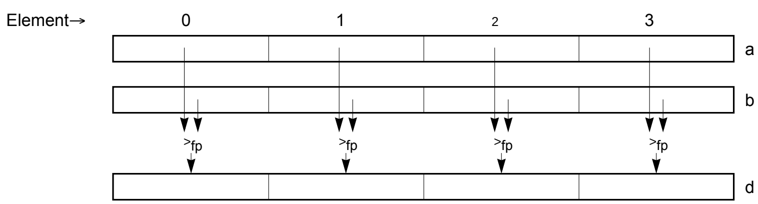
d	a	b	maps to
vector bool short	vector unsigned short	vector unsigned short	vcmpgtuh d,a,b
	vector signed short	vector signed short	vcmpgtsh d,a,b

Figure 4-33. Compare Greater-Than of Eight Integer Elements (16-Bit)



d	a	b	maps to
vector bool int	vector unsigned int	vector unsigned int	vcmpgtuw d,a,b
	vector signed int	vector signed int	vcmpgtsw d,a,b

Figure 4-34. Compare Greater-Than of Four Integer Elements (32-Bit)



d	a	b	maps to
vector bool int	vector float	vector float	vcmpgtfp d,a,b

Figure 4-35. Compare Greater-Than of Four Floating-Point Elements (32-Bit)

vec_cmple

vec_cmple

Vector Compare Less Than or Equal

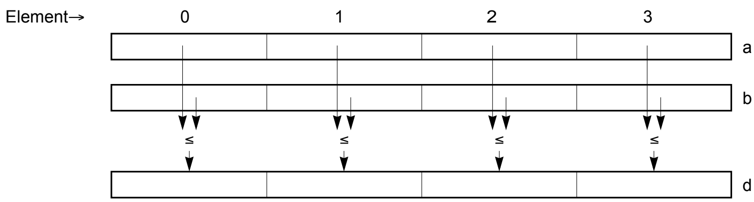
d = vec_cmple(**a**,**b**)

```
do i=0 to 3
if a_i ≤fp b_i
        then d_i ← 32 1
        else d_i ← 32 0
end
```

Each element of the result is all ones if the corresponding element of a is less than or equal to the corresponding element of b. Otherwise, it returns all zeros.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for `d = vec_cmple(a,b)` are shown in Figure 4-36. It is necessary to use the generic name, since the specific operation `vec_vcmpgefp` does not reverse its operands.



d	a	b	maps to
vector bool int	vector float	vector float	vcmpgefp d,b,a

Figure 4-36. Compare Less-Than-or-Equal of Four Floating-Point Elements (32-Bit)

vec_cmplt

vec_cmplt

Vector Compare Less Than

d = vec_cmplt(**a**,**b**)

- Integer compare less than:

```
n ← number of elements
m ← number of bits in an element (128/n)
do i=0 to n-1
if a_i < b_i
        then d_i ← ^m1
        else d_i ← ^m0
end
```

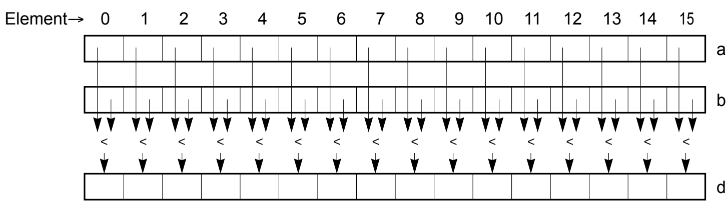
- Floating-point compare less than:

```
do i=0 to 3
if a_i <_fp b_i
        then d_i ← ^32 1
        else d_i ← ^32 0
end
```

Each element of the result is all ones if the corresponding element of a is less than the corresponding element of b. Otherwise, it returns all zeros.

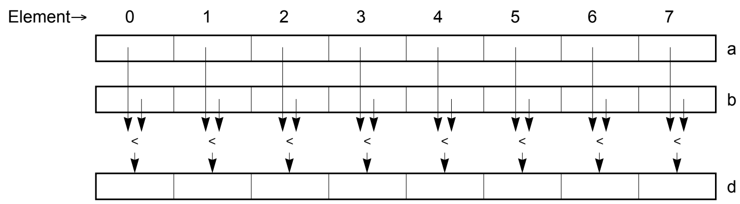
For `vector float` types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result types for `d = vec_cmplt(a,b)` are shown in Figure 4-37, Figure 4-38, Figure 4-39, and Figure 4-40. It is necessary to use the generic name, since the specific operations do not reverse their operands.



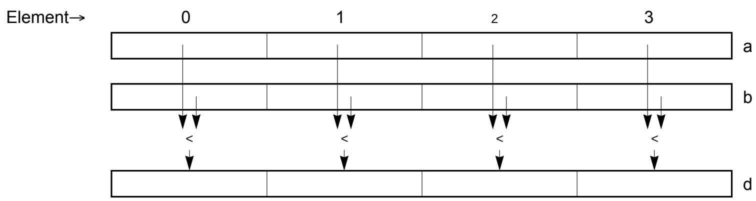
d	a	b	maps to
vector bool char	vector unsigned char	vector unsigned char	vcmpgtub d,b,a
	vector signed char	vector signed char	vcmpgtsb d,b,a

Figure 4-37. Compare Less-Than of Sixteen Integer Elements (8-bits)



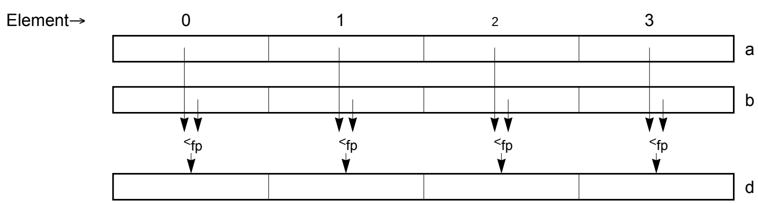
d	a	b	maps to
vector bool short	vector unsigned short	vector unsigned short	vcmpgtuh d,b,a
	vector signed short	vector signed short	vcmpgtsh d,b,a

Figure 4-38. Compare Less-Than of Eight Integer Elements (16-Bit)



d	a	b	maps to
vector bool int	vector unsigned int	vector unsigned int	vcmpgtuw d,b,a
	vector signed int	vector signed int	vcmpgtsw d,b,a

Figure 4-39. Compare Less-Than of Four Integer Elements (32-Bit)



d	a	b	maps to
vector bool int	vector float	vector float	vcmpgtfp d,b,a

Figure 4-40. Compare Less-Than of Four Floating-Point Elements (32-Bit)

vec_ctf

vec_ctf

Vector Convert from Fixed-Point Word

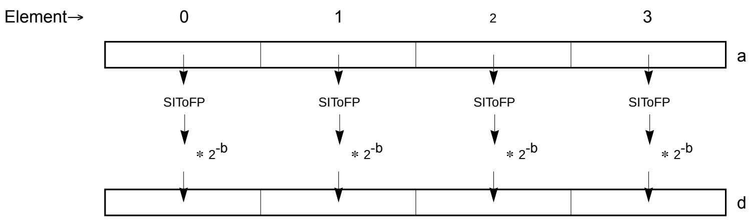
d = vec_ctf(**a**,**b**)

```
do i=0 to 3
d_i ← SIToFP(a_i) * 2^-b
end
```

Each element of the result is the closest floating-point representation of the number obtained by dividing the corresponding element of a by 2 to the power of b.

The operation is independent of VSCR[NJ].

The valid argument types and the corresponding result type for `d = vec_ctf(a,b)` are shown in Figure 4-41.



d	a	b	maps to
vector float	vector unsigned int	5-bit unsigned literal	vcfux d,a,b
	vector signed int	5-bit unsigned literal	vcfsx d,a,b

Figure 4-41. Convert Four Integer Elements to Four Floating-Point Elements (32-Bit)

vec_cts

vec_cts

Vector Convert to Signed Fixed-Point Word Saturated

d = vec_cts(**a**,**b**)

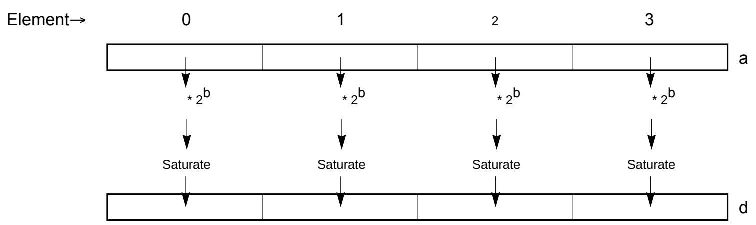
```
do i=0 to 3
d_i ← Saturate(a_i * 2^b)
end
```

Each element of the result is the saturated signed value obtained after truncating the product of the corresponding element of a and 2 to the power of b.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the operation.

If saturation occurs, VSCR[SAT] is set (see Table 4-1).

The valid argument types and the corresponding result type for **d** = vec_cts(**a**,**b**) are shown in Figure 4-42.



d	a	b	maps to
vector signed int	vector float	5-bit unsigned literal	vctsxs d,a,b

Figure 4-42. Convert Four Floating-Point Elements to Four Saturated Signed Integer Elements (32-Bit)

vec_ctu

vec_ctu

Vector Convert to Unsigned Fixed-Point Word Saturated

d = vec_ctu(**a**,**b**)

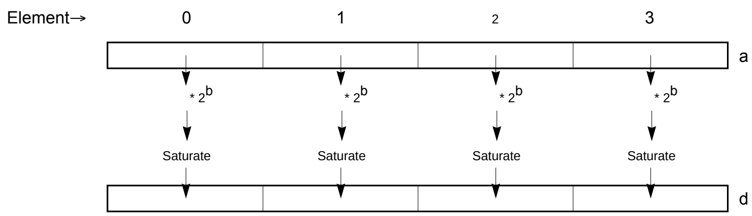
```
do i=0 to 3
di ← Saturate (ai * 2b)
end
```

Each element of the result is the saturated unsigned value obtained after truncating the number obtained by multiplying the corresponding element of a by 2 to the power of b.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the operation.

If saturation occurs, VSCR[SAT] is set (see Table 4-1).

The valid argument types and the corresponding result type for **d** = vec_ctu(**a**,**b**) are shown in Figure 4-43.



d	a	b	maps to
vector unsigned int	vector float	5-bit unsigned literal	vctuxs d,a,b

Figure 4-43. Convert Four Floating-Point Elements to Four Saturated Unsigned Integer Elements (32-Bit)

vec_dss vec_dss

Vector Data Stream Stop

vec_dss(**a**)

```
DataStreamPrefetchControl ← "stop" || a
```

Each operation stops cache touches for the data stream associated with tag a. The result is `void`. The valid argument type for `vec_dss(a)` is shown in Table 4-4. The result type is `void`.

Table 4-4. vec_dss—Vector Data Stream Stop Argument Types

a	maps to
2-bit unsigned literal	dss a

vec_dssall

vec_dssall

Vector Stream Stop All

vec_dssall()

```
DataStreamPrefetchControl ← "stop"
```

The operation stops cache touches for all data streams. All argument and result types for `vec_dssall()` are `void`. `vec_dssall` maps to the `dssall` instruction.

vec_dst

Vector Data Stream Touch

vec_dst(**a**,**b**,**c**)

```
addr[0:63] ← a
DataStreamPrefetchControl ← “start” || c || 0 || b || addr
```

Each operation initiates cache touches for loads for the data stream associated with tag c at the address a using the data block in b. The result type is `void`.

The a type may also be a pointer to a `const`-qualified type. Plain `char *` is excluded in the mapping for a.

The b type is encoded for 32-bit as follows:

- Block size: b[3:7] if b[3:7] != 0; otherwise 32
- Block count: b[8:15] if b[8:15] != 0; otherwise 256
- Block stride: b[16:31] if b[16:31] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
0 2	3 7	8 15	16 31

Figure 4-44. Format of b Type (32-bit)

The b type is encoded for 64-bit as follows:

- Block size: b[35:39] if b[35:39] != 0; otherwise 32
- Block count: b[40:47] if b[40:47] != 0; otherwise 256
- Block stride: b[48:63] if b[48:63] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
32 34	35 39	40 47	48 63

Figure 4-45. Format of b Type (64-bit)

The c type is a 2-bit unsigned literal tag used to identify a specific data stream. Up to four streams can be set up with this mechanism.

The valid combinations of argument types for `vec_dst(a,b,c)` are shown in Table 4-5. The result type is `void`.

Table 4-5. vec_dst—Vector Data Stream Touch Argument Types

a	b	c	maps to
vector unsigned char *	any integral type	2-bit unsigned literal	dst a,b,c
vector signed char *	any integral type	2-bit unsigned literal	
vector bool char *	any integral type	2-bit unsigned literal	
vector unsigned short *	any integral type	2-bit unsigned literal	
vector signed short *	any integral type	2-bit unsigned literal	
vector bool short *	any integral type	2-bit unsigned literal	
vector pixel *	any integral type	2-bit unsigned literal	
vector unsigned int *	any integral type	2-bit unsigned literal	
vector signed int *	any integral type	2-bit unsigned literal	
vector bool int *	any integral type	2-bit unsigned literal	
vector float *	any integral type	2-bit unsigned literal	
unsigned char *	any integral type	2-bit unsigned literal	
signed char *	any integral type	2-bit unsigned literal	
unsigned short *	any integral type	2-bit unsigned literal	
short *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
int *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
float *	any integral type	2-bit unsigned literal	

vec_dstst vec_dstst

Vector Data Stream Touch for Store

vec_dstst(**a**,**b**,**c**)

```
addr[0:63] ← a
DataStreamPrefetchControl ← "start" || 0 || static || b || addr
```

Each operation initiates cache touches for stores for the data stream associated with tag c at the address a using the data block in b. The result type is `void`.

The a type may also be a pointer to a `const`-qualified type. Plain `char *` is excluded in the mapping for a.

The b type is encoded for 32-bit as follows:

- Block size: b[3:7] if b[3:7] != 0; otherwise 32
- Block count: b[8:15] if b[8:15] != 0; otherwise 256
- Block stride: b[16:31] if b[16:31] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
0 2	3 7	8 15	16 31

Figure 4-46. Format of b Type (32-bit)

The b type is encoded for 64-bit as follows:

- Block size: b[35:39] if b[35:39] != 0; otherwise 32
- Block count: b[40:47] if b[40:47] != 0; otherwise 256
- Block stride: b[48:63] if b[48:63] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
32 34	35 39	40 47	48 63

Figure 4-47. Format of b Type (64-bit)

The c type is a 2-bit unsigned literal tag used to identify a specific data stream. Up to four streams can be set up with this mechanism.

The valid combinations of argument types for `vec_dstst(`**`a,b,c`**`)` are shown in Table 4-6. The result type is `void`.

Table 4-6. vec_dstst—Vector Data Stream for Touch Store Argument Types

a	b	c	maps to
vector unsigned char *	any integral type	2-bit unsigned literal	dstst a,b,c
vector signed char *	any integral type	2-bit unsigned literal	
vector bool char *	any integral type	2-bit unsigned literal	
vector unsigned short *	any integral type	2-bit unsigned literal	
vector signed short *	any integral type	2-bit unsigned literal	
vector bool short *	any integral type	2-bit unsigned literal	
vector pixel *	any integral type	2-bit unsigned literal	
vector unsigned int *	any integral type	2-bit unsigned literal	
vector signed int *	any integral type	2-bit unsigned literal	
vector bool int *	any integral type	2-bit unsigned literal	
vector float *	any integral type	2-bit unsigned literal	
unsigned char *	any integral type	2-bit unsigned literal	
signed char *	any integral type	2-bit unsigned literal	
unsigned short *	any integral type	2-bit unsigned literal	
short *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
int *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
float *	any integral type	2-bit unsigned literal	

vec_dststt vec_dststt

Vector Data Stream Touch for Store Transient

vec_dststt(**a**,**b**,**c**)

```
addr[0:63] ← a
DataStreamPrefetchControl ← "start" || 1 || static || b || addr
```

Each operation initiates cache touches for transient stores for the data stream associated with tag c at the address a using the data block in b. The result type is `void`.

The a type may also be a pointer to a `const`-qualified type. Plain `char *` is excluded in the mapping for a.

The b type is encoded for 32-bit as follows:

- Block size: b[3:7] if b[3:7] != 0; otherwise 32
- Block count: b[8:15] if b[8:15] != 0; otherwise 256
- Block stride: b[16:31] if b[16:31] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
0 2	3 7	8 15	16 31

Figure 4-48. Format of b Type (32-bit)

The b type is encoded for 64-bit as follows:

- Block size: b[35:39] if b[35:39] != 0; otherwise 32
- Block count: b[40:47] if b[40:47] != 0; otherwise 256
- Block stride: b[48:63] if b[48:63] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
32 34	35 39	40 47	48 63

Figure 4-49. Format of b Type (64-bit)

The c type is a 2-bit unsigned literal tag used to identify a specific data stream. Up to four streams can be set up with this mechanism.

The valid combinations of argument types for `vec_dststt(`**`a,b,c`**`)` are shown in Table 4-7. The result type is `void`.

Table 4-7. vec_dststt—Vector Data Stream Touch for Store Transient Argument Types

a	b	c	maps to
vector unsigned char *	any integral type	2-bit unsigned literal	dststt a,b,c
vector signed char *	any integral type	2-bit unsigned literal	
vector bool char *	any integral type	2-bit unsigned literal	
vector unsigned short *	any integral type	2-bit unsigned literal	
vector signed short *	any integral type	2-bit unsigned literal	
vector bool short *	any integral type	2-bit unsigned literal	
vector pixel *	any integral type	2-bit unsigned literal	
vector unsigned int *	any integral type	2-bit unsigned literal	
vector signed int *	any integral type	2-bit unsigned literal	
vector bool int *	any integral type	2-bit unsigned literal	
vector float *	any integral type	2-bit unsigned literal	
unsigned char *	any integral type	2-bit unsigned literal	
signed char *	any integral type	2-bit unsigned literal	
unsigned short *	any integral type	2-bit unsigned literal	
short *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
int *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
float *	any integral type	2-bit unsigned literal	

vec_dstt

vec_dstt

Vector Data Stream Touch Transient

vec_dstt(**a**,**b**,**c**)

```
addr[0:63] ← a
DataStreamPrefetchControl ← “start” || c || 1 || b || addr
```

Each operation initiates cache touches for transient loads for the data stream associated with tag c at the address a using the data block in b. The result type is `void`.

The a type may also be a pointer to a `const`-qualified type. Plain `char *` is excluded in the mapping for a.

The b type is encoded for 32-bit as follows:

- Block size: b[3:7] if b[3:7] != 0; otherwise 32
- Block count: b[8:15] if b[8:15] != 0; otherwise 256
- Block stride: b[16:31] if b[16:31] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
0 – 2	3 – 7	8 – 15	16 – 31

Figure 4-50. Format of b Type (32-bit)

The b type is encoded for 64-bit as follows:

- Block size: b[35:39] if b[35:39] != 0; otherwise 32
- Block count: b[40:47] if b[40:47] != 0; otherwise 256
- Block stride: b[48:63] if b[48:63] != 0; otherwise 32768

///	Block Size	Block Count	Block Stride
32 – 34	35 – 39	40 – 47	48 – 63

Figure 4-51. Format of b Type (64-bit)

The c type is a 2-bit unsigned literal tag used to identify a specific data stream. Up to four streams can be set up with this mechanism.

The valid combinations of argument types for `vec_dstt(a,b,c)` are shown in Table 4-8. The result type is `void`.

Table 4-8. vec_dstt—Vector Data Stream Touch Transient Argument Types

a	b	c	maps to
vector unsigned char *	any integral type	2-bit unsigned literal	dst a,b,c
vector signed char *	any integral type	2-bit unsigned literal	
vector bool char *	any integral type	2-bit unsigned literal	
vector unsigned short *	any integral type	2-bit unsigned literal	
vector signed short *	any integral type	2-bit unsigned literal	
vector bool short *	any integral type	2-bit unsigned literal	
vector pixel *	any integral type	2-bit unsigned literal	
vector unsigned int *	any integral type	2-bit unsigned literal	
vector signed int *	any integral type	2-bit unsigned literal	
vector bool int *	any integral type	2-bit unsigned literal	
vector float *	any integral type	2-bit unsigned literal	
unsigned char *	any integral type	2-bit unsigned literal	
signed char *	any integral type	2-bit unsigned literal	
unsigned short *	any integral type	2-bit unsigned literal	
short *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
int *	any integral type	2-bit unsigned literal	
unsigned int *	any integral type	2-bit unsigned literal	
float *	any integral type	2-bit unsigned literal	

vec_expte vec_expte

Vector Is 2 Raised to the Exponent Estimate Floating-Point

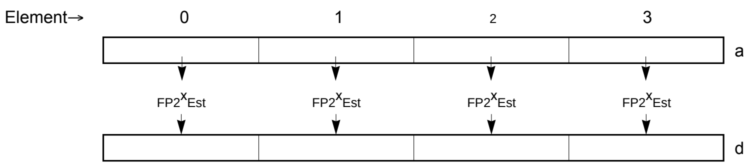
d = vec_expte(**a**)

```
do i=0 to 3
d_i ← FP2^XEst(a_i)
end
```

Each element of the result is an estimate of 2 raised to the corresponding element of a.

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element is truncated to a 0 of the same sign.

The valid argument type and corresponding result type for `d = vec_expte(a) are` shown in Figure 4-52.



d	a	maps to
vector float	vector float	vexptefp d,a

Figure 4-52. 2 Raised to the Exponent Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vec_floor

vec_floor

Vector Floor

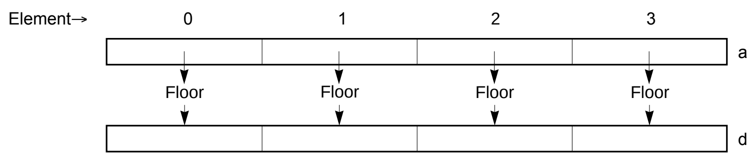
d = vec_floor(**a**)

```
do i=0 to 3
d_i ← Floor(a_i)
end
```

Each single-precision floating-point word element in a is rounded to a single-precision floating-point integer using the rounding mode Round towards –Infinity, and placed into the corresponding word element of d. Each element of the result is thus the largest representable floating-point integer not greater than a. For example, if the floating-point element was 123.85, the resulting integer would be 123.

If VSCR[NJ] = 1, every denormalized operand element is truncated to 0 before rounding.

The valid argument type and corresponding result type for `d = vec_floor(a)` are shown in Figure 4-53.



d	a	maps to
vector float	vector float	vrfim d,a

Figure 4-53. Round to Minus Infinity of Four Floating-Point Integer Elements (32-Bit)

vec_ld vec_ld

Vector Load Indexed

d = vec_ld(**a**,**b**)

```
EA ← BoundAlign(a+b,16)
d ← MEM(EA,16)
```

Each operation performs a 16-byte load at a 16-byte aligned address. The a is taken to be an integer value, while b is a pointer. BoundAlign(a+b,16) is the largest value less than or equal to a + b that is a multiple of 16. This load is the one that is generated for a loading dereference of a pointer to a vector type. The b type may also be a pointer to a `const`-qualified type. Plain `char *` is excluded in the mapping for b. The valid combinations of argument types and the corresponding result types for `d = vec_ld(a,b)` are shown in Table 4-9.

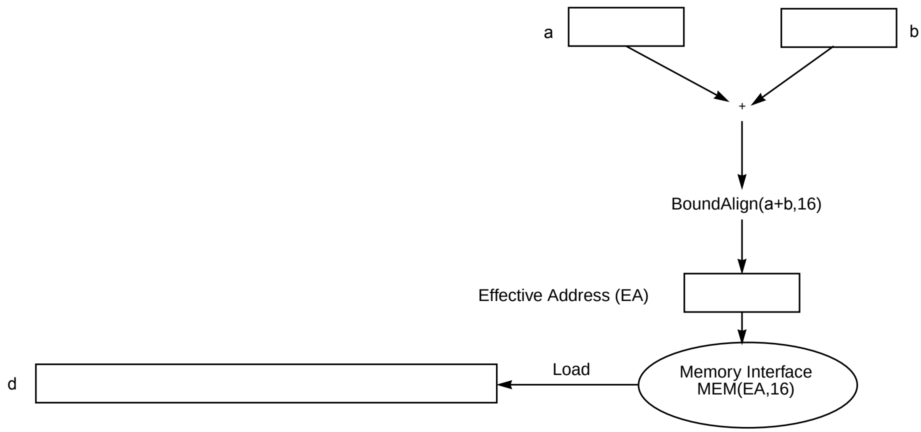


Figure 4-54. Vector Load Indexed Operation

Table 4-9. vec_ld—Load Vector Indexed Argument Types

d	a	b	maps to
vector unsigned char	any integral type	vector unsigned char *	lvx d,a,b
	any integral type	unsigned char *	
vector signed char	any integral type	vector signed char *	
	any integral type	signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	vector unsigned short *	
	any integral type	unsigned short *	
vector signed short	any integral type	vector signed short *	
	any integral type	short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	vector unsigned int *	
	any integral type	unsigned int*	
	any integral type	unsigned int *	
vector signed int	any integral type	vector signed int *	
	any integral type	int *	
	any integral type	int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	vector float *	
	any integral type	float *	

vec_lde

Vector Load Element Indexed

d = vec_lde(**a**,**b**)

```
s ← 16/(number of elements)
EA ← BoundAlign(a+b,s)
i ← mod(EA,16)/s
d_i ← MEM(EA,s)
```

Each operation loads a single element into the position in the vector register corresponding to its address, leaving the remaining elements of the register undefined. The a is taken to be an integer value, while b is a pointer. BoundAlign(a+b,s) is the largest value less than or equal to **a + b** that is a multiple of s, where s is 1 for `char` pointers, 2 for `short` pointers, and 4 for `int` or `float` pointers. The b type may also be a pointer to a `const`-qualified type. Plain `char *` is excluded in the mapping for b. The valid combinations of argument types and the corresponding result types for `d = vec_lde(a,b)` are shown in Table 4-10.

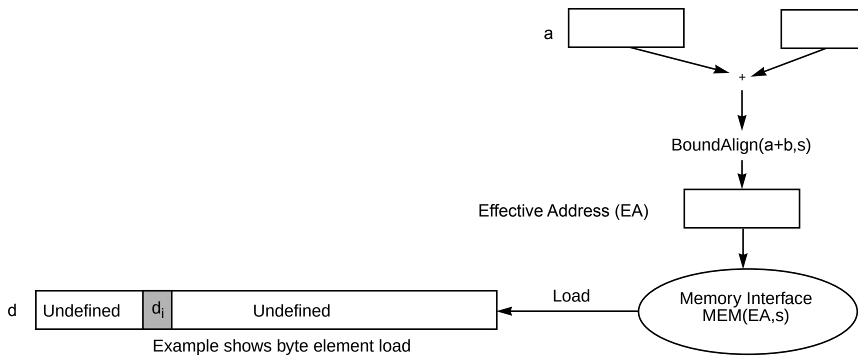


Figure 4-55. Vector Load Element Indexed Operation

Table 4-10. vec_lde(a,b)—Vector Load Element Indexed Argument Types

d	a	b	Maps to
vector unsigned char	any integral type	unsigned char *	lvebx d,a,b
vector signed char	any integral type	signed char *	
vector unsigned short	any integral type	unsigned short *	lvehx d,a,b
vector signed short	any integral type	short *	
vector unsigned int	any integral type	unsigned int *	lvewx d,a,b
	any integral type	unsigned int *	
vector signed int	any integral type	int *	
vector float	any integral type	float *	

vec_ldl

vec_ldl

Vector Load Indexed LRU

d = vec_ldl(**a**,**b**)

```
EA ← BoundAlign(a+b,16)
d ← MEM(EA,16)
```

Each operation performs a 16-byte load at a 16-byte aligned address. The a is taken to be an integer value, while b is a pointer. BoundAlign(a+b,16) is the largest value less than or equal to a + b that is a multiple of 16. These operations mark the cache line as least-recently-used. The b type may also be a pointer to a `const`-qualified type. Plain `char *` is excluded in the mapping for b. The valid combinations of argument types and the corresponding result types for `d = vec_ldl(a,b)` are shown in Table 4-11.

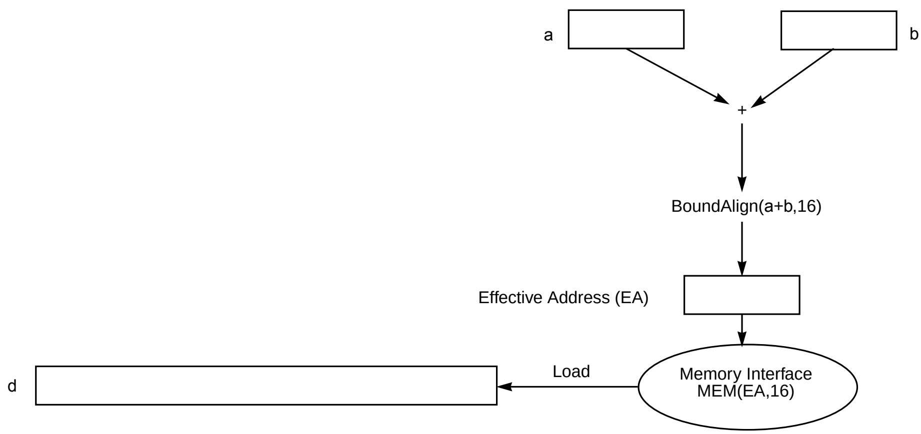


Figure 4-56. Vector Load Indexed LRU Operation

Table 4-11. vec_ldl—Vector Load Indexed LRU Argument Types

d	a	b	Maps to
vector unsigned char	any integral type	vector unsigned char *	lvxl d,a,b
	any integral type	unsigned char *	
vector signed char	any integral type	vector signed char *	
	any integral type	signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	vector unsigned short *	
	any integral type	unsigned short *	
vector signed short	any integral type	vector signed short *	
	any integral type	short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	vector unsigned int *	
	any integral type	unsigned int *	
vector signed int	any integral type	vector signed int *	
	any integral type	int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	vector float *	
	any integral type	float *	

vec_loge

vec_loge

Vector Log_2 Estimate Floating-Point

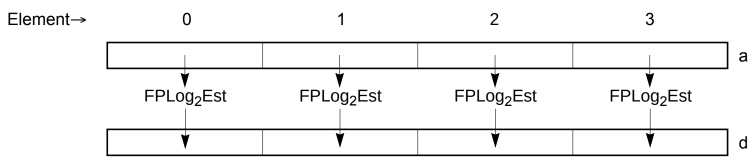
d = vec_loge(**a**)

```
do i=0 to 3
d_i ← FPLog2Est(a_i)
end
```

Each element of the result is an estimate of the logarithm to base 2 of the corresponding element of a.

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out.

The valid argument type and corresponding result type for `d = vec_loge(a)` are shown in Figure 4-57



d	a	maps to
vector float	vector float	vlogefp d,a

Figure 4-57. Log_2 Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vec_lvsl vec_lvsl

Vector Load for Shift Left

d = vec_lvsl(**a**,**b**)

```
EA ← a + b
sh ← EA[28:31]
if sh = 0x0 then d ← 0x000102030405060708090A0B0C0D0E0F
if sh = 0x1 then d ← 0x0102030405060708090A0B0C0D0E0F10
if sh = 0x2 then d ← 0x02030405060708090A0B0C0D0E0F1011
if sh = 0x3 then d ← 0x030405060708090A0B0C0D0E0F101112
if sh = 0x4 then d ← 0x0405060708090A0B0C0D0E0F10111213
if sh = 0x5 then d ← 0x05060708090A0B0C0D0E0F1011121314
if sh = 0x6 then d ← 0x060708090A0B0C0D0E0F101112131415
if sh = 0x7 then d ← 0x0708090A0B0C0D0E0F10111213141516
if sh = 0x8 then d ← 0x08090A0B0C0D0E0F1011121314151617
if sh = 0x9 then d ← 0x090A0B0C0D0E0F101112131415161718
if sh = 0xA then d ← 0x0A0B0C0D0E0F10111213141516171819
if sh = 0xB then d ← 0x0B0C0D0E0F101112131415161718191A
if sh = 0xC then d ← 0x0C0D0E0F101112131415161718191A1B
if sh = 0xD then d ← 0x0D0E0F101112131415161718191A1B1C
if sh = 0xE then d ← 0x0E0F101112131415161718191A1B1C1D
if sh = 0xF then d ← 0x0F101112131415161718191A1B1C1D1E
```

Each operation generates a permutation useful for aligning data from an unaligned address. The b type may also be a pointer to a `const`- or `volatile`-qualified type.
Plain `char *` is excluded in the mapping for b. The valid combination of argument types and the corresponding result type for `d = vec_lvsl(a,b)` are shown in Table 4-12.

Table 4-12. vec_lvsl—Load Vector for Shift Left Argument Types

d	a	b	maps to
vector unsigned char	any integral type	unsigned char *	lvsl d,a,b
	any integral type	signed char *	
	any integral type	unsigned short *	
	any integral type	short *	
	any integral type	unsigned int *	
	any integral type	int *	
	any integral type	float *	

vec_lvsr

vec_lvsr

Vector Load Shift Right

d = vec_lvsr(**a**,**b**)

```
EA ← a + b
sh ← EA[28:31]
if sh=0x0 then d ← 0x101112131415161718191A1B1C1D1E1F
if sh=0x1 then d ← 0x0F101112131415161718191A1B1C1D1E
if sh=0x2 then d ← 0x0E0F101112131415161718191A1B1C1D
if sh=0x3 then d ← 0x0D0E0F101112131415161718191A1B1C
if sh=0x4 then d ← 0x0C0D0E0F101112131415161718191A1B
if sh=0x5 then d ← 0x0B0C0D0E0F101112131415161718191A
if sh=0x6 then d ← 0x0A0B0C0D0E0F10111213141516171819
if sh=0x7 then d ← 0x090A0B0C0D0E0F101112131415161718
if sh=0x8 then d ← 0x08090A0B0C0D0E0F1011121314151617
if sh=0x9 then d ← 0x0708090A0B0C0D0E0F10111213141516
if sh=0xA then d ← 0x060708090A0B0C0D0E0F101112131415
if sh=0xB then d ← 0x05060708090A0B0C0D0E0F1011121314
if sh=0xC then d ← 0x0405060708090A0B0C0D0E0F10111213
if sh=0xD then d ← 0x030405060708090A0B0C0D0E0F101112
if sh=0xE then d ← 0x02030405060708090A0B0C0D0E0F1011
if sh=0xF then d ← 0x0102030405060708090A0B0C0D0E0F10
```

Each operation generates a permutation useful for aligning data from an unaligned address. The b type may also be a pointer to a `const`- or `volatile`-qualified type. Plain `char *` is excluded in the mapping for b. The valid combinations of argument types and the corresponding result type for **`d = vec_lvsr(a,b)`** are shown in Table 4-13.

Table 4-13. vec_lvsr—Vector Load for Shift Right Argument Types

d	a	b	Maps to
vector unsigned char	any integral type	unsigned char *	lvsr d,a,b
	any integral type	signed char *	
	any integral type	unsigned short *	
	any integral type	short *	
	any integral type	unsigned int *	
	any integral type	int *	
	any integral type	float *	

vec_madd

vec_madd

Vector Multiply Add

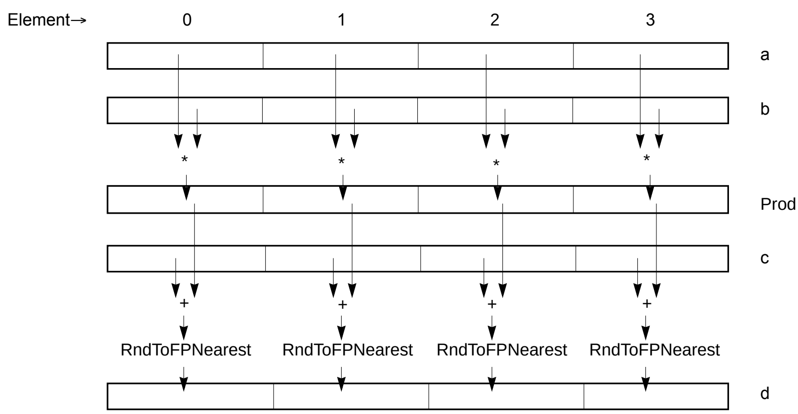
d = vec_madd(**a**,**b**,**c**)

```
do i=0 to 3
di ← RndToFPNearest(ai * bi + ci)
end
```

Each element of the result is the sum of the corresponding element of c and the product of the corresponding elements of a and b.

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

The valid argument types and the corresponding result type for `d = vec_madd(a,b,c)` are shown in Figure 4-58



d	a	b	c	maps to
vector float	vector float	vector float	vector float	vmaddfp d,a,b,c

Figure 4-58. Multiply-Add Four Floating-Point Elements (32-Bit)

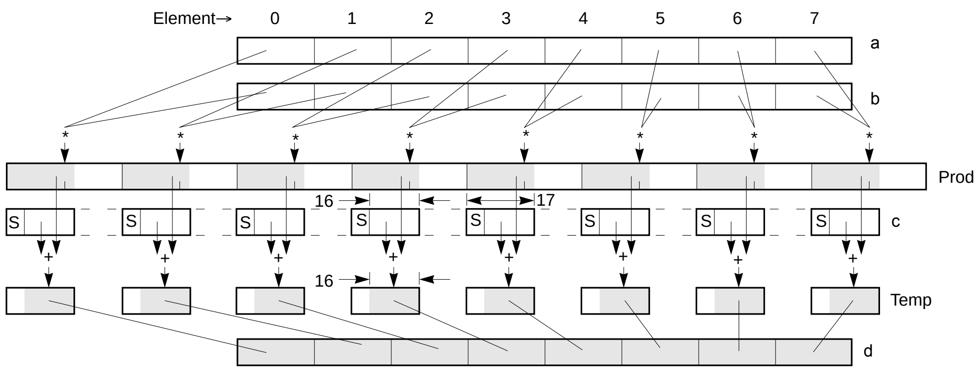
vec_madds vec_madds

Vector Multiply Add Saturated

d = vec_madds(**a**,**b**,**c**)

```
do i=0 to 7
d_i ← Saturate((a_i * b_i)/2^15 + c_i)
end
```

Each element of the result is the 16-bit saturated sum of the corresponding element of c and the high-order 17 bits of the product of the corresponding elements of a and b. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The valid argument types and the corresponding result type for **d = vec_madds(a,b,c)** are shown in Figure 4-59.



d	a	b	c	maps to
vector signed short	vector signed short	vector signed short	vector signed short	vmhaddshs d,a,b,c

Figure 4-59. Multiply-Add Four Floating-Point Elements (32-Bit)

vec_max

Vector Maximum

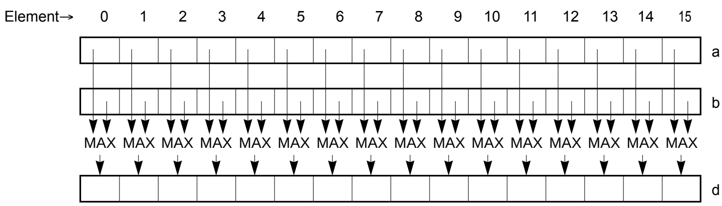
d = vec_max(**a**,**b**)

```
n ← number of elements
do i=0 to n-1
d_i ← MAX(a_i,b_i)
end
```

Each element of the result is the larger of the corresponding elements of a and b.

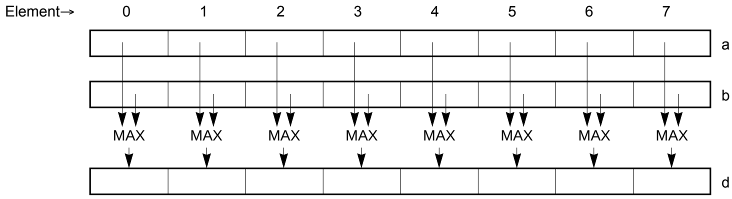
For `vector float` argument types, if VSCR[NJ] is set, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign. The maximum of +0.0 and –0.0 is +0.0. The maximum of any value and a NaN is a QNaN.

The valid combinations of argument types and the corresponding result types for **`d = vec_max(a,b)`** are shown in Figure 4-60, Figure 4-61, Figure 4-62, and Figure 4-63.



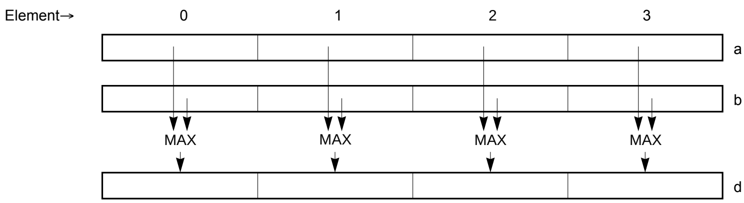
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vmaxub d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	vmaxsb d,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-60. Maximum of Sixteen Integer Elements (8-Bit)



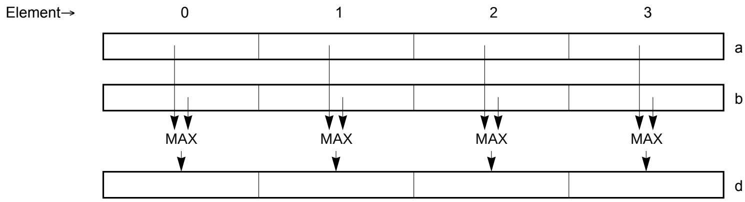
d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vmaxuh d,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	vmaxsh d,a,b
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-61. Maximum of Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vmaxuw d,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	vmaxsw d,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-62. Maximum of Four Integer Elements (32-bit)



d	a	b	maps to
vector float	vector float	vector float	vmaxfp d,a,b

Figure 4-63. Maximum of Four Floating-Point Elements (32-bit)

vec_mergeh

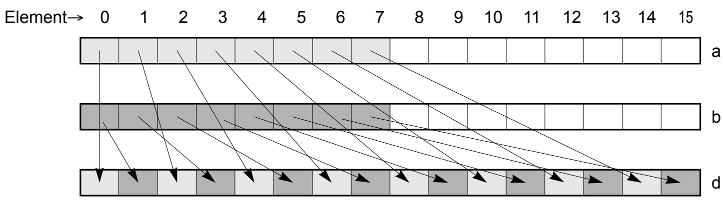
vec_mergeh

Vector Merge High

d = vec_mergeh(**a**,**b**)

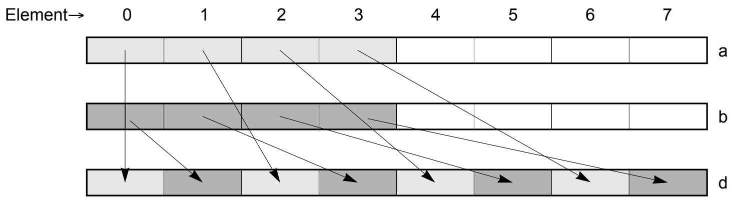
```
m ← (number of elements)/2
do i=0 to m-1
d2i ← ai
d2i+1 ← bi
end
```

The even elements of the result are obtained left-to-right from the high elements of a. The odd elements of the result are obtained left-to-right from the high elements of b. The valid combinations of argument types and the corresponding result types for **`d = vec_mergeh(a,b)`** are shown in Figure 4-64, Figure 4-65, and Figure 4-66.



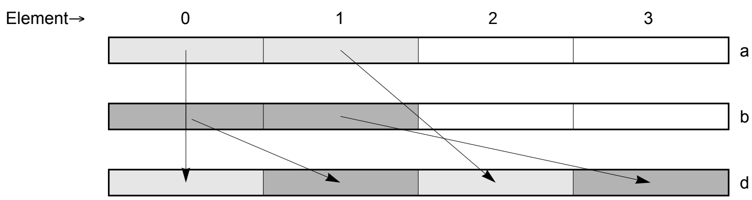
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vmrghb d,a,b
vector signed char	vector signed char	vector signed char	
vector bool char	vector bool char	vector bool char	

Figure 4-64. Merge Eight High-Order Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vmrghh d,a,b
vector signed short	vector signed short	vector signed short	
vector bool short	vector bool short	vector bool short	
vector pixel	vector pixel	vector pixel	

Figure 4-65. Merge Four High-Order Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vmrghw d,a,b
vector signed int	vector signed int	vector signed int	
vector bool int	vector bool int	vector bool int	
vector float	vector float	vector float	

Figure 4-66. Merge Two High-Order Elements (32-bit)

vec_mergel

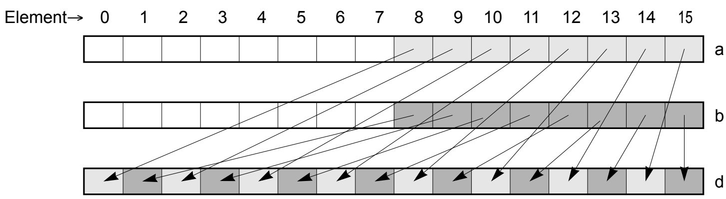
vec_mergel

Vector Merge Low

d = vec_mergel(**a**,**b**)

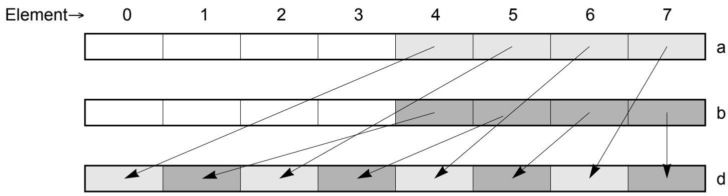
```
m ← (number of elements)/2
do i=0 to m-1
d2i ← ai+m
d2i+1 ← bi+m
end
```

The even elements of the result are obtained left-to-right from the low elements of a. The odd elements of the result are obtained left-to-right from the low elements of b. The valid combinations of argument types and the corresponding result types for **d** = vec_mergel(**a**,**b**) are shown in Figure 4-67, Figure 4-68, and Figure 4-69.



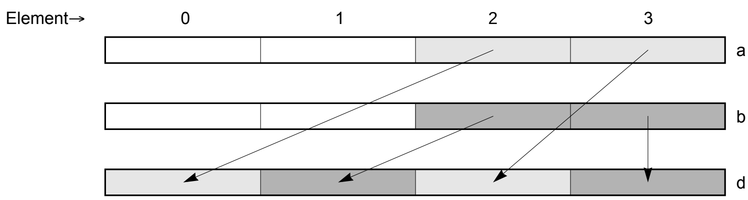
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vmrglb d,a,b
vector signed char	vector signed char	vector signed char	
vector bool char	vector bool char	vector bool char	

Figure 4-67. Merge Eight Low-Order Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vmrglh d,a,b
vector signed short	vector signed short	vector signed short	
vector bool short	vector bool short	vector bool short	
vector pixel	vector pixel	vector pixel	

Figure 4-68. Merge Four Low-Order Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vmrglw d,a,b
vector signed int	vector signed int	vector signed int	
vector bool int	vector bool int	vector bool int	
vector float	vector float	vector float	

Figure 4-69. Merge Two Low-Order Elements (32-bit)

vec_mfvscr vec_mfvscr

Vector Move from Vector Status and Control Register

d = vec_mfvscr

$\mathbf{d} \leftarrow {}^{96}0 \parallel (\mathrm{VSCR})$

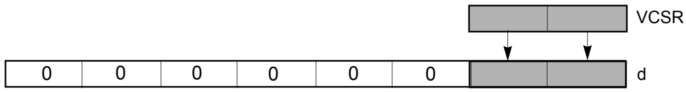


Figure 4-70. Vector Move from VSCR

Table 4-14. Vector Move from Vector Status and Control Registers Argument Type and Mapping

d	Maps to
vector unsigned short	mfvscr

vec_min vec_min

Vector Minimum

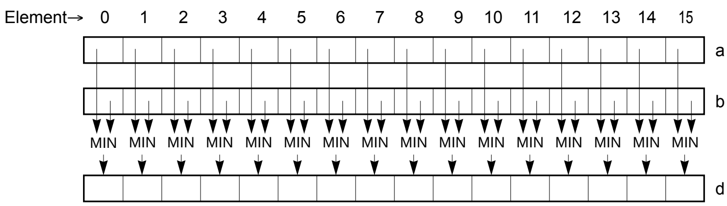
d = vec_min(**a**,**b**)

```
n ← number of elements
do i=0 to n-1
d_i ← MIN(a_i,b_i)
end
```

Each element of the result is the smaller of the corresponding elements of a and b.

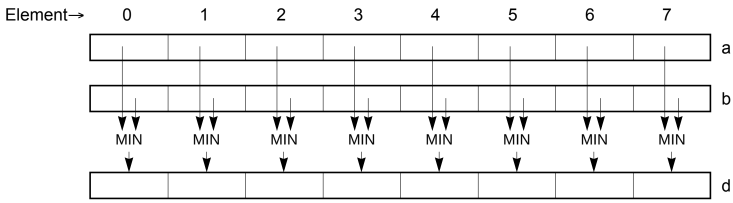
For `vector float` argument types, if VSCR[NJ] is set, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign. The minimum of +0.0 and –0.0 is –0.0. The minimum of any value and a NaN is a QNaN.

The valid combinations of argument types and the corresponding result types for **`d`** `= vec_min(`**`a,b`**`)` are shown in Figure 4-71, Figure 4-72, Figure 4-73, and Figure 4-74.



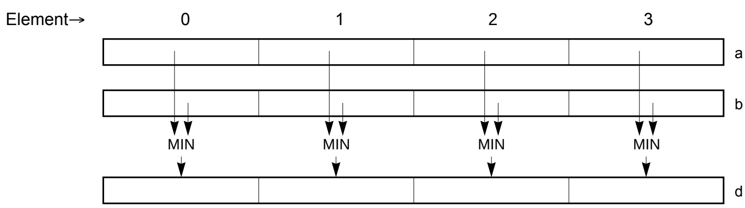
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vminub d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	vminsb d,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-71. Minimum of Sixteen Integer Elements (8-Bit)



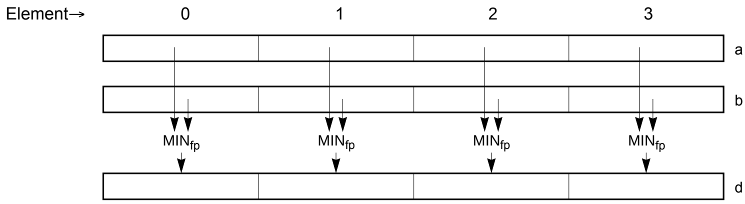
d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vminuh d,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	vminsh d,a,b
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-72. Minimum of Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vminuw d,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	vminsw d,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-73. Minimum of Four Integer Elements (32-bit)



d	a	b	maps to
vector float	vector float	vector float	vminfp d,a,b

Figure 4-74. Minimum of Four Floating-Point Elements (32-bit)

vec_mladd

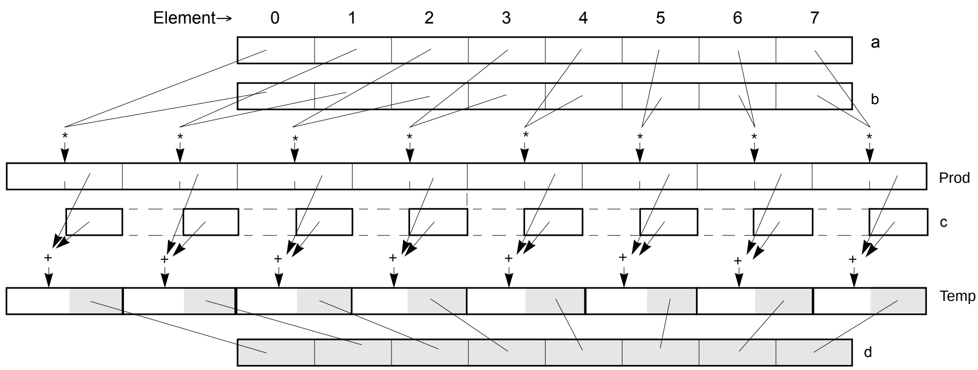
vec_mladd

Vector Multiply Low and Add Unsigned Half Word

d = vec_mladd(**a**,**b**,**c**)

```
do i=0 to 7
   di ← (ai * bi) + ci
end
```

Each element of the result is the low-order 16 bits of the sum of the corresponding element of c and the product of the corresponding elements of a and b. The valid combinations of argument types and the corresponding result types for `d = vec_mladd(a,b)` are shown in Figure 4-75.



d	a	b	c	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	vmladduhm d,a,b,c
vector signed short	vector unsigned short	vector signed short	vector signed short	
	vector signed short	vector unsigned short	vector unsigned short	
	vector signed short	vector signed short	vector signed short	

Figure 4-75. Multiply-Add of Eight Integer Elements (16-Bit)

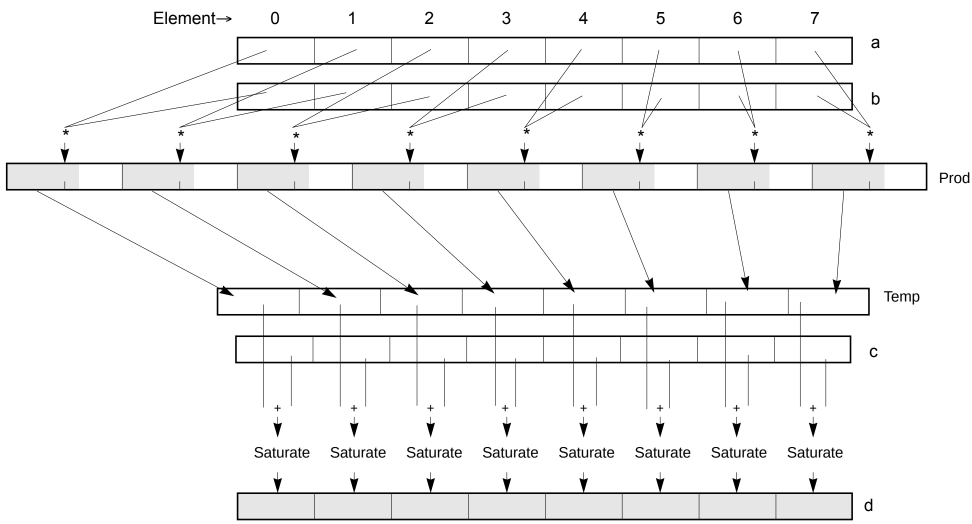
vec_mradds vec_mradds

Vector Multiply Round and Add Saturated

d = vec_mradds(**a**,**b**,**c**)

```
do i=0 to 7
di ← Saturate((ai * bi + 2^14)/2^15 + ci)
end
```

Each element of the result is the 16-bit saturated sum of the corresponding element of c and the high-order 17 bits of the rounded product of the corresponding elements of a and b. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The valid argument types and the corresponding result type for `d = vec_mradds(a,b,c)` are shown in Figure 4-76.



d	a	b	c	maps to
vector signed short	vector signed short	vector signed short	vector signed short	vmhraddshs d,a,b,c

Figure 4-76. Multiply-Add of Eight Integer Elements (16-Bit)

vec_msum

vec_msum

Vector Multiply Sum

d = vec_msum(**a**,**b**,**c**)

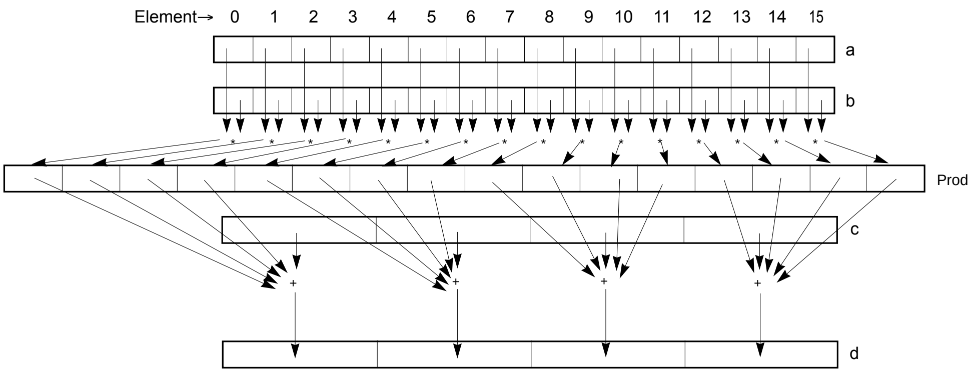
- For Multiply Sum of Sixteen 8-bit elements

```
do i=0 to 3
d_i ← (a_4i * b_4i) + (a_4i+1 * b_4i+1) + (a_4i+2 * b_4i+2) + (a_4i+3 * b_4i+3) +c_i
end
```

- For Multiply Sum of Eight 16-bit elements

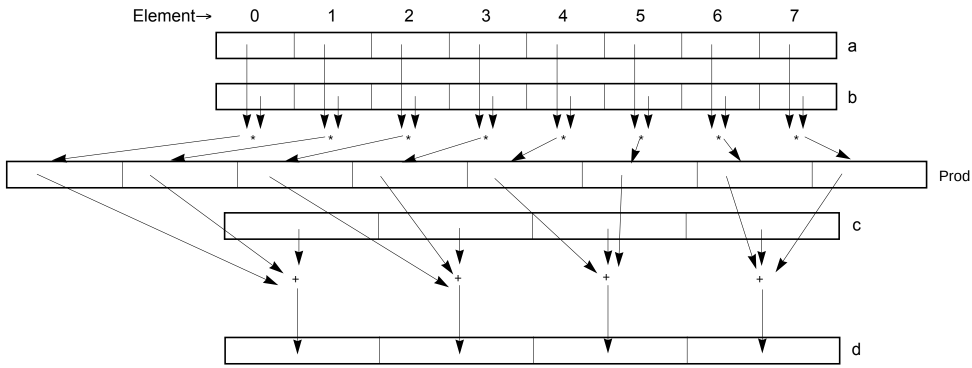
```
do i=0 to 3
d_i ← (a_2i * b_2i) + (a_2i+1 * b_2i+1) +c_i
end
```

Each element of the result is the sum of the corresponding element of c and the products of the elements of a and b which overlap the positions of that element of c. For `vec_msum`, the sum is performed with 32-bit modular addition. The valid combinations of argument types and the corresponding result types for `d = vec_msum(a,b,c)` are shown in Figure 4-77 and Figure 4-78.



d	a	b	c	maps to
vector unsigned int	vector unsigned char	vector unsigned char	vector unsigned int	vmsumubm d,a,b,c
vector signed int	vector signed char	vector unsigned char	vector signed int	vmsummbm d,a,b,c

Figure 4-77. Multiply Sum of Sixteen Integer Elements (8-Bit)



d	a	b	c	maps to
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	vmsumuhm d,a,b,c
vector signed int	vector signed short	vector signed short	vector signed int	vmsumshm d,a,b,c

Figure 4-78. Multiply Sum of Eight Integer Elements (16-Bit)

vec_msums

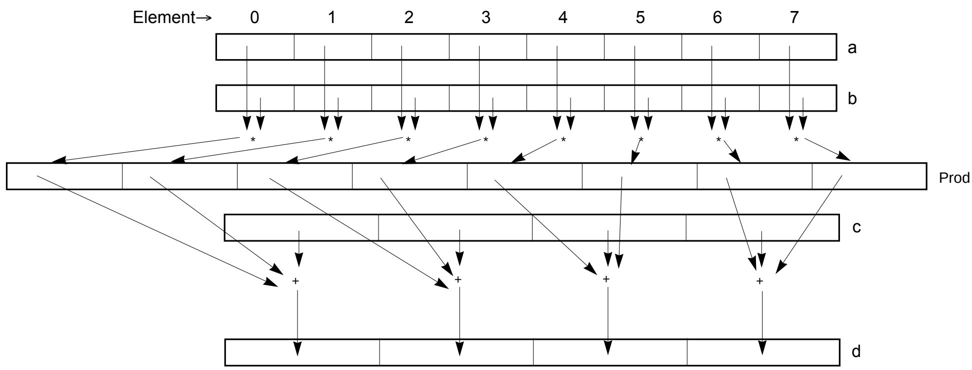
vec_msums

Vector Multiply Sum Saturated

d = vec_msums(**a**,**b**,**c**)

```
do i=0 to 3
di ← Saturate((a2i * b2i) + (a2i+1 * b2i+1) + ci)
end
```

Each element of the result is the sum of the corresponding element of c and the products of the elements of a and b which overlap the positions of that element of c. The sum is performed with 32-bit saturating addition. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The valid combinations of argument types and the corresponding result types for `d = vec_msums(a,b,c)` are shown in Figure 4-79.



d	a	b	c	maps to
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	vmsumuhs d,a,b,c
vector signed int	vector signed short	vector signed short	vector signed int	vmsumshs d,a,b,c

Figure 4-79. Multiply-Sum of Integer Elements (16-Bit to 32-Bit)

vec_mtvscr

vec_mtvscr

Vector Move to Vector Status and Control Register

vec_mtvscr(**a**)

```
VSCR ← a[96:127]
```

The VSCR is set by the elements in a which occupy the last 32 bits. The result is `void`.

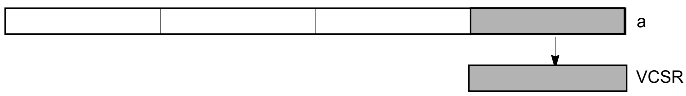


Figure 4-80. Vector Move to VSCR

Refer to the description of `vec_mfvscr` for a detailed description of the VSCR (see Figure 4-1). The valid argument types for `vec_mtvscr(`**`a`**`)` are shown in Table 4-15. The result type is `void`.

Table 4-15. vec_mtvscr—Vector Move to Vector Status and Control Register Argument Types

a	Maps to
vector unsigned char	mtvscr a
vector signed char	
vector bool char	
vector unsigned short	
vector signed short	
vector bool short	
vector pixel	
vector unsigned int	
vector signed int	
vector bool int	

vec_mule

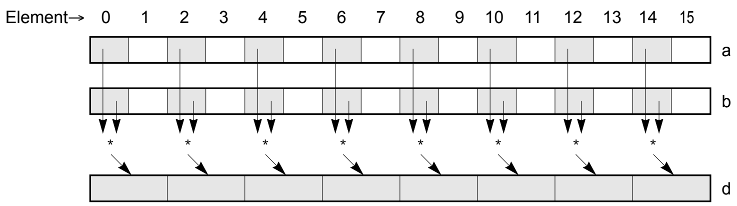
vec_mule

Vector Multiply Even

d = vec_mule(**a**,**b**)

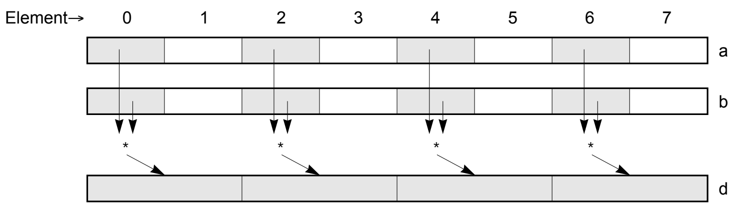
```
n ← number of elements in d
do i=0 to n-1
d_i      ← a_2i * b_2i
end
```

Each element of the result is the product of the corresponding high half-width elements of a and b. The odd elements of a and b are ignored. The valid combinations of argument types and the corresponding result types for `d = vec_mule(a,b)` are shown in Figure 4-81 and Figure 4-82.



d	a	b	maps to
vector unsigned short	vector unsigned char	vector unsigned char	vmuleub d,a,b
vector signed short	vector signed char	vector signed char	vmulesb d,a,b

Figure 4-81. Even Multiply of Eight Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned int	vector unsigned short	vector unsigned short	vmuleuh d,a,b
vector signed int	vector signed short	vector signed short	vmulesh d,a,b

Figure 4-82. Even Multiply of Four Integer Elements (16-Bit)

vec_mulo

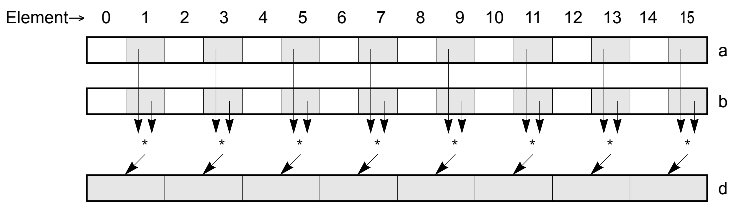
vec_mulo

Vector Multiply Odd

d = vec_mulo(**a**,**b**)

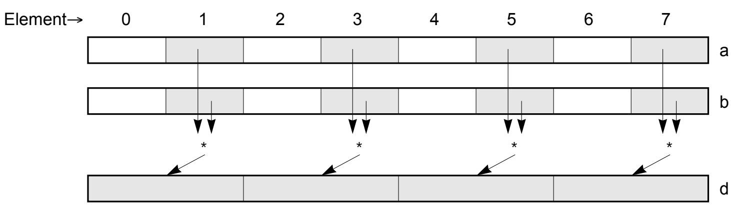
```
n ← number of elements in d
do i=0 to n-1
d_i      ← a_2i+1 * b_2i+1
end
```

Each element of the result is the product of the corresponding low half-width elements of a and b. The even elements of a and b are ignored. The valid combinations of argument types and the corresponding result types for `d = vec_mulo(a,b)` are shown in Figure 4-83 and Figure 4-84.



d	a	b	maps to
vector unsigned short	vector unsigned char	vector unsigned char	vmuloub d,a,b
vector signed short	vector signed char	vector signed char	vmulosb d,a,b

Figure 4-83. Odd Multiply of Eight Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned int	vector unsigned short	vector unsigned short	vmulouh d,a,b
vector signed int	vector signed short	vector signed short	vmulosh d,a,b

Figure 4-84. Odd Multiply of Four Integer Elements (16-Bit)

vec_nmsub

vec_nmsub

Vector Negative Multiply Subtract

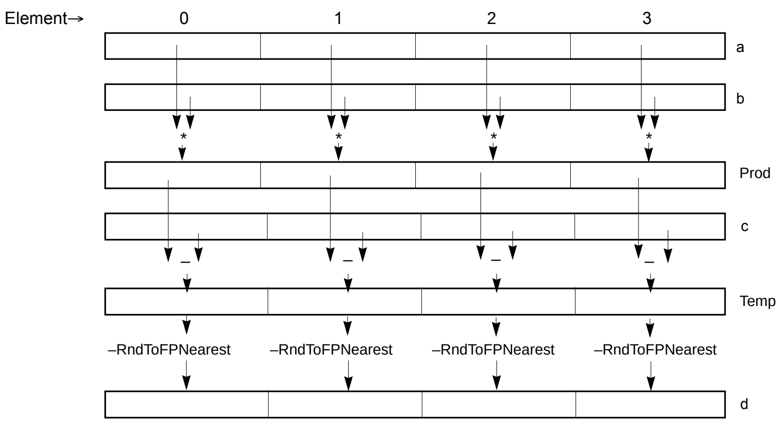
d = vec_nmsub(**a**,**b**,**c**)

```
do i=0 to 3
di ← -RndToFPNearest(ai * bi - ci)
end
```

Each element of the result is the negative of the difference of the corresponding element of c and the product of the corresponding elements of a and b.

For `vector float` argument types, if VSCR[NJ] is set, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

The valid argument types and the corresponding result type for `d = vec_nmsub(a,b,c)` are shown in Figure 4-85.



d	a	b	c	maps to
vector float	vector float	vector float	vector float	vnmsubfp d,a,b,c

Figure 4-85. Negative Multiply-Subtract of Four Floating-Point Elements (32-Bit)

vec_nor vec_nor

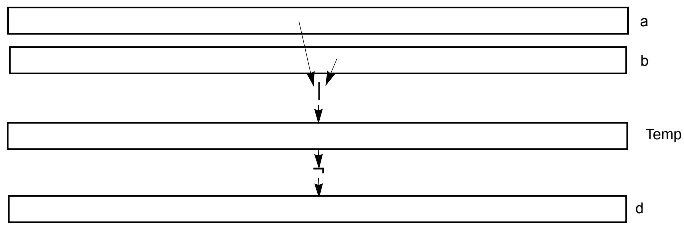
Vector Logical NOR

d = vec_nor(**a**,**b**)

d ← ¬(**a** | **b**)

Each bit of the result is the logical NOR of the corresponding bits of a and b.

The valid combinations of argument types and the corresponding result types for **d** = vec_nor(**a**,**b**) are shown in Figure 4-86.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vnor d,a,b
vector signed char	vector signed char	vector signed char	
vector bool char	vector bool char	vector bool char	
vector unsigned short	vector unsigned short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
vector bool short	vector bool short	vector bool short	
vector unsigned int	vector unsigned int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
vector bool int	vector bool int	vector bool int	
vector float	vector float	vector float	

Figure 4-86. Logical Bit-Wise NOR

vec_or

vec_or

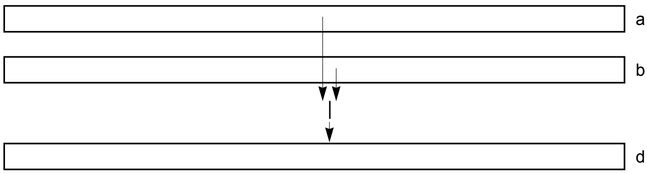
Vector Logical OR

d = vec_or(**a**,**b**)

`d ← a | b`

Each bit of the result is the logical OR of the corresponding bits of a and b.

The valid combinations of argument types and the corresponding result types for `d = vec_or(a,b)` are shown in Figure 4-87.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vor d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector signed char	
vector bool char	vector bool char	vector bool char	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector signed short	
vector bool short	vector bool short	vector bool short	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector signed int	
vector bool int	vector bool int	vector bool int	
vector float	vector bool int	vector float	
	vector float	vector bool int	
	vector float	vector float	

Figure 4-87. Logical Bit-Wise OR

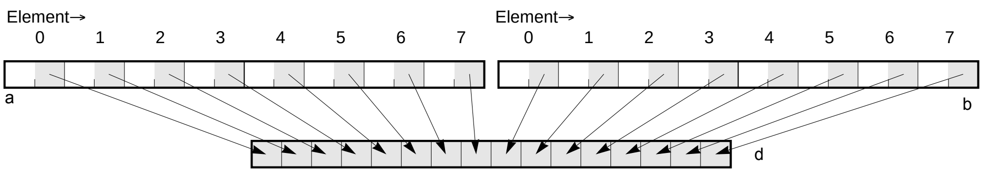
vec_pack vec_pack

Vector Pack

d = vec_pack(**a**,**b**)

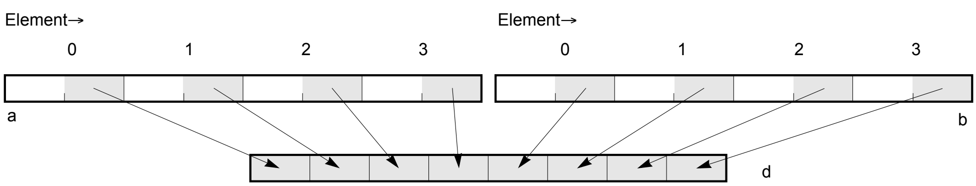
```
n ← number of elements in a
s ← element size in d (64/n)
do i=0 to n-1
d_i ← UIToUImod(a_i,s)
d_i+n ← UIToUImod(b_i,s)
end
```

Each high element of the result is the truncation of the corresponding wider element of a. Each low element of the result is the truncation of the corresponding wider element of b. The valid combinations of argument types and the corresponding result types for `d = vec_pack(a,b)` are shown in Figure 4-88 and Figure 4-89.



d	a	b	maps to
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhum d,a,b
vector signed char	vector signed short	vector signed short	
vector bool char	vector bool short	vector bool short	

Figure 4-88. Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned int	vector unsigned int	vpkuwum d,a,b
vector signed short	vector signed int	vector signed int	
vector bool short	vector bool int	vector bool int	

Figure 4-89. Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vec_packpx

vec_packpx

Vector Pack Pixel

d = vec_packpx(**a**,**b**)

```
do i=0 to 3
di       ← ai[7] ‖ ai[8:12] ‖ ai[16:20] ‖ ai[24:28]
di+4     ← bi[7] ‖ bi[8:12] ‖ bi[16:20] ‖ bi[24:28]
end
```

Each high element of the result is the packed pixel from the corresponding wider element of a. Each low element of the result is the packed pixel from the corresponding wider element of b.

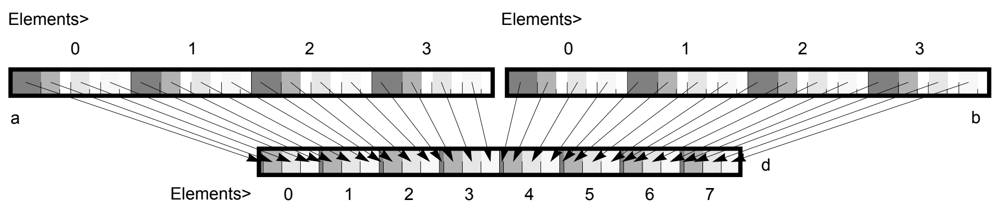
Programming note: Each source word can be considered to be a 32-bit pixel consisting of four 8-bit channels. Each target half-word can be considered to be a 16-bit pixel consisting of one 1-bit channel and three 5-bit channels. A channel can be used to specify the intensity of a particular color, such as red, green, or blue, or to provide other information needed by the application.

The usual transformation from a 32-bit pixel to a 16-bit pixel uses the most significant bit of the 8-bit intensity channel. This operation uses the least significant bit. To use the most significant bit, first perform the following operation:

```
(vector unsigned int) vec_rl ((vector unsigned char) a,
                                   (vector unsigned char) (1,0,0,0,1,0,0,0,
                                                           1,0,0,0,1,0,0,0))
```

on each input a and b.

The valid argument types and the corresponding result type for **d** = vec_packpx(**a**,**b**) are shown in Figure 4-90.



d	a	b	maps to
vector pixel	vector unsigned int	vector unsigned int	vpkpx d,a,b

Figure 4-90. Pack Eight Pixel Elements (32-Bit) to Eight Elements (16-Bit)

vec_packs

vec_packs

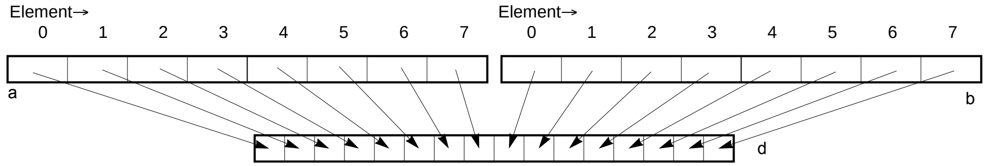
Vector Pack Saturated

d = vec_packs(**a**,**b**)

```
n ← number of elements in a
do i=0 to n-1
d_i ← Saturate(a_i)
d_i+n ← Saturate(b_i)
end
```

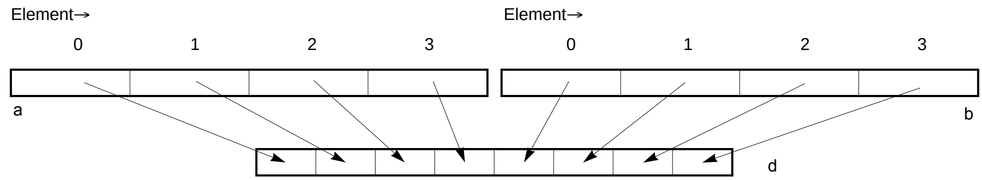
Each high element of the result is the saturated value of the corresponding wider element of a. Each low element of the result is the saturated value of the corresponding wider element of b. If saturation occurs, VSCR[SAT] is set (see Table 4-1).

The valid combinations of argument types and the corresponding result types for **d** = vec_packs(**a**,**b**) are shown in Figure 4-91 and Figure 4-92.



d	a	b	maps to
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhus d,a,b
vector signed char	vector signed short	vector signed short	vpkshss d,a,b

Figure 4-91. Pack Sixteen Integer Elements (16-Bit) to Sixteen Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned int	vector unsigned int	vpkuwus d,a,b
vector signed short	vector signed int	vector signed int	vpkswss d,a,b

Figure 4-92. Pack Eight Integer Elements (32-Bit) to Eight Integer Elements (16-Bit)

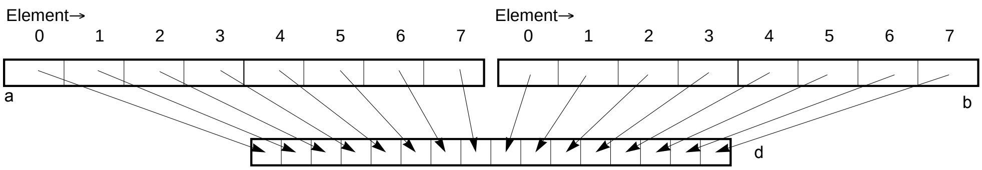
vec_packsu vec_packsu

Vector Pack Saturated Unsigned

d = vec_packsu(**a**,**b**)

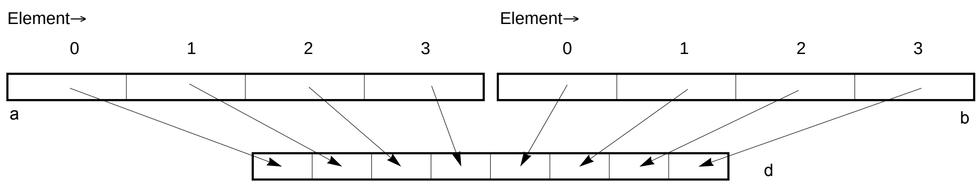
```
n ← number of elements in a
do i=0 to n-1
d_i ← Saturate(a_i)
d_i+n ← Saturate(b_i)
end
```

Each high element of the result is the saturated value of the corresponding wider element of a. Each low element of the result is the saturated value of the corresponding wider element of b. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The result elements are all unsigned. The valid combinations of argument types and the corresponding result types for **`d = vec_packsu(a,b)`** are shown in Figure 4-93 and Figure 4-94.



d	a	b	maps to
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhus d,a,b
vector unsigned char	vector signed short	vector signed short	vpkshus d,a,b

Figure 4-93. Pack Sixteen Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned int	vector unsigned int	vpkuwus d,a,b
vector unsigned short	vector signed int	vector signed int	vpkswus d,a,b

Figure 4-94. Pack Eight Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vec_perm

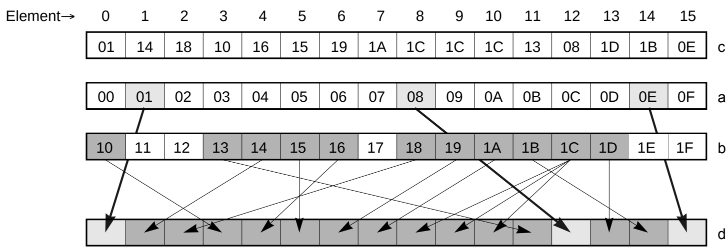
vec_perm

Vector Permute

d = vec_perm(**a**,**b**,**c**)

```
do i=0 to 15
j ← c{i}[4:7]
if c{i}[3] = 0
        then d{i} ← a{j}
        else d{i} ← b{j}
end
```

Each element of the result is selected independently by indexing the byte elements of a and b by the value of the corresponding element of c. For example, 0x1C in **`c`** selects byte 12 in b. The value 0x0C selects byte 12 in a. The valid combinations of argument types and the corresponding result types for **`d`** `= vec_perm(`**`a,b,c`**`)` are shown in Figure 4-95.



d	a	b	c	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	vperm d,a,b,c
vector signed char	vector signed char	vector signed char	vector unsigned char	
vector bool char	vector bool char	vector bool char	vector unsigned char	
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned char	
vector signed short	vector signed short	vector signed short	vector unsigned char	
vector bool short	vector bool short	vector bool short	vector unsigned char	
vector pixel	vector pixel	vector pixel	vector unsigned char	
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned char	
vector signed int	vector signed int	vector signed int	vector unsigned char	
vector bool int	vector bool int	vector bool int	vector unsigned char	
vector float	vector float	vector float	vector unsigned char	

Figure 4-95. Permute Sixteen Integer Elements (8-Bit)

vec_re

Vector Reciprocal Estimate

d = vec_re(**a**)

```
do i=0 to 3
di ← FPRecipEst(ai)
end
```

Each element of the result **d** is an estimate of the reciprocal to the corresponding element of a. For results that are not a +0, –0, +∞, –∞, or QNaN, the estimate has a relative error in precision no greater than one part in 4096, that is:

$$\left|\frac{\text{estimate} - 1/x}{1/x}\right| \le \frac{1}{4096}$$

where x is the value of the element in a. Note that the value placed into the element of d may vary between implementations, and between different executions on the same implementation.

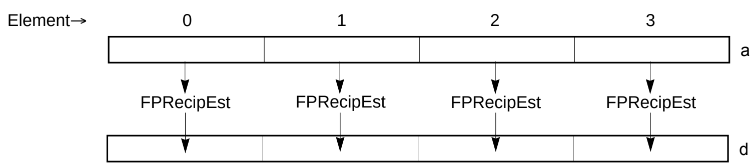
Operation with various special values of the element in a is summarized below.

Table 4-16. Special Value Results of Reciprocal Estimates

a	d
-∞	-0
-0	-∞
+0	+∞
+∞	+0
NaN	QNaN

If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign.

The valid argument type and corresponding result type for `d = vec_re(a)` are shown in Figure 4-96.



d	a	maps to
vector float	vector float	vrefp d,a

Figure 4-96. Reciprocal Estimate of Four Floating-Point Elements (32-Bit)

vec_rl

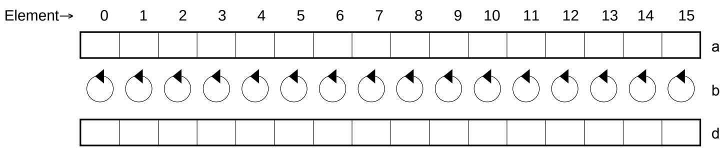
vec_rl

Vector Rotate Left

d = vec_rl(**a**,**b**)

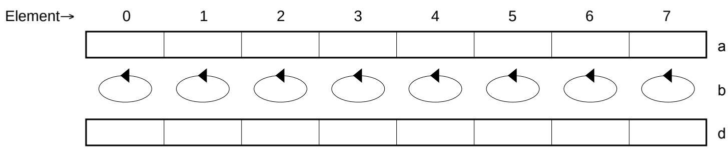
```
n ← number of elements
do i=0 to n-1
di ← ROTL(ai, bi)

end
```

Each element of the result is the result of rotating left the corresponding element of a by the number of bits indicated by the corresponding element of b. The valid combinations of argument types and the corresponding result types for `d = vec_rl(a,b)` are shown in Figure 4-97, Figure 4-98, and Figure 4-99.



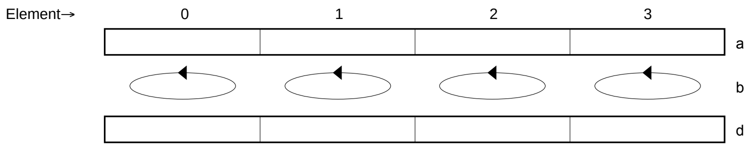
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vrlb d,a,b
vector signed char	vector signed char	vector unsigned char	

Figure 4-97. Left Rotate of Sixteen Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vrlh d,a,b
vector signed short	vector signed short	vector unsigned short	

Figure 4-98. Left Rotate of Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vrlw d,a,b
vector signed int	vector signed int	vector unsigned int	

Figure 4-99. Left Rotate of Four Integer Elements (32-bit)

vec_round

vec_round

Vector Round

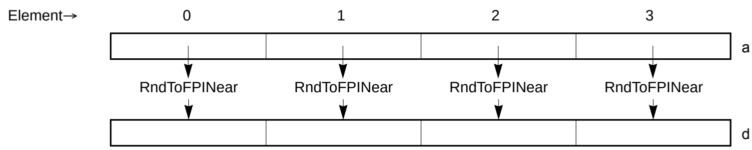
d = vec_round(**a**)

```
do i=0 to 3
d_i ← RndToFPINear(a_i)
end
```

Each element of the result is the nearest representable single-precision floating-point integer to the corresponding element of a, using IEEE Round-to-Nearest mode. If the integers are equally near, rounding is to the even integer.

The operation is independent of VSCR[NJ].

The valid argument type and corresponding result type for `d = vec_round(a)` are shown in Figure 4-100.



d	a	maps to
vector float	vector float	vrfin d,a

Figure 4-100. Round to Nearest of Four Floating-Point Integer Elements (32-Bit)

vec_rsqrte

vec_rsqrte

Vector Reciprocal Square Root Estimate

d = vec_rsqrte(**a**)

```
do i=0 to 3
di ← RecipSQRTEst(ai)
end
```

Each element of the result is an estimate of the reciprocal square root of the corresponding element of a. The single-precision estimate of the reciprocal of the square root of each single-precision element in a is placed into the corresponding word element of d. The estimate has a relative error in precision no greater than one part in 4096, that is:

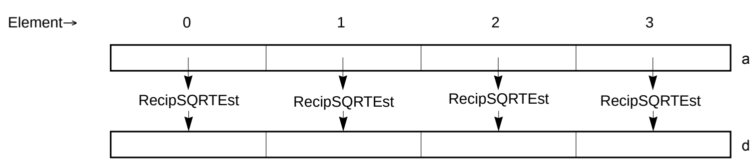
$$\left|\frac{\text{estimate} - 1/\sqrt{x}}{1/\sqrt{x}}\right| \le \frac{1}{4096}$$

where x is the value of the element in a. The value placed into the element of d may vary between implementations and between different executions on the same implementation. If VSCR[NJ] = 1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign. Operation with various special values of the element in a is summarized below.

Table 4-17. Special Value Results of Reciprocal Square Root Estimates

a	d
-∞	QNaN
less than 0	QNaN
-0	-∞
+0	+∞
+∞	+0
NaN	QNaN

The valid argument type and corresponding result type for `d = vec_rsqrte(a)` are shown in Figure 4-101.



d	a	maps to
vector float	vector float	vrsqrtefp d,a

Figure 4-101. Reciprocal Square Root Estimate of Four Floating-Point Elements (32-Bit)

vec_sel

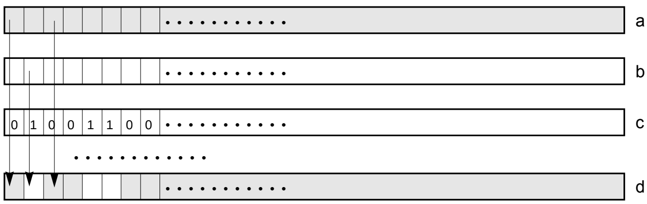
vec_sel

Vector Select

d = vec_sel(**a**,**b**,**c**)

```
do i=0 to 127
if c_i=0
        then d[i] ← a[i]
        else d[i] ← b[i]
end
```

Each bit of the result is the corresponding bit of a if the corresponding bit of c is 0. Otherwise, it is the corresponding bit of b. The valid combinations of argument types and the corresponding result types for `d = vec_sel(a,b,c)` are shown in Figure 4-102.



d	a	b	c	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	vsel d,a,b,c
	vector unsigned char	vector unsigned char	vector bool char	
vector signed char	vector signed char	vector signed char	vector unsigned char	
	vector signed char	vector signed char	vector bool char	
vector bool char	vector bool char	vector bool char	vector unsigned char	
	vector bool char	vector bool char	vector bool char	
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	
	vector unsigned short	vector unsigned short	vector bool short	
vector signed short	vector signed short	vector signed short	vector unsigned short	
	vector signed short	vector signed short	vector bool short	
vector bool short	vector bool short	vector bool short	vector unsigned short	
	vector bool short	vector bool short	vector bool short	
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	
	vector unsigned int	vector unsigned int	vector bool int	
vector signed int	vector signed int	vector signed int	vector unsigned int	
	vector signed int	vector signed int	vector bool int	
vector bool int	vector bool int	vector bool int	vector unsigned int	
	vector bool int	vector bool int	vector bool int	
vector float	vector float	vector float	vector unsigned int	
	vector float	vector float	vector bool int	

Figure 4-102. Bit-Wise Conditional Select of Vector Contents (128-bit)

vec_sl

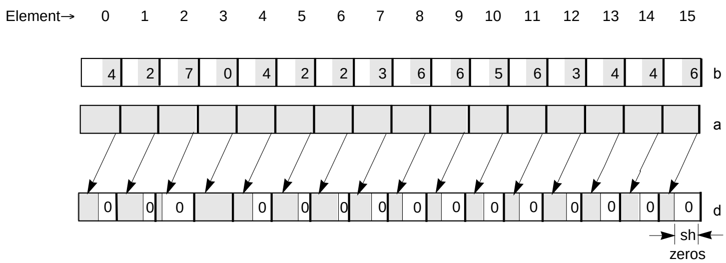
vec_sl

Vector Shift Left

d = vec_sl(**a**,**b**)

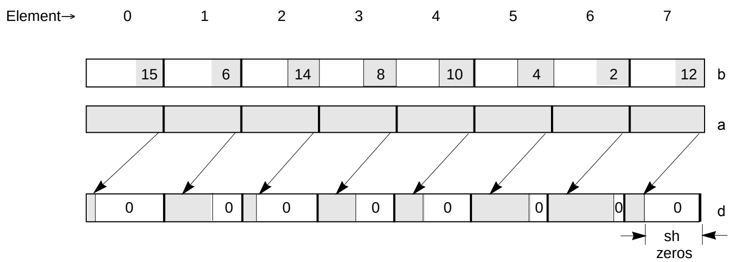
```
n ← number of elements
s ← 128/n
do i=0 to n-1
di ← ShiftLeft(ai,mod(bi,s))
end
```

Each element in d is the result of shifting the corresponding element of a left by the number of bits of the corresponding element of b. The valid combinations of argument types and the corresponding result types for `d = vec_sl(a,b)` are shown in Figure 4-103, Figure 4-104, and Figure 4-105.



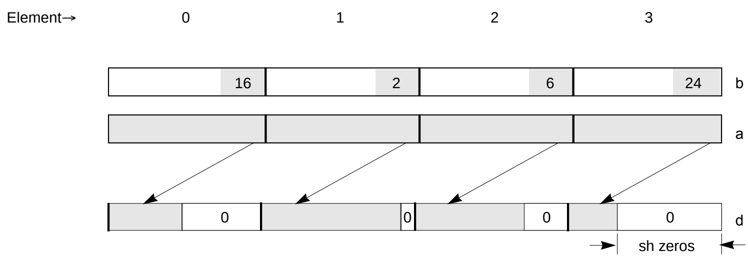
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vslb d,a,b
vector signed char	vector signed char	vector unsigned char	

Figure 4-103. Shift Bits Left in Sixteen Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vslh d,a,b
vector signed short	vector signed short	vector unsigned short	

Figure 4-104. Shift Bits Left in Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vslw d,a,b
vector signed int	vector signed int	vector unsigned int	

Figure 4-105. Shift Bits Left in Four Integer Elements (32-Bit)

vec_sld

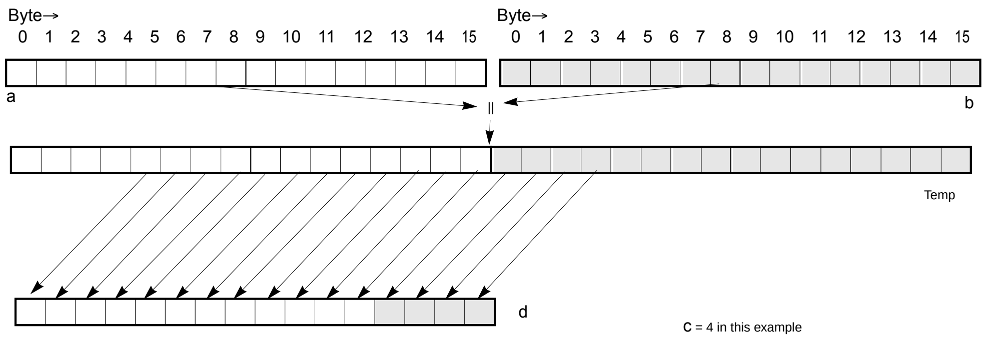
vec_sld

Vector Shift Left Double

d = vec_sld(**a**,**b**,**c**)

```
do i=0 to 15
if (i+c) < 16
        then d{i} ← a{i+c}
        else d{i} ← b{i+c-16}
end
```

The result is obtained by selecting the top 16 bytes obtained by shifting left (unsigned) by the value of c bytes a 32-byte quantity formed by catenating a with b. The valid combinations of argument types and the corresponding result types for **d** = vec_sld(**a**,**b**,**c**) are shown in Figure 4-106.



d	a	b	c	maps to
vector unsigned char	vector unsigned char	vector unsigned char	4-bit unsigned literal	vsldoi d,a,b,c
vector signed char	vector signed char	vector signed char	4-bit unsigned literal	
vector unsigned short	vector unsigned short	vector unsigned short	4-bit unsigned literal	
vector signed short	vector signed short	vector signed short	4-bit unsigned literal	
vector pixel	vector pixel	vector pixel	4-bit unsigned literal	
vector unsigned int	vector unsigned int	vector unsigned int	4-bit unsigned literal	
vector signed int	vector signed int	vector signed int	4-bit unsigned literal	
vector float	vector float	vector float	4-bit unsigned literal	

Figure 4-106. Bit-Wise Conditional Select of Vector Contents (128-bit)

vec_sll vec_sll

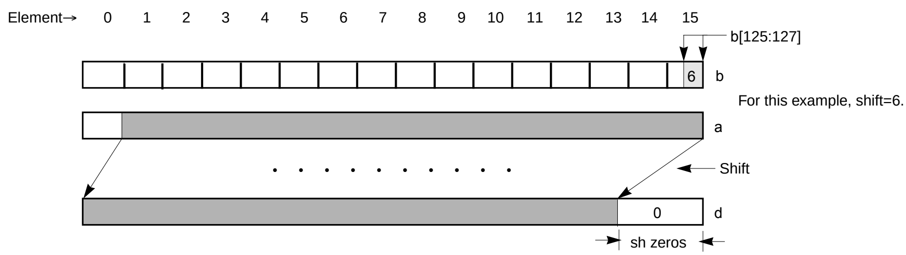
Vector Shift Left Long

d = vec_sll(**a**,**b**)

```
m ← b[125:127]
If each bi[5:7] = m, where i ranges from 0 to 14
then d ← ShiftLeft(a,m)
else d ← Undefined
```

The result is obtained by shifting a left by a number of bits specified by the last 3 bits of the last element of b. The valid combinations of argument types and the corresponding result types for `d = vec_sll(a,b)` are shown in Figure 4-107.

Note that the three low-order bits of all byte elements in b must be the same; otherwise the value placed into d is undefined.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vsl d,a,b
	vector unsigned char	vector unsigned short	
	vector unsigned char	vector unsigned int	
vector signed char	vector signed char	vector unsigned char	
	vector signed char	vector unsigned short	
	vector signed char	vector unsigned int	
vector bool char	vector bool char	vector unsigned char	
	vector bool char	vector unsigned short	
	vector bool char	vector unsigned int	
vector unsigned short	vector unsigned short	vector unsigned char	
	vector unsigned short	vector unsigned short	
	vector unsigned short	vector unsigned int	
vector signed short	vector signed short	vector unsigned char	
	vector signed short	vector unsigned short	
	vector signed short	vector unsigned int	
vector bool short	vector bool short	vector unsigned char	
	vector bool short	vector unsigned short	
	vector bool short	vector unsigned int	
vector pixel	vector pixel	vector unsigned char	
	vector pixel	vector unsigned short	
	vector pixel	vector unsigned int	
vector unsigned int	vector unsigned int	vector unsigned char	
	vector unsigned int	vector unsigned short	
	vector unsigned int	vector unsigned int	
vector signed int	vector signed int	vector unsigned char	
	vector signed int	vector unsigned short	
	vector signed int	vector unsigned int	
vector bool int	vector bool int	vector unsigned char	
	vector bool int	vector unsigned short	
	vector bool int	vector unsigned int	

Figure 4-107. Shift Bits Left in Vector (128-Bit)

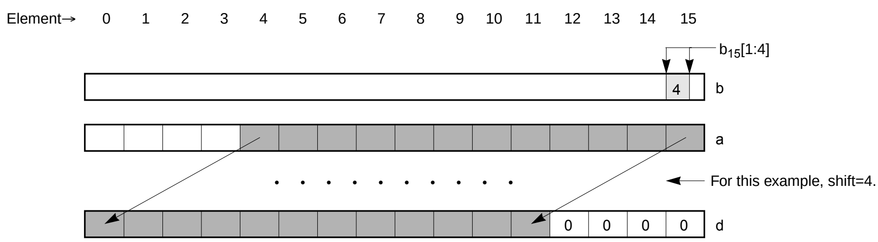
vec_slo vec_slo

Vector Shift Left by Octet

d = vec_slo(**a**,**b**)

```
m ← b15[1:4]
do i=0 to 15
j ← i + m
if j < 16
        then d{i} ← a{j}
        else d{i} ← 0
end
```

The contents of a are shifted left by the number of bytes specified by bits $b_{15}[1:4]$; only these 4 bits in b are significant for the shift value. Bytes shifted out of byte 0 are lost. Zeros are supplied to the vacated bytes on the right. The result is placed into d. The valid combinations of argument types and the corresponding result types for **d** = vec_slo(**a**,**b**) are shown in Figure 4-108.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vslo d,a,b
	vector unsigned char	vector signed char	
vector signed char	vector signed char	vector unsigned char	
	vector signed char	vector signed char	
vector unsigned short	vector unsigned short	vector unsigned char	
	vector unsigned short	vector signed char	
vector signed short	vector signed short	vector unsigned char	
	vector signed short	vector signed char	
vector pixel	vector pixel	vector unsigned char	
	vector pixel	vector signed char	
vector unsigned int	vector unsigned int	vector unsigned char	
	vector unsigned int	vector signed char	
vector signed int	vector signed int	vector unsigned char	
	vector signed int	vector signed char	
vector float	vector float	vector unsigned char	
	vector float	vector signed char	

Figure 4-108. Left Byte Shift of Vector (128-Bit)

vec_splat

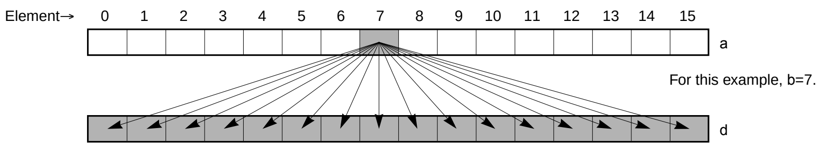
vec_splat

Vector Splat

d = vec_splat(**a**,**b**)

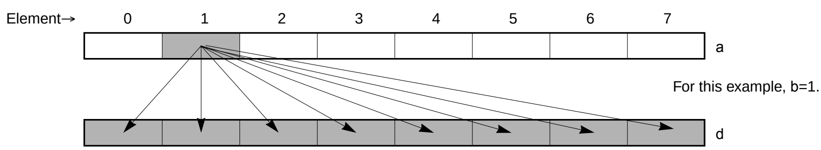
```
n ← number of elements
do i=0 to n-1
j ← mod(b,n)
d_i ← a_j
end
```

Each element of the result is component b of a. The valid combinations of argument types and the corresponding result types for `d = vec_splat(a,b)` are shown in Figure 4-109, Figure 4-110, and Figure 4-111.



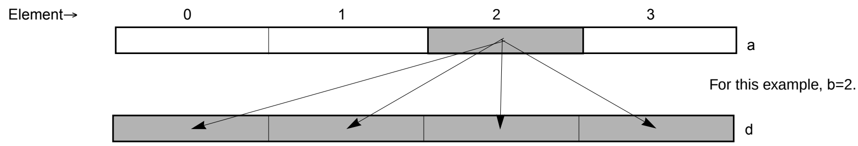
d	a	b	maps to
vector unsigned char	vector unsigned char	5-bit unsigned literal	vspltb d,a,b
vector signed char	vector signed char	5-bit unsigned literal	
vector bool char	vector bool char	5-bit unsigned literal	

Figure 4-109. Copy Contents to Sixteen Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	5-bit unsigned literal	vsplth d,a,b
vector signed short	vector signed short	5-bit unsigned literal	
vector bool short	vector bool short	5-bit unsigned literal	
vector pixel	vector pixel	5-bit unsigned literal	

Figure 4-110. Copy Contents to Eight Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	5-bit unsigned literal	vspltw d,a,b
vector signed int	vector signed int	5-bit unsigned literal	
vector bool int	vector bool int	5-bit unsigned literal	
vector float	vector float	5-bit unsigned literal	

Figure 4-111. Copy Contents to Four Integer Elements (32-Bit)

vec_splat_s8

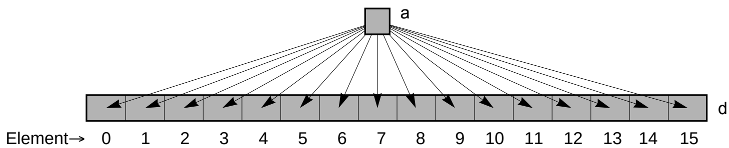
vec_splat_s8

Vector Splat Signed Byte

d = vec_splat_s8(**a**)

```
do i=0 to 15
di ← SignExtend(a)
end
```

Each element of the result is the value obtained by sign-extending a. This permits values ranging from -16 to 15 only. The valid argument type and corresponding result type for **`d`** `= vec_splat_s8(`**`a`**`)` are shown in Figure 4-112.



d	a	maps to
vector signed char	5-bit signed literal	vspltisb d,a

Figure 4-112. Copy Value into Sixteen Signed Integer Elements (8-Bit)

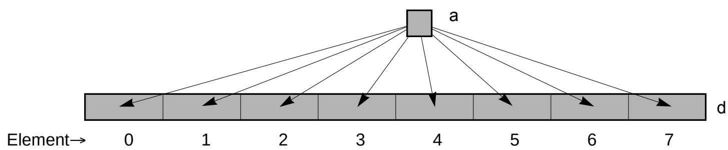
vec_splat_s16 vec_splat_s16

Vector Splat Signed Half-Word

d = vec_splat_s16(**a**)

```
do i=0 to 7
d_i ← SignExtend(a)
end
```

Each element of the result is the value obtained by sign-extending a. This permits values ranging from -16 to 15 only. The valid argument type and corresponding result type for `d = vec_splat_s16(a)`, tare shown in Figure 4-113.



d	a	maps to
vector signed short	5-bit signed literal	vspltish d,a

Figure 4-113. Copy Value into Eight Signed Integer Elements (16-Bit)

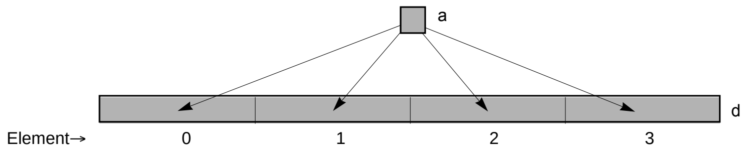
vec_splat_s32 vec_splat_s32

Vector Splat Signed Word

d = vec_splat_s32(**a**)

```
do i=0 to 3
di ← SignExtend(a)
end
```

Each element of the result is the value obtained by sign-extending a. This permits values ranging from -16 to 15 only. The valid argument type are corresponding result type for `d = vec_splat_s32(a)` are shown in Figure 4-114.



d	a	maps to
vector signed int	5-bit signed literal	vspltisw d,a

Figure 4-114. Copy Value into Four Signed Integer Elements (32-Bit)

vec_splat_u8

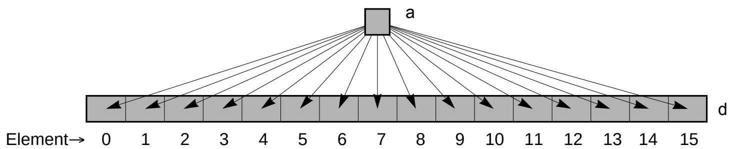
vec_splat_u8

Vector Splat Unsigned Byte

d = vec_splat_u8(**a**)

```
do i=0 to 15
d_i ← SignExtend(a)
end
```

Each element of the result is the value obtained by sign-extending a and casting it to an unsigned char value. Each element of **d** is set to 256*sign(a) + a, where sign(a) is 0 for non-negative a and 1 for negative a. The valid argument type and corresponding result type for `d = vec_splat_u8(a)` are shown in Figure 4-115. It is necessary to use the generic name, since the specific operation `vec_vspltisb` returns a `vector signed char` value.



d	a	maps to
vector unsigned char	5-bit signed literal	vspltisb d,a

Figure 4-115. Copy Value into Sixteen Signed Integer Elements (8-Bit)

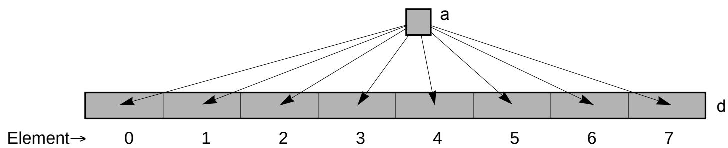
vec_splat_u16 vec_splat_u16

Vector Splat Unsigned Half-Word

d = vec_splat_u16(**a**)

```
do i=0 to 7
di ← SignExtend(a)
end
```

Each element of the result is the value obtained by sign-extending a and casting it to an unsigned short value. Each element of d is set to 65536*sign(a) + a, where sign(a) is 0 for non-negative a and 1 for negative a. The valid argument type and corresponding result type for `d = vec_splat_u16(a)` are shown in Figure 4-116. It is necessary to use the generic name, since the specific operation `vec_vspltish` returns a `vector signed short` value.



d	a	maps to
vector unsigned short	5-bit signed literal	vspltish d,a

Figure 4-116. Copy Value into Eight Signed Integer Elements (16-Bit)

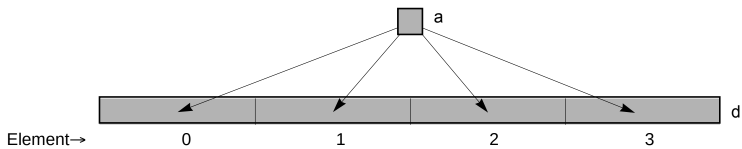
vec_splat_u32 vec_splat_u32

Vector Splat Unsigned Word

d = vec_splat_u32(**a**)

```
do i=0 to 3
di ← SignExtend(a)
end
```

Each element of the result is the value obtained by sign-extending a. and casting it to an unsigned int value. Each element of d is set to 4294967296*sign(a) + a, where sign(a) is 0 for non-negative a and 1 for negative a. The valid argument type and corresponding result type for `d = vec_splat_u32(a)` areshown in Figure 4-117. It is necessary to use the generic name, since the specific operation `vec_vspltisw` returns a `vector signed int` value.



d	a	maps to
vector unsigned int	5-bit signed literal	vspltisw d,a

Figure 4-117. Copy Value into Four Signed Integer Elements (32-Bit)

vec_sr

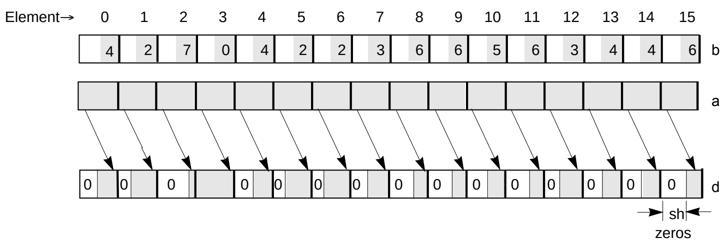
vec_sr

Vector Shift Right

d = vec_sr(**a**,**b**)

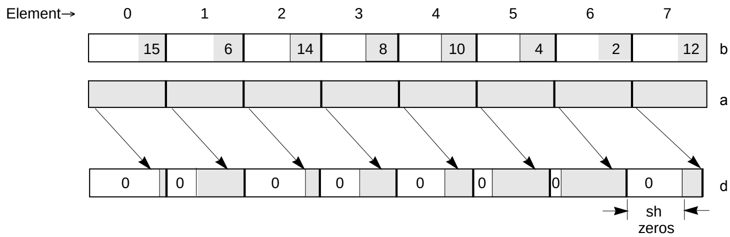
```
n ← number of elements
s ← 128/n
do i=0 to n-1
d_i ← ShiftRight(a_i,mod(b_i,s))
end
```

Each element of the result is the result of shifting the corresponding element of a right by the number of bits of the corresponding element of b. Zero bits are shifted in from the left for both signed and unsigned argument types. The valid combinations of argument types and the corresponding result types for **d** = vec_sr(**a**,**b**) are shown in Figure 4-118, Figure 4-119, and Figure 4-120.



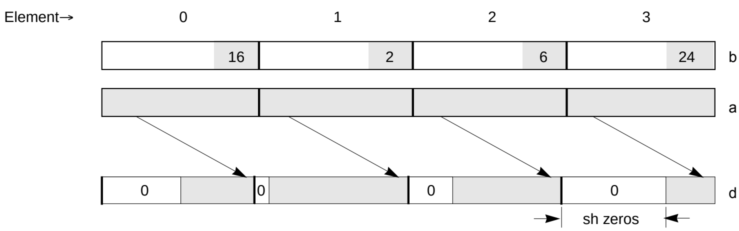
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vsrb d,a,b
vector signed char	vector signed char	vector unsigned char	

Figure 4-118. Shift Bits Right in Sixteen Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vsrh d,a,b
vector signed short	vector signed short	vector unsigned short	

Figure 4-119. Shift Bits Right in Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vsrw d,a,b
vector signed int	vector signed int	vector unsigned int	

Figure 4-120. Shift Bits Right in Four Integer Elements (32-Bit)

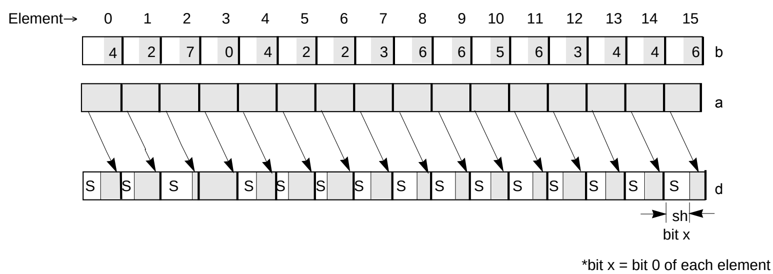
vec_sra vec_sra

Vector Shift Right Algebraic

d = vec_sra(**a**,**b**)

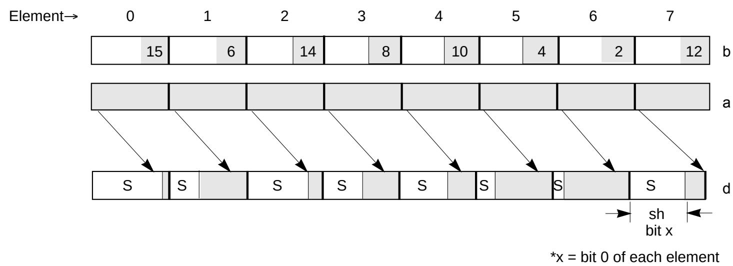
```
n ← number of elements
s ← 128/n
do i=0 to n-1
d_i ← ShiftRightA(a_i,mod(b_i,s))
end
```

Each element of the result is the result of shifting the corresponding element of a right by the number of bits of the corresponding element of b. Copies of the sign bit are shifted in from the left for both signed and unsigned argument types. The valid combinations of argument types and the corresponding result types for `d = vec_sra(a,b)` are shown in Figure 4-121, Figure 4-122, and Figure 4-123.



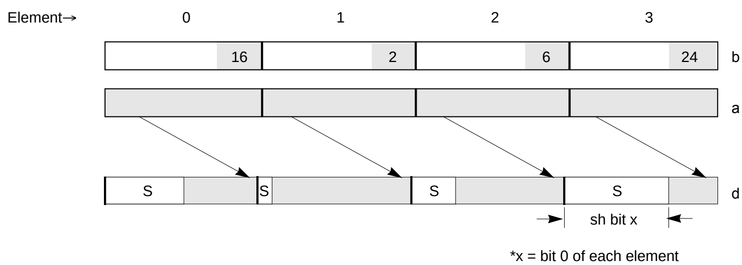
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vsrab d,a,b
vector signed char	vector signed char	vector unsigned char	

Figure 4-121. Shift Bits Right in Sixteen Integer Elements (8-Bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vsrah d,a,b
vector signed short	vector signed short	vector unsigned short	

Figure 4-122. Shift Bits Right in Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vsraw d,a,b
vector signed int	vector signed int	vector unsigned int	

Figure 4-123. Shift Bits Right in Four Integer Elements (32-Bit)

vec_srl

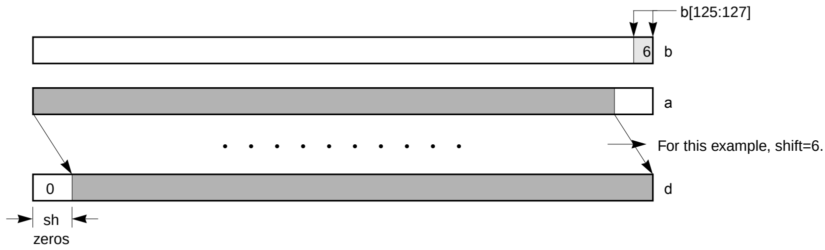
Vector Shift Right Long

d = vec_srl(**a**,**b**)

```
m ← b[125:127]
if each b_i[5:7] = m, where i ranges from 0 to 14
then d ← ShiftRight(a,m)
else d ← Undefined
```

The result is obtained by shifting a right by a number of bits specified by the last 3 bits of the last element of b. The valid combinations of argument types and the corresponding result types for `d = vec_srl(a,b)` are shown in Figure 4-124.

Note that the low-order 3 bits of all byte elements in b must be the same; otherwise the value placed into d is undefined.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vsr d,a,b
	vector unsigned char	vector unsigned short	
	vector unsigned char	vector unsigned int	
vector signed char	vector signed char	vector unsigned char	
	vector signed char	vector unsigned short	
	vector signed char	vector unsigned int	
vector bool char	vector bool char	vector unsigned char	
	vector bool char	vector unsigned short	
	vector bool char	vector unsigned int	
vector unsigned short	vector unsigned short	vector unsigned char	
	vector unsigned short	vector unsigned short	
	vector unsigned short	vector unsigned int	
vector signed short	vector signed short	vector unsigned char	
	vector signed short	vector unsigned short	
	vector signed short	vector unsigned int	
vector bool short	vector bool short	vector unsigned char	
	vector bool short	vector unsigned short	
	vector bool short	vector unsigned int	
vector pixel	vector pixel	vector unsigned char	
	vector pixel	vector unsigned short	
	vector pixel	vector unsigned int	
vector unsigned int	vector unsigned int	vector unsigned char	
	vector unsigned int	vector unsigned short	
	vector unsigned int	vector unsigned int	
vector signed int	vector signed int	vector unsigned char	
	vector signed int	vector unsigned short	
	vector signed int	vector unsigned int	
vector bool int	vector bool int	vector unsigned char	
	vector bool int	vector unsigned short	
	vector bool int	vector unsigned int	

Figure 4-124. Shift Bits Right in Vector (128-Bit)

vec_sro

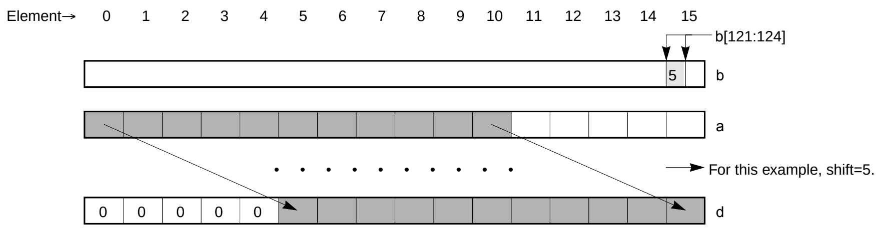
vec_sro

Vector Shift Right by Octet

d = vec_sro(**a**,**b**)

```
m ← b[121:124]
do i=0 to 15
j ← i - m
if j ≥ 0
        then d{i} ←a{j}
        else d{i} ←0
end
```

The result is obtained by shifting (unsigned) a right by a number of bytes specified by the shifting the value of the last element of b by 3 bits. The valid combinations of argument types and the corresponding result types for `d = vec_sro(a,b)` are shown in Figure 4-125.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vsro d,a,b
	vector unsigned char	vector signed char	
vector signed char	vector signed char	vector unsigned char	
	vector signed char	vector signed char	
vector unsigned short	vector unsigned short	vector unsigned char	
	vector unsigned short	vector signed char	
vector signed short	vector signed short	vector unsigned char	
	vector signed short	vector signed char	
vector pixel	vector pixel	vector unsigned char	
	vector pixel	vector signed char	
vector unsigned int	vector unsigned int	vector unsigned char	
	vector unsigned int	vector signed char	
vector signed int	vector signed int	vector unsigned char	
	vector signed int	vector signed char	
vector float	vector float	vector unsigned char	
	vector float	vector signed char	

Figure 4-125. Right Byte Shift of Vector (128-Bit)

vec_st vec_st

Vector Store Indexed

vec_st(**a**,**b**,**c**)

```
EA ← BoundAlign((b + c), 16)
MEM(EA,16) ← a
```

Each operation performs a 16-byte store of the value of a at a 16-byte aligned address. The b is taken to be an integer value, while c is a pointer. BoundAlign(b+c,16) is the largest value less than or equal to a b+c that is a multiple of 16. This is not, by itself, an acceptable way to store aligned data to unaligned addresses. This store is the one that is generated for a storing dereference of a pointer to a vector type. Plain `char *` is excluded in the mapping for c. The valid combinations of argument types for `vec_st(a,b,c)` are shown in Table 4-18. The result type is `void`.

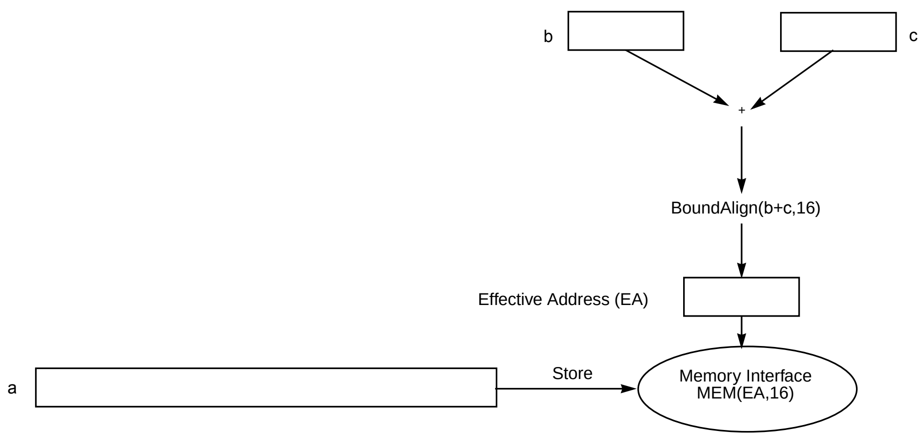


Figure 4-126. Vector Store Indexed

Table 4-18. vec_st—Vector Store Indexed Argument Types

a	b	c	Maps to
vector unsigned char	any integral type	vector unsigned char *	stvx a,b,c
vector unsigned char	any integral type	unsigned char *	
vector signed char	any integral type	vector signed char *	
vector signed char	any integral type	signed char *	
vector bool char	any integral type	vector bool char *	
vector bool char	any integral type	unsigned char *	
vector bool char	any integral type	signed char *	
vector unsigned short	any integral type	vector unsigned short *	
vector unsigned short	any integral type	unsigned short *	
vector signed short	any integral type	vector signed short *	
vector signed short	any integral type	short *	
vector bool short	any integral type	vector bool short *	
vector bool short	any integral type	unsigned short *	
vector bool short	any integral type	short *	
vector pixel	any integral type	vector pixel short *	
vector pixel	any integral type	unsigned short *	
vector pixel	any integral type	short *	
vector unsigned int	any integral type	vector unsigned int *	
vector unsigned int	any integral type	unsigned int *	
vector signed int	any integral type	vector signed int *	
vector signed int	any integral type	int *	
vector bool int	any integral type	vector bool int *	
vector bool int	any integral type	unsigned int *	
vector bool int	any integral type	int *	
vector float	any integral type	vector float *	
vector float	any integral type	float *	

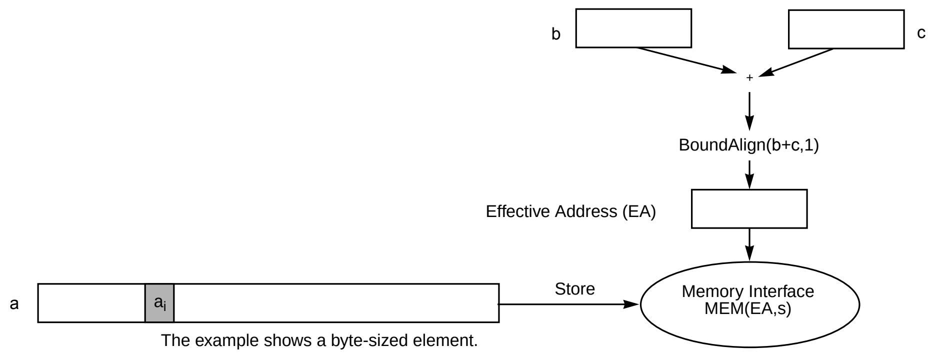
vec_ste vec_ste

Vector Store Element Indexed

vec_ste(**a**,**b**,**c**)

```
s ← 16/(number of elements)
EA ← BoundAlign (b + c,s)
i ← mod(EA,16)/s
MEM(EA,s) ← a_i
```

A single element of a is stored at the effective address. BoundAlign(b+c,s) is the largest value less than or equal to b+c that is a multiple of s, where s is 1 for `char` pointers, 2 for `short` pointers, and 4 for `int` or `float` pointers. The element stored is the one whose position in the register matches the position of the adjusted address relative to 16-byte alignment (A16). If you do not know the alignment of the sum of b and c, you will not know which element is stored. Plain `char *` is excluded in the mapping for c. The valid combinations of argument types for `vec_ste(a,b,c)` are shown in Figure 4-127. The result type is `void`.



a	b	c	Maps to
vector unsigned char	any integral type	unsigned char *	stvebx a,b,c
vector signed char	any integral type	signed char *	
vector bool char	any integral type	unsigned char *	
vector bool char	any integral type	signed char *	
vector unsigned short	any integral type	unsigned short *	stvehx a,b,c
vector signed short	any integral type	short *	
vector bool short	any integral type	unsigned short *	
vector bool short	any integral type	short *	
vector pixel	any integral type	unsigned short *	
vector pixel	any integral type	short *	
vector unsigned int	any integral type	unsigned int *	stvewx a,b,c
vector unsigned int	any integral type	unsigned int *	
vector signed int	any integral type	int *	
vector signed int	any integral type	int *	
vector bool int	any integral type	unsigned int *	
vector bool int	any integral type	unsigned int *	
vector bool int	any integral type	int *	
vector bool int	any integral type	int *	
vector float	any integral type	float *	

Figure 4-127. Vector Store Element

vec_stl vec_stl

Vector Store Indexed LRU

vec_stl(**a**,**b**,**c**)

```
EA ← BoundAlign(b + c, 16)
MEM(EA,16) ← a
```

Each operation performs a 16-byte store of the value of a at a 16-byte aligned address. The b is taken to be an integer value, while c is a pointer. BoundAlign(b+c,16) is the largest value less than or equal to a b+c that is a multiple of 16. This is not, by itself, an acceptable way to store aligned data to unaligned addresses. The cache line stored into is marked Least Recently Used (LRU). Plain `char *` is excluded in the mapping for c. The valid combinations of argument types for `vec_stl(a,b,c)` are shown in Table 4-19. The result type is `void`.

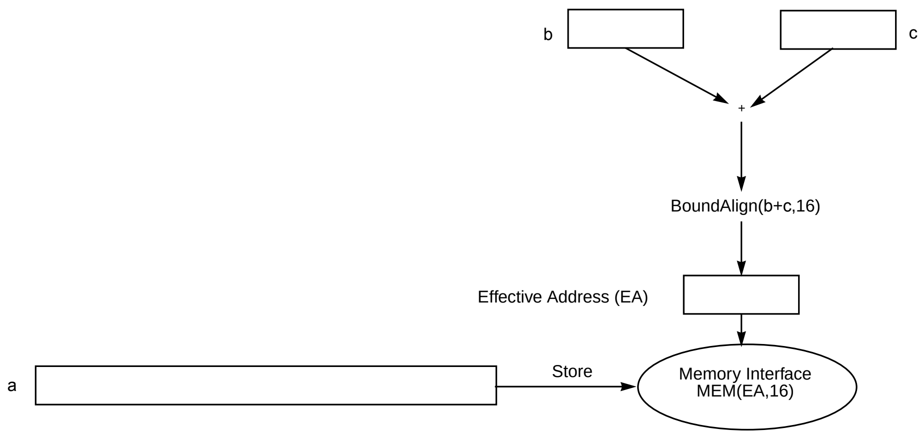


Figure 4-128. Vector Store Indexed LRU

Table 4-19vec_stl—Vector Store Index Argument Types

a	b	c	Maps to
vector unsigned char	any integral type	vector unsigned char *	stvxl a,b,c
vector unsigned char	any integral type	unsigned char *	
vector signed char	any integral type	vector signed char *	
vector signed char	any integral type	signed char *	
vector bool char	any integral type	vector bool char *	
vector bool char	any integral type	unsigned char *	
vector bool char	any integral type	signed char *	
vector unsigned short	any integral type	vector unsigned short *	
vector unsigned short	any integral type	unsigned short *	
vector signed short	any integral type	vector signed short *	
vector signed short	any integral type	short *	
vector bool short	any integral type	vector bool short *	
vector bool short	any integral type	unsigned short *	
vector bool short	any integral type	short *	
vector pixel	any integral type	vector pixel *	
vector pixel	any integral type	unsigned short *	
vector pixel	any integral type	short *	
vector unsigned int	any integral type	vector unsigned int *	
vector unsigned int	any integral type	unsigned int *	
vector signed int	any integral type	vector signed int *	
vector signed int	any integral type	int *	
vector bool int	any integral type	vector bool int *	
vector bool int	any integral type	unsigned int *	
vector bool int	any integral type	unsigned int *	
vector bool int	any integral type	int *	
vector float	any integral type	vector float *	
vector float	any integral type	float *	

vec_sub

vec_sub

Vector Subtract

d = vec_sub(**a**,**b**)

- Integer Subtract:

```
n ← number of elements
do i=0 to n-1
d_i ← a_i - b_i
end
```

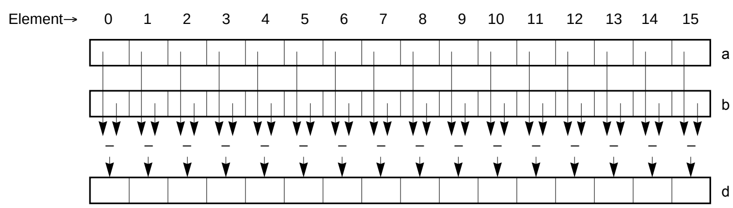
- Floating-Point Subtract:

```
do i=0 to 3
d_i ← a_i -_fp b_i
end
```

Each element of the result is the difference between the corresponding elements of a and b. The arithmetic is modular for integer types.

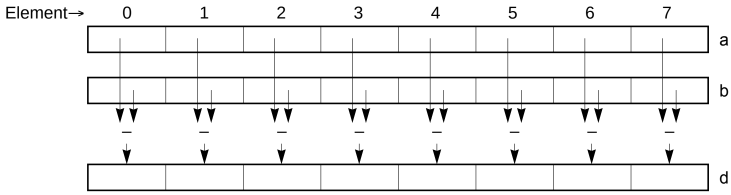
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized `vector float` operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized `vector float` result element truncates to a 0 of the same sign.

The valid combinations of argument types and the corresponding result types for `d = vec_sub(a,b)` are shown in Figure 4-129, Figure 4-130, Figure 4-131, and Figure 4-132.



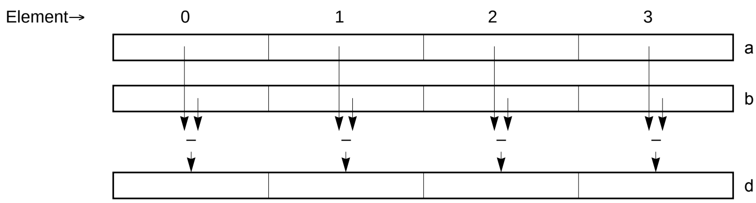
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vsububm d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-129. Subtract Sixteen Integer Elements (8-bit)



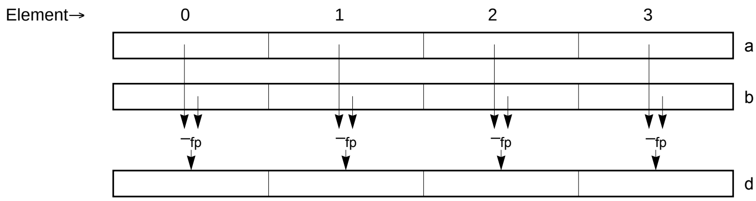
d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vsubuhm d,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-130. Subtract Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vsubuwm d,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-131. Subtract Four Integer Elements (32-bit)



d	a	b	maps to
vector float	vector float	vector float	vsubfp d,a,b

Figure 4-132. Subtract Four Floating-Point Elements (32-bit)

vec_subc

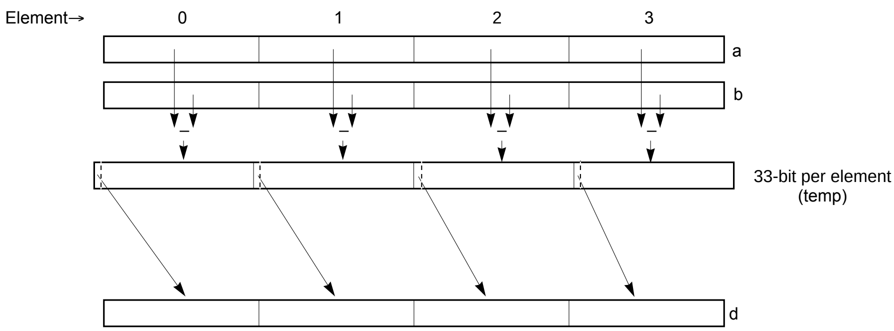
vec_subc

Vector Subtract Carryout

d = vec_subc(**a**,**b**)

```
do i=0 to 3
d_i = BorrowOut(a_i - b_i)
end
```

Each element of b is subtracted from the corresponding element in a. The borrow from each difference is complemented and zero-extended and placed into the corresponding element of d. BorrowOut (a – b) is 0 if a borrow occurred and 1 if no borrow occurred. The valid combination of argument types and the corresponding result type for **`d`** `= vec_subc(`**`a,b`**`)` are shown in Figure 4-133.



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vsubcuw d,a,b

Figure 4-133. Carryout of Four Unsigned Integer Subtracts (32-bit)

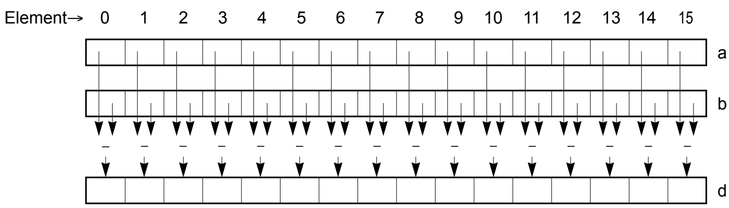
vec_subs vec_subs

Vector Subtract Saturated

d = vec_subs(**a**,**b**)

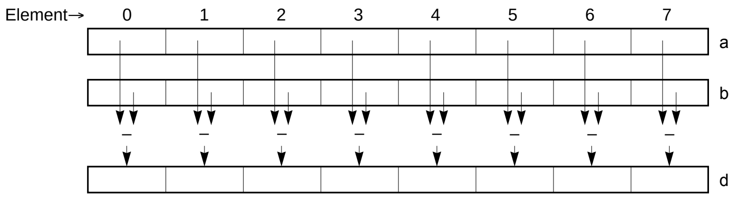
```
n ← number of elements
do i=0 to n-1
di ← Saturate (ai - bi)
end
```

Each element of the result is the saturated difference between the corresponding elements of a and b. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The valid combinations of argument types and the corresponding result types for `d = vec_subs(a,b)` are shown in Figure 4-134, Figure 4-135, and Figure 4-136.



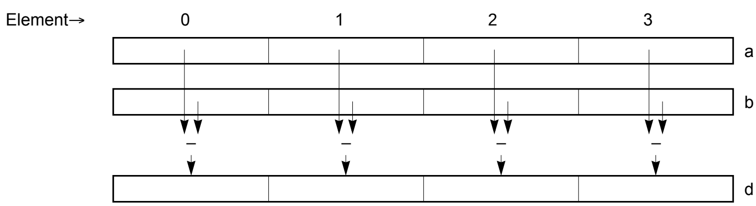
d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vsububs d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	vsubsbs d,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-134. Subtract Saturating Sixteen Integer Elements (8-bit)



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vsubuhs d,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	vsubshs d,a,b
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-135. Subtract Saturating Eight Integer Elements (16-bit)



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vsubuws d,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	vsubsws d,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-136. Subtract Saturating Four Integer Elements (32-bit)

vec_sum4s

vec_sum4s

Vector Sum Across Partial (1/4) Saturated

d = vec_sum4s(**a**,**b**)

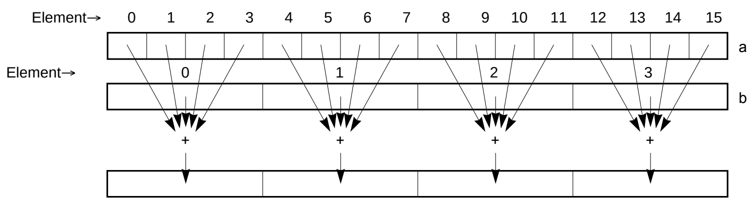
- For a with 8-bit elements:

```
do i=0 to 3
d_i ← Saturate (a_4i+ a_4i+1 + a_4i+2 + a_4i+3 + b_i)
end
```

- For a with 16-bit elements:

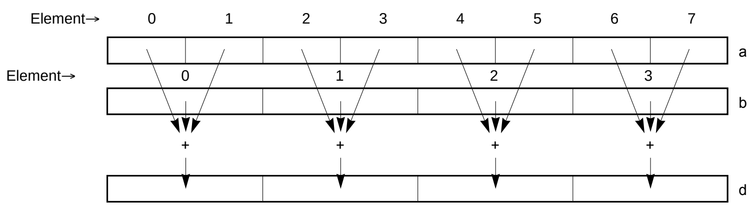
```
do i=0 to 3
d_i ← Saturate(a_2i+ a_2i+1 + b_i)
end
```

Each element of the result is the 32-bit saturated sum of the corresponding element in b and all elements in a with positions overlapping those of that element. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The valid combinations of argument types and the corresponding result types for `d = vec_sum4s(a,b)` are shown in Figure 4-137 and Figure 4-138.



d	a	b	maps to
vector unsigned int	vector unsigned char	vector unsigned int	vsum4ubs d,a,b
vector signed int	vector signed char	vector signed int	vsum4sbs d,a,b

Figure 4-137. Four Sums in the Integer Elements (32-Bit)



d	a	b	maps to
vector signed int	vector signed short	vector signed int	vsum4shs d,a,b

Figure 4-138. Four Sums in the Integer Elements (32-Bit)

vec_sum2s

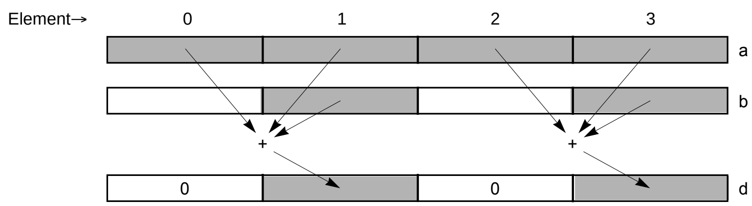
vec_sum2s

Vector Sum Across Partial (1/2) Saturated

d = vec_sum2s(**a**,**b**)

```
do i=0 to 1
d_2i ← 0
d_2i+1 ← Saturate(a_2i + a_2i+1 + b_2i+1)
end
```

The first and third elements of the result are 0. The second element of the result is the 32-bit saturated sum of the first two elements of a and the second element of b. The fourth element of the result is the 32-bit saturated sum of the last two elements of a and the fourth element of b. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The valid combination of argument types and the corresponding result type for **`d`** `= vec_sum2s(`**`a,b`**`)` are shown in Figure 4-139.



d	a	b	maps to
vector signed int	vector signed int	vector signed int	vsum2sws d,a,b

Figure 4-139. Two Saturated Sums in the Four Signed Integer Elements (32-Bit)

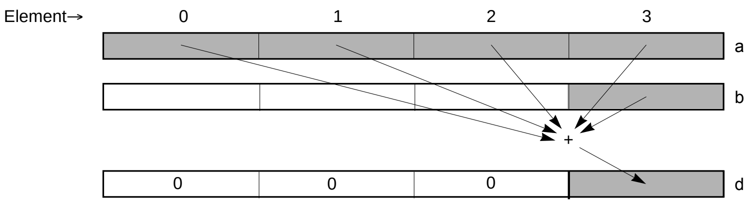
vec_sums vec_sums

Vector Sum Saturated

d = vec_sums(**a**,**b**)

```
do i=0 to 2
d_i ← 0
end
d_3 ← Saturate(a_0 + a_1 + a_2 + a_3 + b_3)
```

The first three elements of the result are 0. The fourth element of the result is the 32-bit saturated sum of all elements of a and the fourth element of b. If saturation occurs, VSCR[SAT] is set (see Table 4-1). The valid combination of argument types and the corresponding result type for `d = vec_sums(a,b)` are shown in Figure 4-140.



d	a	b	maps to
vector signed int	vector signed int	vector signed int	vsumsws d,a,b

Figure 4-140. Saturated Sum of Five Signed Integer Elements (32-Bit)

vec_trunc

vec_trunc

Vector Truncate

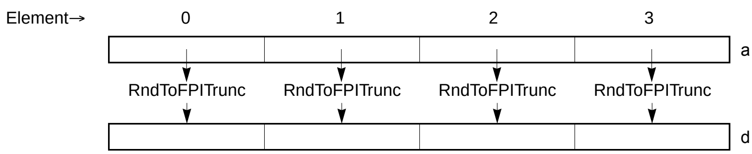
d = vec_trunc(**a**)

```
do i=0 to 3
di ← RndToFPITrunc(ai)
end
```

Each single-precision floating-point word element in a is rounded to a single-precision floating-point integer, using the Round-toward-Zero mode, and placed into the corresponding word element of d. Each element of the result is thus the value of the corresponding element of a truncated to an integral value.

The operation is independent of VSCR[NJ].

The valid argument type and corresponding result type for `d = vec_trunc(a)` are shown in Figure 4-141.



d	a	maps to
vector float	vector float	vrfiz d,a

Figure 4-141. Round-to-Zero of Four Floating-Point Integer Elements (32-Bit)

vec_unpackh

vec_unpackh

Vector Unpack High Element

d = vec_unpackh(**a**)

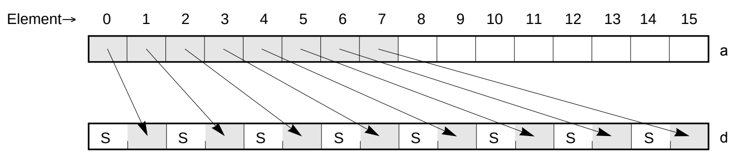
- Integer value:

```
n ← number of elements in d
do i=0 to n-1
d_i ← SignExtend(a_i)
end
```

- Pixel value:

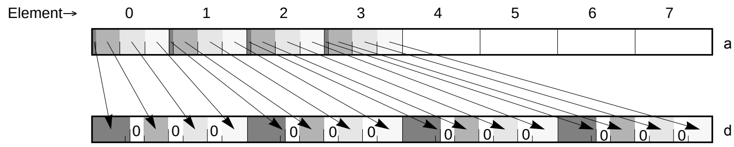
```
do i=0 to 3
d_i ← SignExtend(a_i[0]) || 000 || a_i[1:5] || 000 || a_i[6:10] || 000 || a_i[11:15]
end
```

Each element of the result is the result of extending the corresponding half-width high element of a. The valid argument types and corresponding result types for `d = vec_unpackh(a)` are shown in Figure 4-142, Figure 4-143, and Figure 4-144.



d	a	maps to
vector signed short	vector signed char	vupkhsb d,a
vector bool short	vector bool char	

Figure 4-142. Unpack High-Order Elements (8-Bit) to Elements (16-Bit)



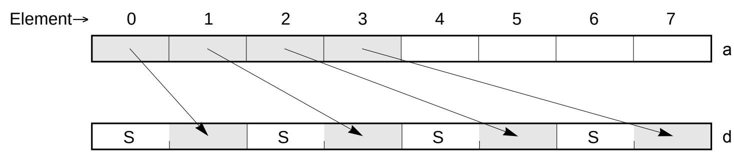
d	a	maps to
vector unsigned int	vector pixel	vupkhpx d,a

Figure 4-143. Unpack High-Order Pixel Elements (16-Bit) to Elements (32-Bit)

Programming note: Notice that the unpacking done by the vector unpack element operations for vector pixel values does not reverse the packing done by the vector pack pixel operation. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, vector unpack element inserts high-order bits while vector pack pixel discards low-order bits.)

This was designed to optimize image processing where the unpacked values would be multiplied by small coefficients and accumulated in a digital filter. The usual transformation from the 16-bit pixel to a 32-bit pixel involves multiplication of the RGB channels by 255/31. This can be accomplished by replicating the 3 most significant bits in the least significant bits using the operations:

```
d = vec_unpackh(a);
d = (vector unsigned int) vec_or(vec_sl((vector unsigned char)d,
           (vector unsigned char)(3)),
          vec_sr((vector unsigned char)d,
           (vector unsigned char)(2)));
```



d	a	maps to
vector signed int	vector signed short	vupkhsh d,a
vector bool int	vector bool short	

Figure 4-144. Unpack High-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

vec_unpackl

vec_unpackl

Vector Unpack Low Element

d = vec_unpackl(**a**)

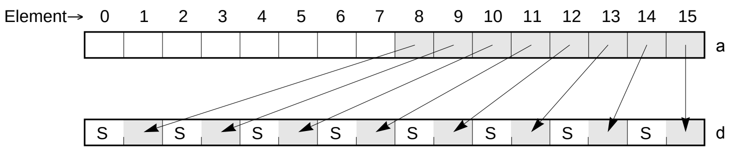
- Integer value:

```
n ← number of elements in d
do i=0 to n-1
d_i ← SignExtend(a_i+n)
end
```

- Pixel value:

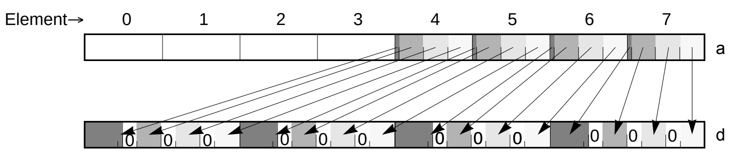
```
do i=0 to 3
d_i ← SignExtend(a_i+n[0])‖000‖a_i+n[1:5]‖000‖a_i+n[6:10]‖000‖a_i+n[11:15]
end
```

Each element of the result is the result of extending the corresponding half-width low element of a. The valid argument types and corresponding result types for `d = vec_unpackl(a)` are shown in Figure 4-145, Figure 4-146, and Figure 4-147.



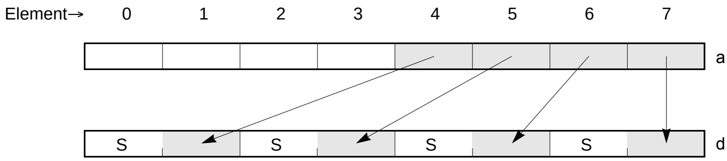
d	a	maps to
vector signed short	vector signed char	vupklsb d,a
vector bool short	vector bool char	

Figure 4-145. Unpack Low-Order Elements (8-Bit) to Elements (16-Bit)



d	a	maps to
vector unsigned int	vector pixel	vupklpx d,a

Figure 4-146. Unpack Low-Order Pixel Elements (16-Bit) to Elements (32-Bit)



d	a	maps to
vector signed int	vector signed short	vupklsh d,a
vector bool int	vector bool short	

Figure 4-147. Unpack Low-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

Programming note: Notice that the unpacking done by the vector unpack element operations for vector pixel values does not reverse the packing done by the vector pack pixel operation. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, vector unpack element inserts high-order bits while vector pack pixel discards low-order bits.)

This was designed to optimize image processing where the unpacked values would be multiplied by small coefficients and accumulated in a digital filter. The usual transformation from the 16-bit pixel to a 32-bit pixel involves multiplication of the RGB channels by 255/31. This can be accomplished by replicating the 3 most significant bits in the least significant bits using the operations:

```
d = vec_unpackh(a);
d = (vector unsigned int) vec_or(vec_sl((vector unsigned char)d,
                                          (vector unsigned char)(3)),
                                      vec_sr((vector unsigned char)d,
                                         (vector unsigned char)(2)));
```

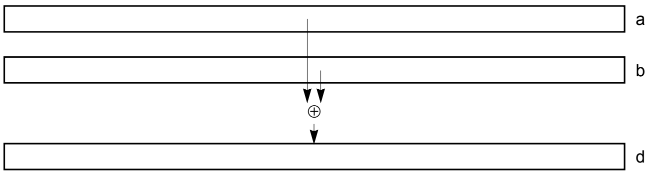
vec_xor vec_xor

Vector Logical XOR

d = vec_xor(**a**,**b**)

d ← **a** ⊕ **b**

Each bit of the result is the logical XOR of the corresponding bits of a and b.
The valid combinations of argument types and the corresponding result types for **`d = vec_xor(a,b)`** are shown in Figure 4-148.



d	a	b	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vxor d,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
vector signed char	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector signed char	
vector bool char	vector bool char	vector bool char	
vector unsigned short	vector unsigned short	vector unsigned short	
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector signed short	
vector bool short	vector bool short	vector bool short	
vector unsigned int	vector unsigned int	vector unsigned int	
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector signed int	
vector bool int	vector bool int	vector bool int	
vector float	vector bool int	vector float	
	vector float	vector bool int	
	vector float	vector float	

Figure 4-148. Logical Bit-Wise XOR

4.5 AltiVec Predicates

The AltiVec predicates all begin with `vec_all_` or `vec_any_`. The AltiVec predicates are organized alphabetically by predicate name with a definition of the permitted generic AltiVec predicates. The specific operations do not exist for the predicates.

Where possible, the description is supported by reference figures indicating data modifications and including a table that lists:

- the valid set of argument types for that predicate, and
- the specific AltiVec instruction generated for that set of arguments. The AltiVec instruction is in the form v-----. x,a,b, where v-----. represents the instruction and x,a,b represent the operands. The x represents an unused vector result of the vector compare instruction used to implement the predicate. The order of operands listed after the instruction indicate the order in which they are applied for that predicate.

For example,

```
vec_any_lt(vector unsigned char, vector unsigned char)
```

maps to the instruction

```
vcmpgtb. x,b,a
```

indicating that the operands are applied in reverse order for this predicate.

vec_all_eq

vec_all_eq

All Elements Equal

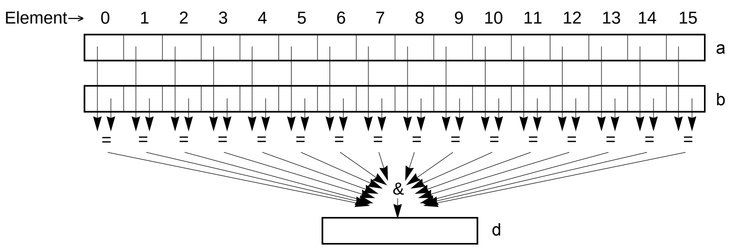
d = vec_all_eq(**a**,**b**)

```
n ← number of elements
if each a_i =_int b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_all_eq` returns 1 if every element of a is equal to the corresponding element of b. Otherwise, it returns 0.

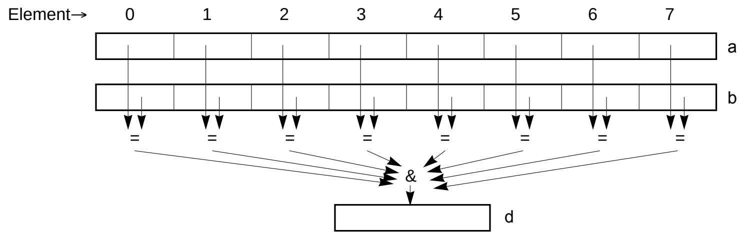
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for **`d = vec_all_eq(a,b)`** are shown in Figure 4-149, Figure 4-150, Figure 4-151, and Figure 4-152.



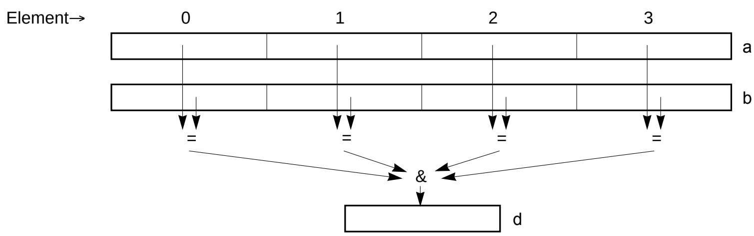
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpequb. x,a,b
	vector unsigned char	vector bool char	
	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector unsigned char	
	vector bool char	vector signed char	
	vector bool char	vector bool char	

Figure 4-149. All Equal of Sixteen Integer Elements (8-bits)



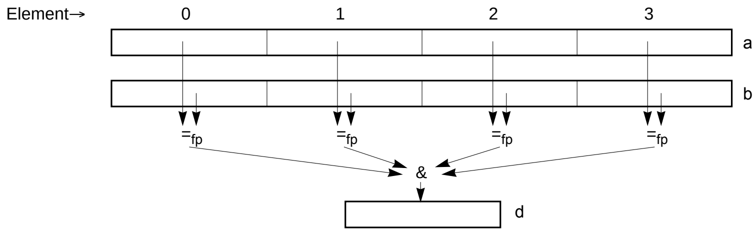
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpequh. x,a,b
	vector unsigned short	vector bool short	
	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector unsigned short	
	vector bool short	vector signed short	
	vector bool short	vector bool short	
	vector pixel	vector pixel	

Figure 4-150. All Equal of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpequw. x,a,b
	vector unsigned int	vector bool int	
	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector unsigned int	
	vector bool int	vector signed int	
	vector bool int	vector bool int	

Figure 4-151. All Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpeqfp. x,a,b

Figure 4-152. All Equal of Four Floating-Point Elements (32-Bit)

vec_all_ge

vec_all_ge

All Elements Greater Than or Equal

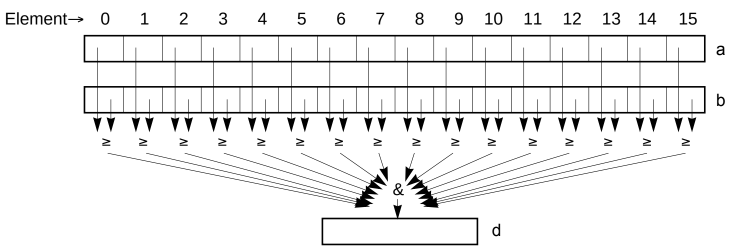
d = vec_all_ge(**a**,**b**)

```
n ← number of elements
if each a_i ≥ b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_all_ge` returns 1 if every element of a is greater than or equal to the corresponding element of b. Otherwise, it returns 0.

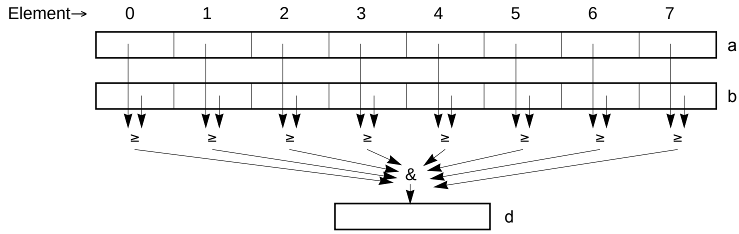
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for **`d = vec_all_ge(a,b)`** are shown in Figure 4-153, Figure 4-154, Figure 4-155, and Figure 4-156.



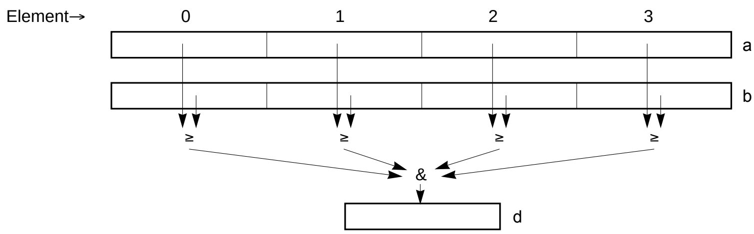
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x.b,a
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,b,a
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-153. All Greater Than or Equal of Sixteen Integer Elements (8-bits)



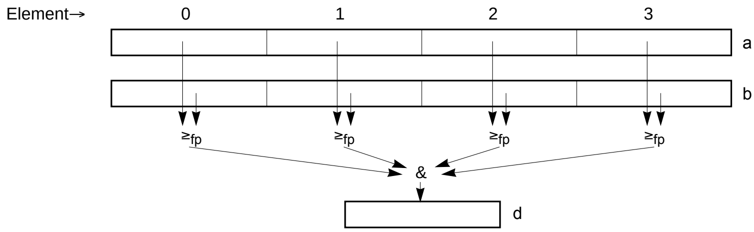
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,b,a
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,b,a
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-154. All Greater Than or Equal of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,b,a
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,b,a
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-155. All Greater Than or Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x,a,b

Figure 4-156. All Greater Than or Equal of Four Floating-Point Elements (32-Bit)

vec_all_gt

vec_all_gt

All Elements Greater Than

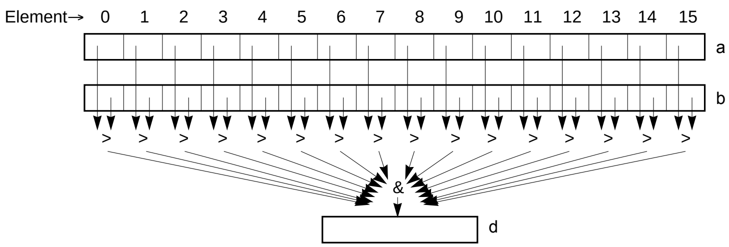
d = vec_all_gt(**a**,**b**)

```
n ← number of elements
if each a_i > b_i, where i ranges from 0 to n-1
then d ← 1
else  ← 0
```

The predicate `vec_all_gt` returns 1 if every element of a is greater than the corresponding element of b. Otherwise, it returns 0.

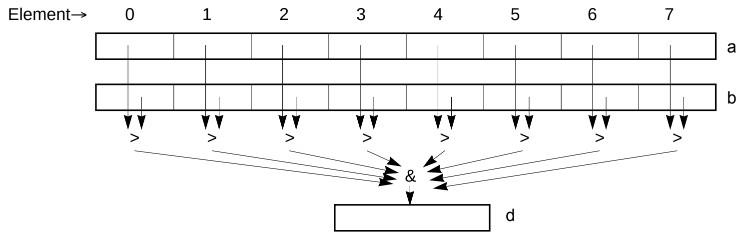
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for `d = vec_all_gt(a,b)` are shown in Figure 4-157, Figure 4-158, Figure 4-159, and Figure 4-160.



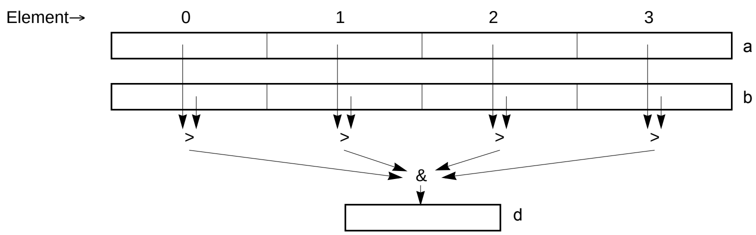
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-157. All Greater Than of Sixteen Integer Elements (8-bits)



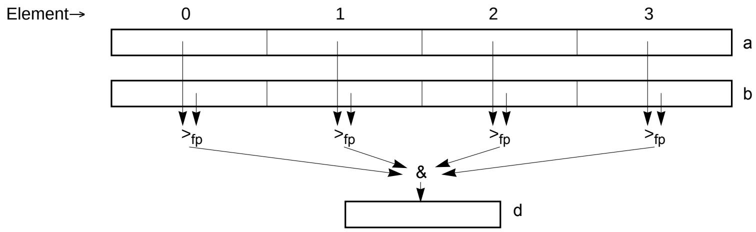
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,a,b
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-158. All Greater Than of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-159. All Greater Than of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,a,b

Figure 4-160. All Greater Than of Four Floating-Point Elements (32-Bit)

vec_all_in

vec_all_in

All Elements in Bounds

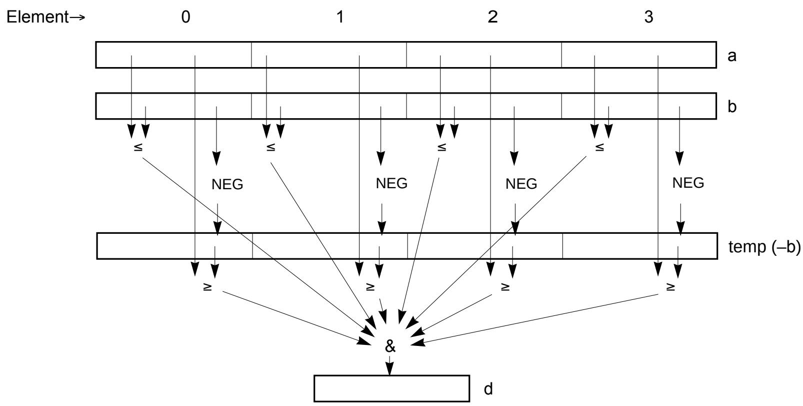
d = vec_all_in(**a**,**b**)

```
if each a_i ≤ b_i and a_i ≥ -b_i, where i ranges from 0 to 3
then d ← 1
else  ← 0
```

The predicate `vec_all_in` returns 1 if every element of a is less than or equal to the corresponding element of b (high bound) and greater than or equal to the negative (NEG) of the corresponding element of b (low bound). Otherwise, it returns 0.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for **`d`** `= vec_all_in(`**`a,b`**`)` are shown in Figure 4-161.



d	a	b	Maps to
int	vector float	vector float	vcmpbfp. x,a,b

Figure 4-161. All in Bounds of Four Floating-Point Elements (32-Bit)

vec_all_le

vec_all_le

All Elements Less Than or Equal

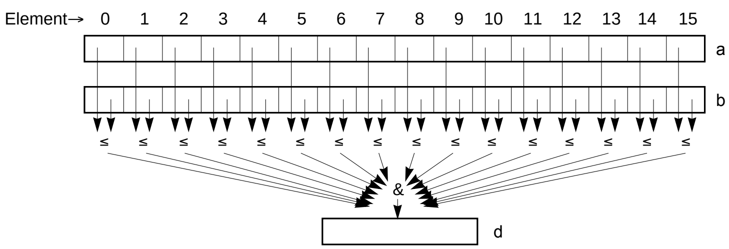
d = vec_all_le(**a**,**b**)

```
n ← number of elements
if each ai ≤ bi, where i ranges from 0 to n-1
then d ← 0
else d ← 1
```

The predicate `vec_all_le` returns 1 if every element of a is less than or equal to the corresponding element of b. Otherwise, it returns 0.

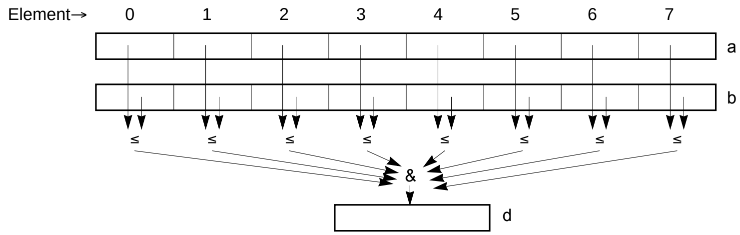
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for **`d = vec_all_le(a,b)`** are shown in Figure 4-162, Figure 4-163, Figure 4-164, and Figure 4-165.



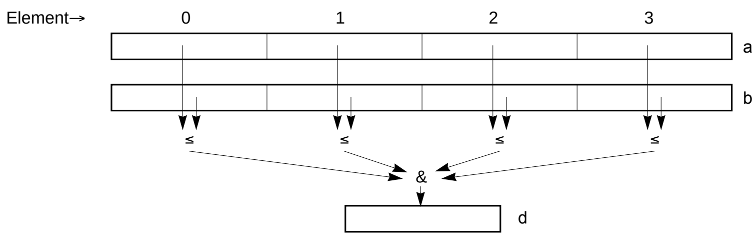
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-162. All Less Than or Equal of Sixteen Integer Elements (8-bits)



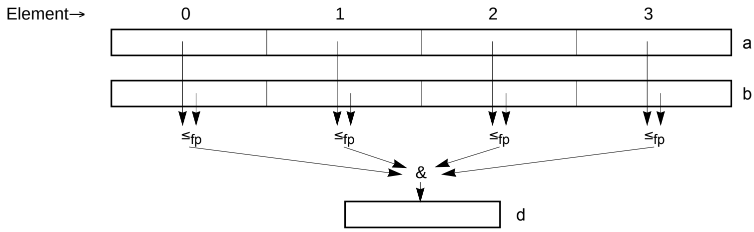
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,b,a
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-163. All Less Than or Equal of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-164. All Less Than or Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x,b,a

Figure 4-165. All Less Than or Equal of Four Floating-Point Elements (32-Bit)

vec_all_lt vec_all_lt

All Elements Less Than

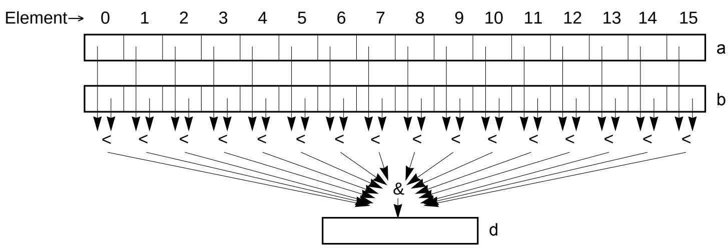
d = vec_all_lt(**a**,**b**)

```
n ← number of elements
if each a_i < b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_all_lt` returns 1 if every element of a is less than the corresponding element of b. Otherwise, it returns 0.

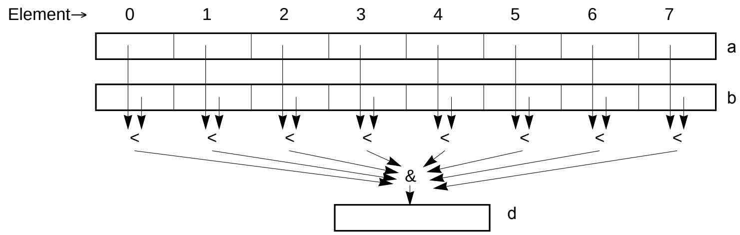
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for **`d = vec_all_lt(a,b)`** are shown in Figure 4-166, Figure 4-167, Figure 4-168, and Figure 4-169.



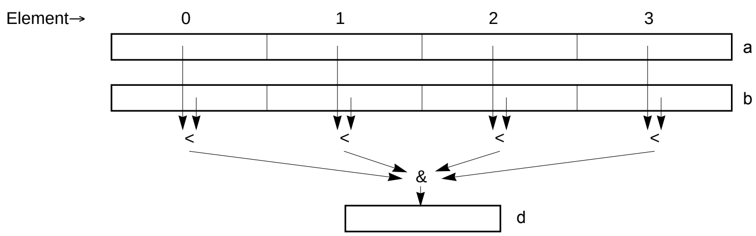
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x,b,a
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,b,a
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-166. All Less Than of Sixteen Integer Elements (8-bits)



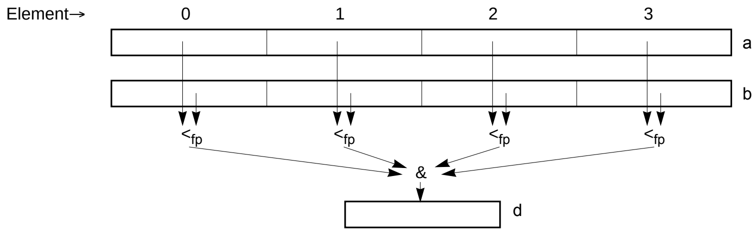
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,b,a
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,b,a
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-167. All Less Than of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,b,a
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,b,a
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-168. All Less Than of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,b,a

Figure 4-169. All Less Than of Four Floating-Point Elements (32-Bit)

vec_all_nan vec_all_nan

All Elements Not a Number

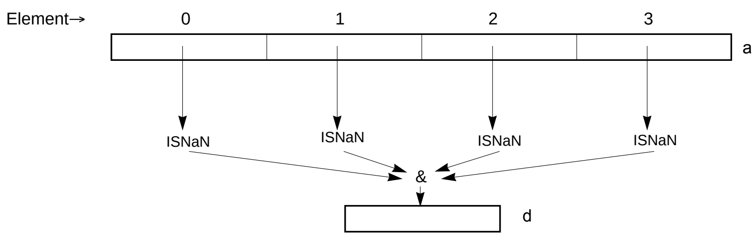
d = vec_all_nan(**a**)

```
if each ISNaN(ai) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_all_nan` returns 1 if every element of a is Not a Number (NaN). Otherwise, it returns 0.

The operation is independent of VSCR[NJ].

The valid argument type and corresponding result type for **`d`** `= vec_all_nan(`**`a`**`)` are shown in Figure 4-170.



d	a	Maps to
int	vector float	vcmpeqfp. x,a,a

Figure 4-170. All NaN of Four Floating-Point Elements (32-Bit)

vec_all_ne

vec_all_ne

All Elements Not Equal

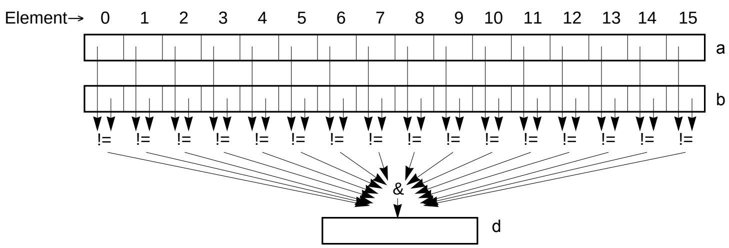
d = vec_all_ne(**a**,**b**)

```
n ← number of elements
if each ai != bi, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_all_ne` returns 1 if every element of a is not equal to (!=) the corresponding element of b. Otherwise, it returns 0.

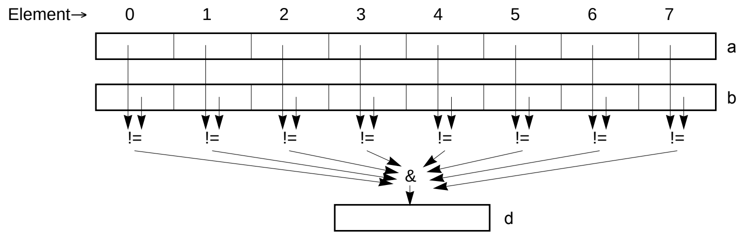
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for `d = vec_all_ne(a,b)` are shown in Figure 4-171, Figure 4-172, Figure 4-173, and Figure 4-174.



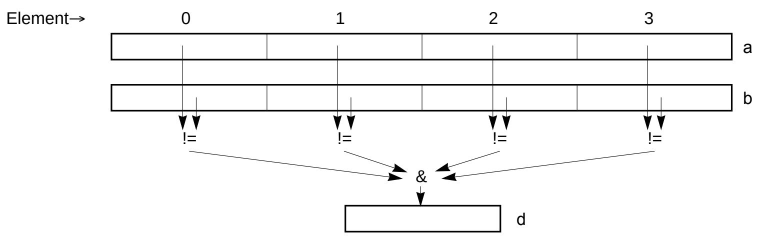
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpequb. x,a,b
	vector unsigned char	vector bool char	
	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector unsigned char	
	vector bool char	vector signed char	
	vector bool char	vector bool char	

Figure 4-171. All Not Equal of Sixteen Integer Elements (8-bits)



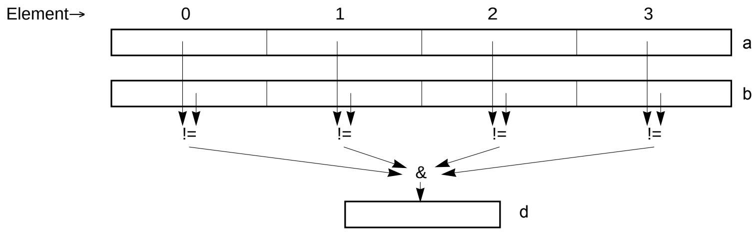
int	vector unsigned short	vector unsigned short	vcmpequh. x,a,b
	vector unsigned short	vector bool short	
	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector unsigned short	
	vector bool short	vector signed short	
	vector bool short	vector bool short	
	vector pixel	vector pixel	

Figure 4-172. All Not Equal of Eight Integer Elements (16-Bit)



int	vector unsigned int	vector unsigned int	vcmpequw. x,a,b
	vector unsigned int	vector bool int	
	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector unsigned int	
	vector bool int	vector signed int	
	vector bool int	vector bool int	

Figure 4-173. All Not Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpeqfp. x,a,b

Figure 4-174. All Not Equal of Four Floating-Point Elements (32-Bit)

vec_all_nge

vec_all_nge

All Elements Not Greater Than or Equal

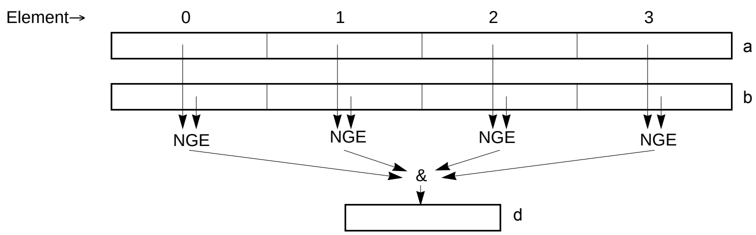
d = vec_all_nge(**a**,**b**)

```
if each NGE(a_i, b_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_all_nge` returns 1 if every element of a is not greater than or equal to (NGE) the corresponding element of b. Otherwise, it returns 0. Not greater than or equal can mean either less than or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for `d = vec_all_nge(a,b)` are shown in Figure 4-175.



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x,a,b

Figure 4-175. All Not Greater Than or Equal of Four Floating-Point Elements (32-Bit)

vec_all_ngt vec_all_ngt

All Elements Not Greater Than

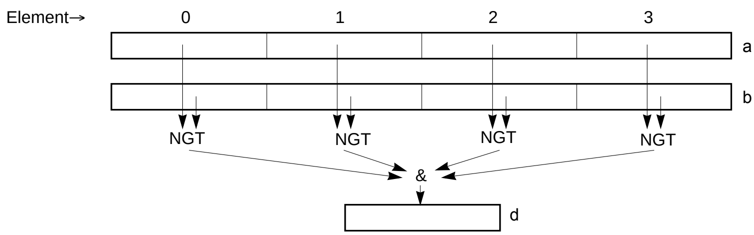
d = vec_all_ngt(**a**,**b**)

```
if each NGT(a_i, b_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_all_ngt` returns 1 if every element of a is not greater than (NGT) the corresponding element of b. Otherwise, it returns 0. Not greater than can either mean less than or equal to or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for `d = vec_all_ngt(a,b)` is shown in Figure 4-176.



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,a,b

Figure 4-176. All Not Greater Than of Four Floating-Point Elements (32-Bit)

vec_all_nle vec_all_nle

All Elements Not Less Than or Equal

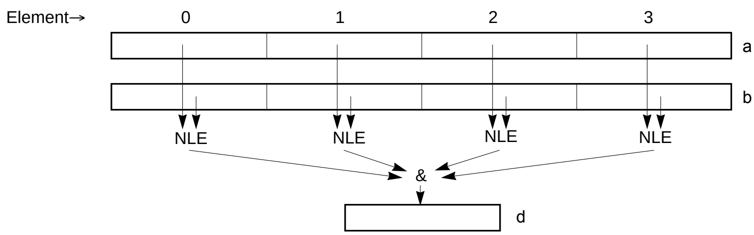
d = vec_all_nle(**a**,**b**)

```
if each NLE(a_i, b_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_all_nle` returns 1 if every element of a is not less than or equal to (NLE) the corresponding element of b. Otherwise, it returns 0. Not less than or equal to can either mean greater than or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for **`d`** `= vec_all_nle(`**`a,b`**`)` are shown in Figure 4-177.



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x, b, a

Figure 4-177. All Not Less Than or Equal of Four Floating-Point Elements (32-Bit)

vec_all_nlt

vec_all_nlt

All Elements Not Less Than

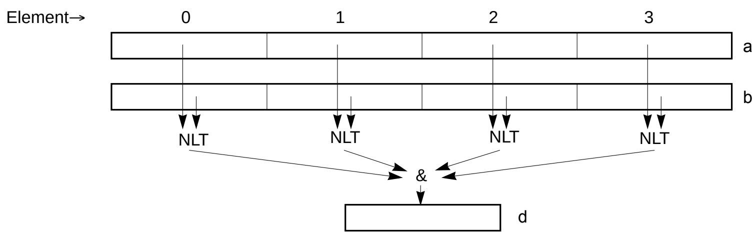
d = vec_all_nlt(**a**,**b**)

```
if each NLT(ai, bi), where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_all_nlt` returns 1 if every element of a is not less than (NLT) the corresponding element of b. Otherwise, it returns 0. Not less than can either mean greater than or equal to or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid argument types and the corresponding result type for `d = vec_all_nlt(a,b)` are shown in Figure 4-178.



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,b,a

Figure 4-178. All Not Less Than of Four Floating-Point Elements (32-Bit)

vec_all_numeric vec_all_numeric

All Elements Numeric

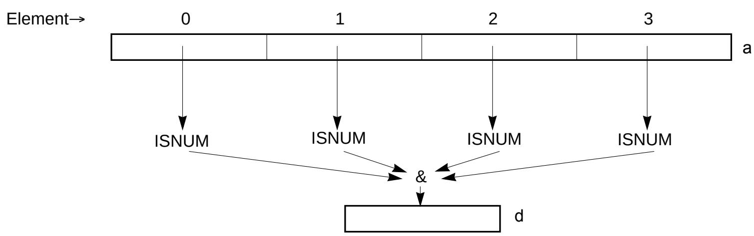
d = vec_all_numeric(**a**)

```
if each ISNUM(a_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_all_numeric` returns 1 if every element of a is numeric. Otherwise, it returns 0.

The operation is independent of VSCR[NJ].

The valid argument types and the corresponding result type for **`d`** `= vec_all_numeric()` are shown in Figure 4-179.



d	a	Maps to
int	vector float	vcmpeqfp. x,a,a

Figure 4-179. All Numeric of Four Floating-Point Elements (32-Bit)

vec_any_eq vec_any_eq

Any Element Equal

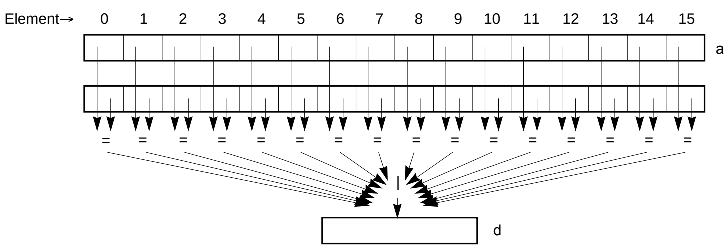
d = vec_any_eq(**a**,**b**)

```
n ← number of elements
if any a_i =_int b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_any_eq` returns 1 if any element of a is equal to the corresponding element of b. Otherwise, it returns 0.

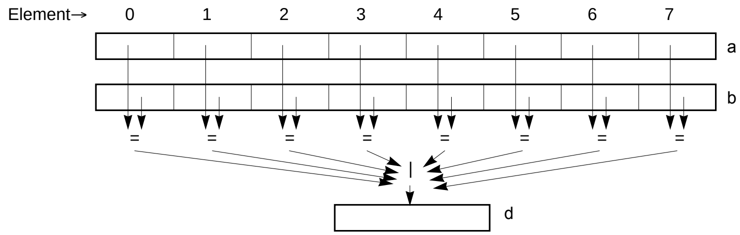
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for **`d = vec_any_eq(a,b)`** are shown in Figure 4-180, Figure 4-181, Figure 4-182, and Figure 4-183.



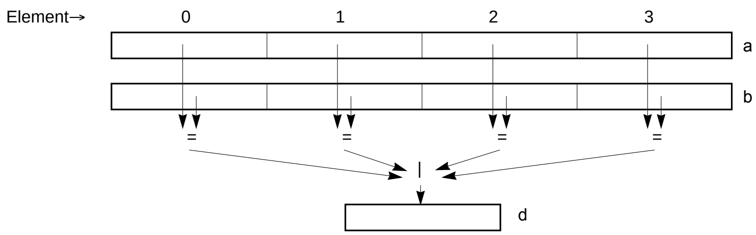
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpequb. x,a,b
	vector unsigned char	vector bool char	
	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector unsigned char	
	vector bool char	vector signed char	
	vector bool char	vector bool char	

Figure 4-180. Any Equal of Sixteen Integer Elements (8-bits)



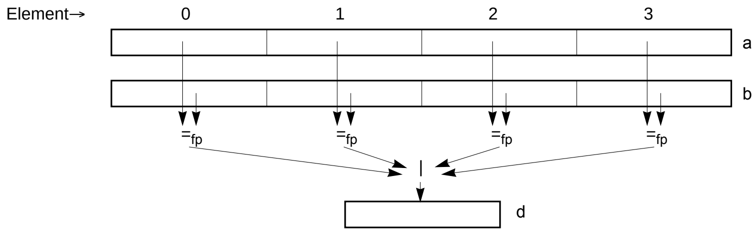
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpequh. x,a,b
	vector unsigned short	vector bool short	
	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector unsigned short	
	vector bool short	vector signed short	
	vector bool short	vector bool short	
	vector pixel	vector pixel	

Figure 4-181. Any Equal of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpequw. x,a,b
	vector unsigned int	vector bool int	
	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector unsigned int	
	vector bool int	vector signed int	
	vector bool int	vector bool int	

Figure 4-182. Any Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpeqfp. x,a,b

Figure 4-183. Any Equal of Four Floating-Point Elements (32-Bit)

vec_any_ge

vec_any_ge

Any Element Greater Than or Equal

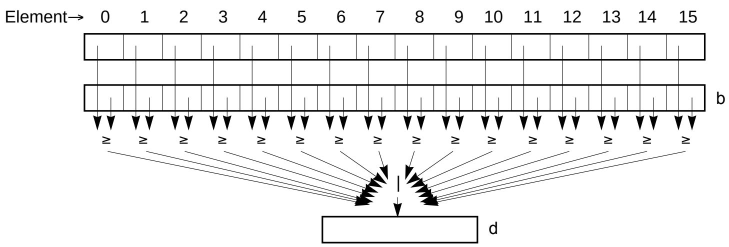
d = vec_any_ge(**a**,**b**)

```
n ← number of elements
if any a_i ≥ b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_any_ge` returns 1 if any element of a is greater than or equal to the corresponding element of b. Otherwise, it returns 0.

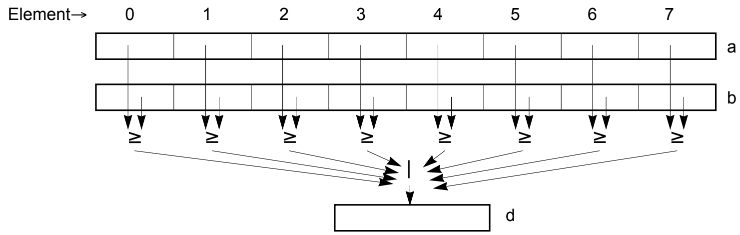
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for `d = vec_any_ge(a,b)` are shown in Figure 4-184, Figure 4-185, Figure 4-186, and Figure 4-187.



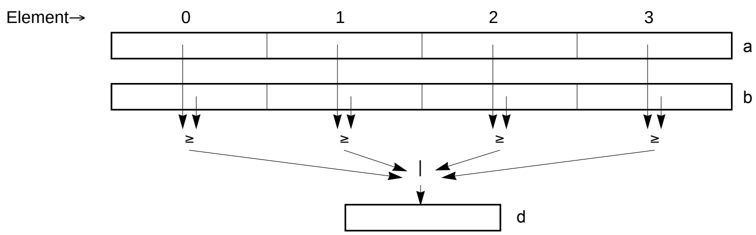
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x,b,a
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,b,a
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-184. Any Greater Than or Equal of Sixteen Integer Elements (8-bits)



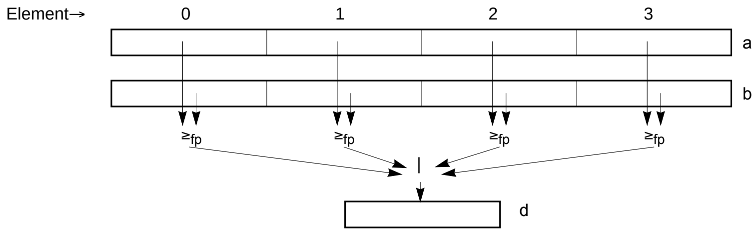
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,b,a
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,b,a
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-185. Any Greater Than or Equal of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,b,a
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,b,a
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-186. Any Greater Than or Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x,a,b

Figure 4-187. Any Greater Than or Equal of Four Floating-Point Elements (32-Bit)

vec_any_gt

vec_any_gt

Any Element Greater Than

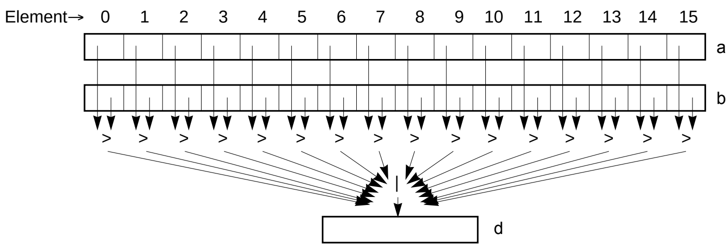
d = vec_any_gt(**a**,**b**)

```
n ← number of elements
if any a_i > b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_any_gt` returns 1 if any element of a is greater than the corresponding element of b. Otherwise, it returns 0.

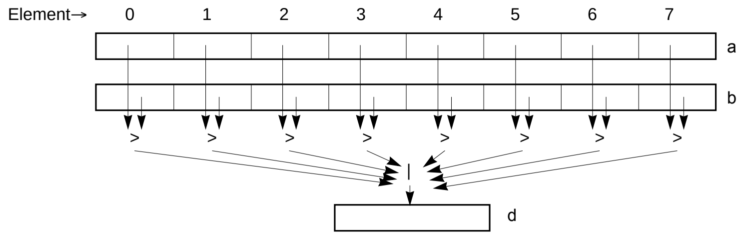
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for `d = vec_any_gt(a,b)` are shown in Figure 4-188, Figure 4-189, Figure 4-190, and Figure 4-191.



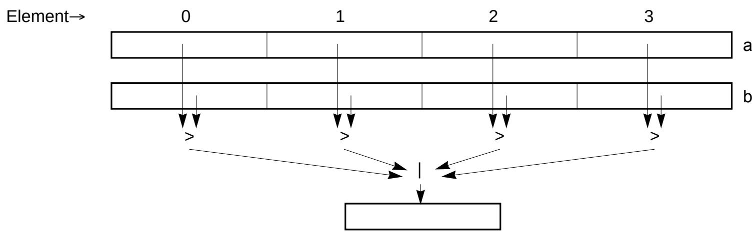
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-188. Any Greater Than of Sixteen Integer Elements (8-bits)



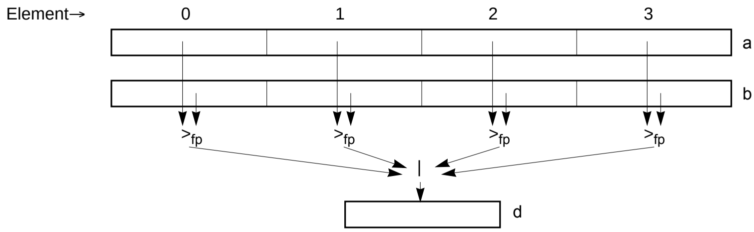
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,a,b
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-189. Any Greater Than of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-190. Any Greater Than of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,a,b

Figure 4-191. Any Greater Than of Four Floating-Point Elements (32-Bit)

vec_any_le vec_any_le

Any Element Less Than or Equal

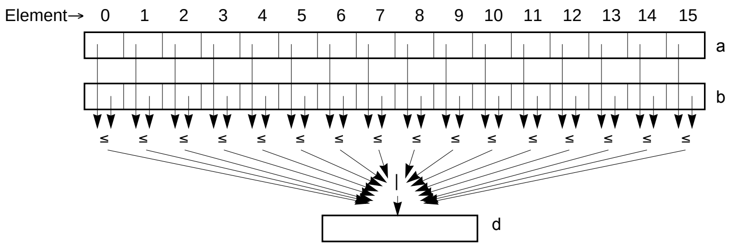
d = vec_any_le(**a**,**b**)

```
n ← number of elements
if any a_i ≤ b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_any_le` returns 1 if any element of a is less than or equal to the corresponding element of b. Otherwise, it returns 0.

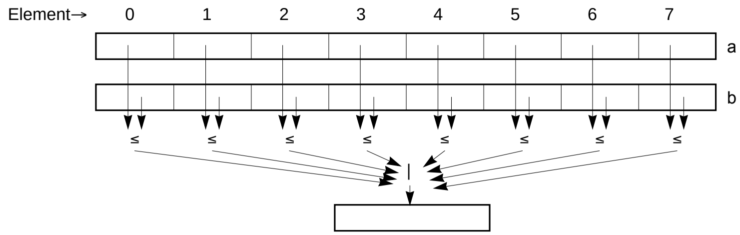
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for **`d = vec_any_le(a,b)`** are shown in Figure 4-192, Figure 4-193, Figure 4-194, and Figure 4-195.



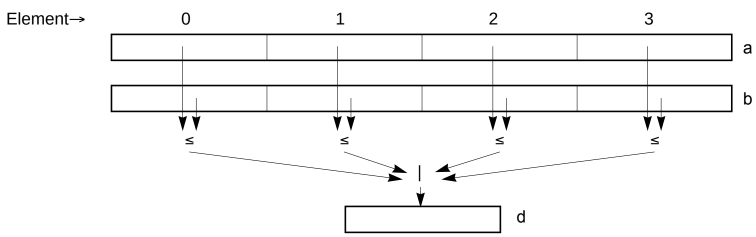
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x,a,b
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,a,b
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-192. Any Less Than or Equal of Sixteen Integer Elements (8-bits)



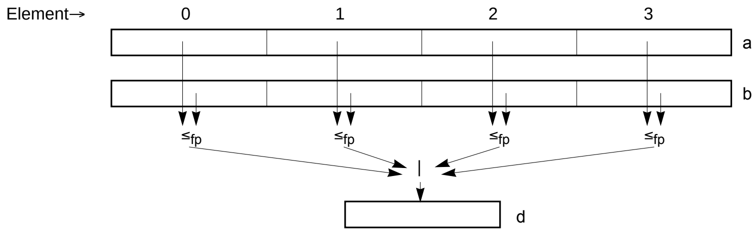
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,a,b
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-193. Any Less Than or Equal of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,a,b
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-194. Any Less Than or Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x,b,a

Figure 4-195. Any Less Than or Equal of Four Floating-Point Elements (32-Bit)

vec_any_lt

vec_any_lt

Any Element Less Than

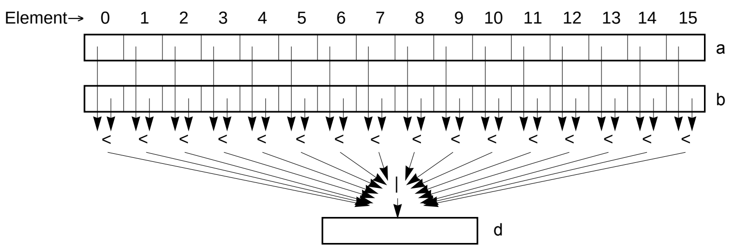
d = vec_any_lt(**a**,**b**)

```
n ← number of elements
if any a_i < b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_any_lt` returns 1 if any element of a is less than the corresponding element of b. Otherwise, it returns 0.

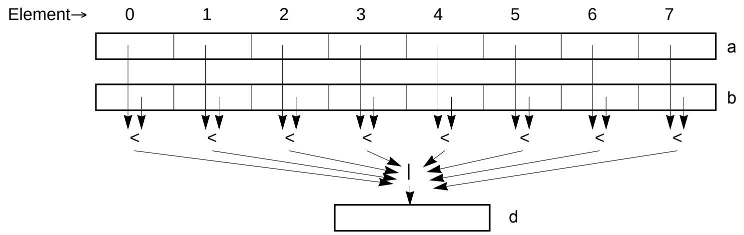
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result type for **`d = vec_any_lt(a,b)`** are shown in Figure 4-196, Figure 4-197, Figure 4-198, and Figure 4-199.



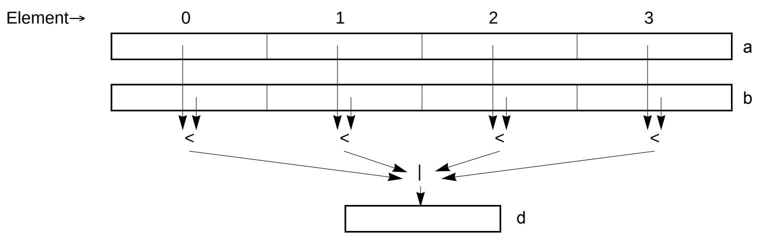
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpgtub. x,b,a
	vector unsigned char	vector bool char	
	vector bool char	vector unsigned char	
	vector signed char	vector signed char	vcmpgtsb. x,b,a
	vector signed char	vector bool char	
	vector bool char	vector signed char	

Figure 4-196. Any Less Than of Sixteen Integer Elements (8-bits)



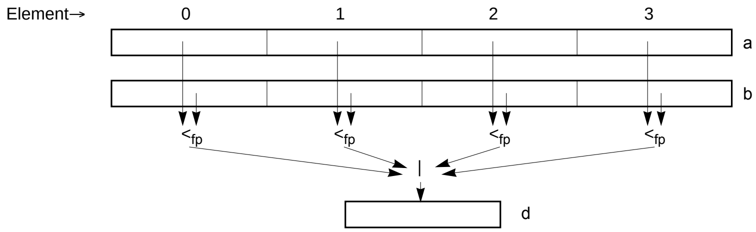
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpgtuh. x,b,a
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
	vector signed short	vector signed short	vcmpgtsh. x,b,a
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Figure 4-197. Any Less Than of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpgtuw. x,b,a
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
	vector signed int	vector signed int	vcmpgtsw. x,b,a
	vector signed int	vector bool int	
	vector bool int	vector signed int	

Figure 4-198. Any Less Than of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,b,a

Figure 4-199. Any Less Than of Four Floating-Point Elements (32-Bit)

vec_any_nan

vec_any_nan

Any Element Not a Number

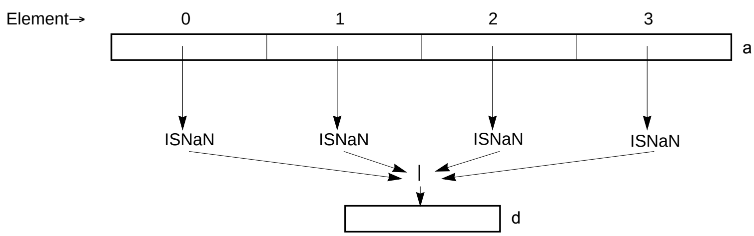
d = vec_any_nan(**a**)

```
if any ISNaN(ai) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_any_nan` returns 1 if any element of a is Not a Number (NaN). Otherwise, it returns 0.

The operation is independent of VSCR[NJ].

The valid argument type and corresponding result type for `d = vec_any_nan(a)` are shown in Figure 4-200.



d	a	Maps to
int	vector float	vcmpeqfp. x,a,a

Figure 4-200. Any NaN of Four Floating-Point Elements (32-Bit)

vec_any_ne vec_any_ne

Any Element Not Equal

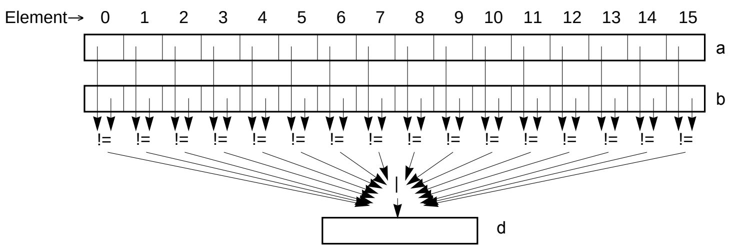
d = vec_any_ne(**a**,**b**)

```
n ← number of elements
if any a_i != b_i, where i ranges from 0 to n-1
then d ← 1
else d ← 0
```

The predicate `vec_any_ne` returns 1 if any element of a is not equal to (!=) the corresponding element of b. Otherwise, it returns 0.

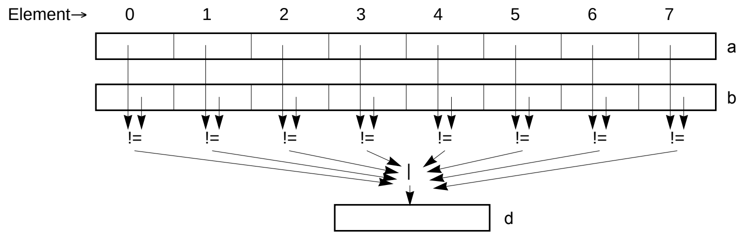
For `vector float` argument types, if VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combinations of argument types and the corresponding result types for **`d = vec_any_ne(a,b)`** are shown in Figure 4-201, Figure 4-202, Figure 4-203, and Figure 4-204.



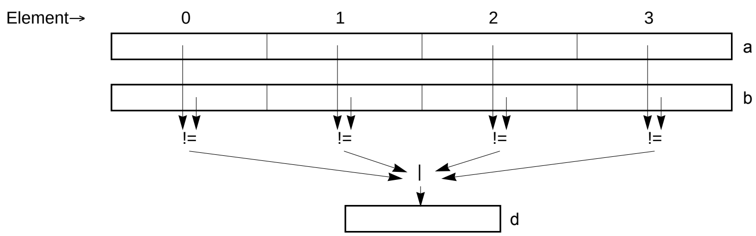
d	a	b	Maps to
int	vector unsigned char	vector unsigned char	vcmpequb. x,a,b
	vector unsigned char	vector bool char	
	vector signed char	vector signed char	
	vector signed char	vector bool char	
	vector bool char	vector unsigned char	
	vector bool char	vector signed char	
	vector bool char	vector bool char	

Figure 4-201. Any Not Equal of Sixteen Integer Elements (8-bits)



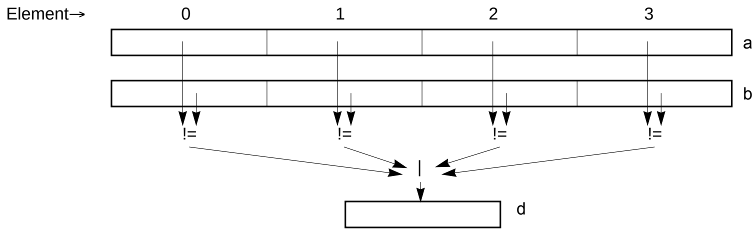
d	a	b	Maps to
int	vector unsigned short	vector unsigned short	vcmpequh. x,a,b
	vector unsigned short	vector bool short	
	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector unsigned short	
	vector bool short	vector signed short	
	vector bool short	vector bool short	
	vector pixel	vector pixel	

Figure 4-202. Any Not Equal of Eight Integer Elements (16-Bit)



d	a	b	Maps to
int	vector unsigned int	vector unsigned int	vcmpequw. x,a,b
	vector unsigned int	vector bool int	
	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector unsigned int	
	vector bool int	vector signed int	
	vector bool int	vector bool int	

Figure 4-203. Any Not Equal of Four Integer Elements (32-Bit)



d	a	b	Maps to
int	vector float	vector float	vcmpeqfp. x,a,b

Figure 4-204. Any Not Equal of Four Floating-Point Elements (32-Bit)

vec_any_nge

vec_any_nge

Any Element Not Greater Than or Equal

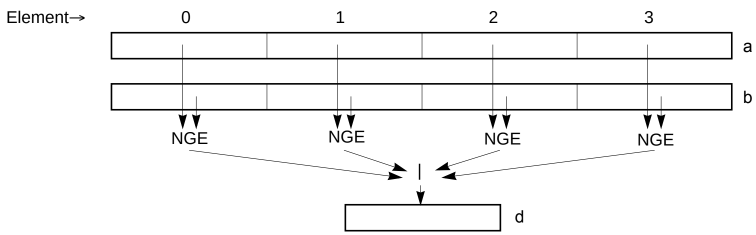
d = vec_any_nge(**a**,**b**)

```
if any NGE(a_i, b_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_any_nge` returns 1 if any element of a is not greater than or equal to (NGE) the corresponding element of b. Otherwise, it returns 0. Not greater than or equal can either mean less than or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combination of argument types and the corresponding result type for **`d = vec_any_nge(a,b)`** are shown in Figure 4-205.



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x,a,b

Figure 4-205. Any Not Greater Than or Equal of Four Floating-Point Elements (32-Bit)

vec_any_ngt vec_any_ngt

Any Element Not Greater Than

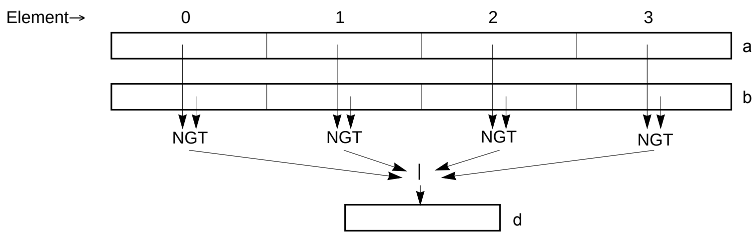
d = vec_any_ngt(**a**,**b**)

```
if any NGT(a_i, b_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_any_ngt` returns 1 if any element of a is not greater than (NGT) the corresponding element of b. Otherwise, it returns 0. Not greater than can either mean less than or equal to or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combination of argument types and the corresponding result type for **`d = vec_any_ngt(a,b)`** are shown in Figure 4-206.



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,a,b

Figure 4-206. Any Not Greater Than of Four Floating-Point Elements (32-Bit)

vec_any_nle vec_any_nle

Any Element Not Less Than or Equal

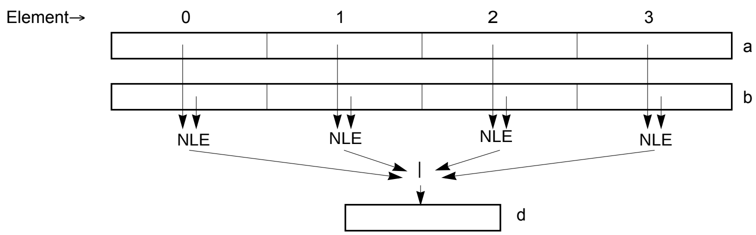
d = vec_any_nle(**a**,**b**)

```
if any NLE(a_i, b_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_any_nle` returns 1 if any element of a is not less than or equal to (NLE) the corresponding element of b. Otherwise, it returns 0. Not less than or equal to can either mean greater than or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combination of argument types and the corresponding result type for **`d = vec_any_nle(a,b)`** are shown in Figure 4-207.



d	a	b	Maps to
int	vector float	vector float	vcmpgefp. x,b,a

Figure 4-207. Any Not Less Than or Equal of Four Floating-Point Elements (32-Bit)

vec_any_nlt

vec_any_nlt

Any Element Not Less Than

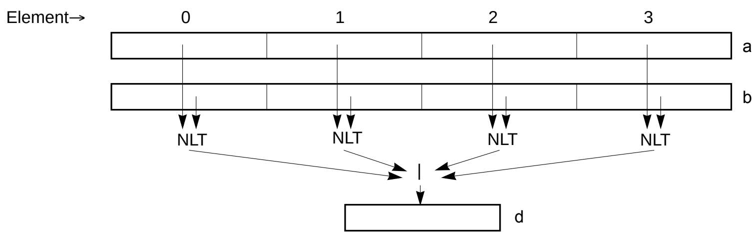
d = vec_any_nlt(**a**,**b**)

```
if any NLT(ai, bi) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_any_nlt` returns 1 if any element of a is not less than (NLT) the corresponding element of b. Otherwise, it returns 0. Not less than can either mean greater than or equal to or that one of the elements is NaN.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combination of argument types and the corresponding result type for **`d = vec_any_nlt(a,b)`** are shown in Figure 4-208.



d	a	b	Maps to
int	vector float	vector float	vcmpgtfp. x,b,a

Figure 4-208. Any Not Less Than of Four Floating-Point Elements (32-Bit)

vec_any_numeric vec_any_numeric

Any Element Numeric

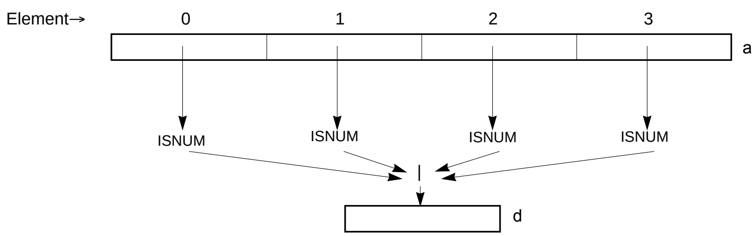
d = vec_any_numeric(**a**)

```
if any ISNUM(a_i) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_any_numeric` returns 1 if any element of a is numeric. Otherwise, it returns 0.

The operation is independent of VSCR[NJ].

The valid argument type and the corresponding result type for **`d`** `= vec_any_numeric(`**`a`**`)` are shown in Figure 4-209.



d	a	Maps to
int	vector float	vcmpeqfp. x,a,a

Figure 4-209. Any Numeric of Four Floating-Point Elements (32-Bit)

vec_any_out

vec_any_out

Any Element Out of Bounds

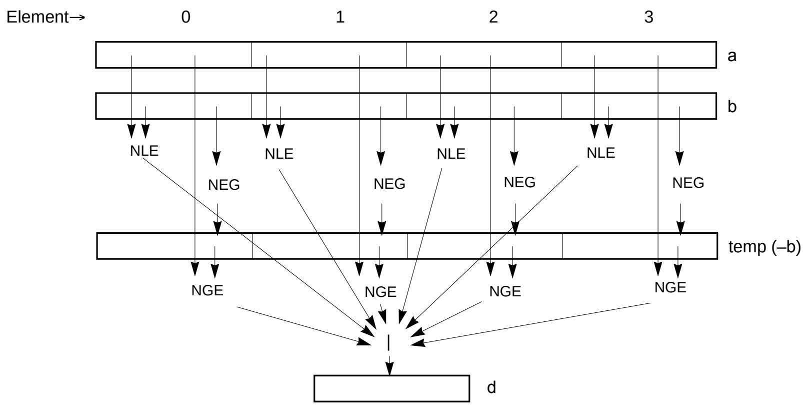
d = vec_any_out(**a**,**b**)

```
if any NLE(ai, bi) = 1 or any NGE(ai, -bi) = 1, where i ranges from 0 to 3
then d ← 1
else d ← 0
```

The predicate `vec_any_out` returns 1 if any element of a is greater than the corresponding element of b (high bound) or is less than the negative (NEG) of the corresponding element of b (low bound). Otherwise, it returns 0.

If VSCR[NJ] = 1, every denormalized floating-point operand element is truncated to 0 before the comparison.

The valid combination of argument types and the corresponding result type for **d** = `vec_any_out(`**a**,**b**`)` are shown in Figure 4-210.



d	a	b	Maps to
int	vector float	vector float	vcmpbfp. x,a,b

Figure 4-210. Any Out of Bounds of Four Floating-Point Elements (32-Bit)

Appendix A
AltiVec Instruction Set/Operation/Predicate Cross-Reference

This appendix cross-references the instruction set for the AltiVec™ technology, the AltiVec vector operations, and the AltiVec predicates. Table A-1 lists the instructions and the alternate vector operation form cross-referenced to the vector operations and predicates.

Table A-1. Instructions to Operations/Predicates Cross-Reference

AltiVec Instruction	Specific Operation	Generic Operation/Predicate
dss	vec_dss	vec_dss
dssall	vec_dssall	vec_dssall
dst	vec_dst	vec_dst
dstst	vec_dstst	vec_dstst
dststt	vec_dststt	vec_dststt
dstt	vec_dstt	vec_dstt
lvebx	vec_lvebx	vec_lde
lvehx	vec_lvehx	vec_lde
lvewx	vec_lvewx	vec_lde
lvsl	vec_lvsl	vec_lvsl
lvsr	vec_lvsr	vec_lvsr
lvx	vec_lvx	vec_ld
lvxl	vec_lvxl	vec_lvxl
mfvscr	vec_mfvscr	vec_mfvscr
mtvscr	vec_mtvscr	vec_mtvscr
stvebx	vec_stvebx	vec_ste
stvehx	vec_stvehx	vec_ste
stvewx	vec_stvewx	vec_ste

Table A-1. Instructions to Operations/Predicates Cross-Reference (Continued)

AltiVec Instruction	Specific Operation	Generic Operation/Predicate
stvx	vec_stvx	vec_st
stvxl	vec_stvxl	vec_stl
vaddcuw	vec_vaddcuw	vec_addc
vaddfp	vec_vaddfp	vec_add
vaddsbs	vec_vaddsbs	vec_adds
vaddshs	vec_vaddshs	vec_adds
vaddsws	vec_vaddsws	vec_adds
vaddubm	vec_vaddubm	vec_add
vaddubs	vec_vaddubs	vec_adds
vadduhm	vec_vadduhm	vec_add
vadduhs	vec_vadduhs	vec_adds
vadduwm	vec_vadduwm	vec_add
vadduws	vec_vadduws	vec_adds
vand	vec_vand	vec_and
vandc	vec_vandc	vec_andc
vavgsb	vec_vavgsb	vec_avg
vavgsh	vec_vavgsh	vec_avg
vavgsw	vec_vavgsw	vec_avg
vavgub	vec_vavgub	vec_avg
vavguh	vec_vavguh	vec_avg
vavguw	vec_vavguw	vec_avg
vcfsx	vec_vcfsx	vec_ctf
vcfux	vec_vcfux	vec_ctf
vcmpbfpx	vec_vcmpbfpx	vec_cmpb
vcmpbfp.	—	vec_all_in, vec_any_out
vcmpeqfx	vec_vcmpeqfx	vec_cmpeq
vcmpeqfp.	—	vec_all_eq, vec_all_nan, vec_all_ne, vec_all_numeric, vec_any_eq, vec_any_nan, vec_any_ne, vec_any_numeric
vcmpequbx	vec_vcmpequbx	vec_cmpeq
vcmpequb.	—	vec_all_eq, vec_all_ne, vec_any_eq, vec_any_ne
vcmpequhx	vec_vcmpequhx	vec_cmpeq

Table A-1. Instructions to Operations/Predicates Cross-Reference (Continued)

AltiVec Instruction	Specific Operation	Generic Operation/Predicate
vcmpequh.	—	vec_all_eq, vec_all_ne, vec_any_eq, vec_any_ne
vcmpequwx	vec_vcmpequwx	vec_cmpeq
vcmpequw.	—	vec_all_eq, vec_all_ne, vec_any_eq, vec_any_ne
vcmpgefpx	vec_vcmpgefpx	vec_cmpge, vec_cmple
vcmpgefp.	—	vec_all_ge, vec_all_le, vec_all_nge, vec_all_nle, vec_any_ge, vec_any_le, vec_any_nge, vec_any_nle
vcmpgtfpx	vec_vcmpgtfpx	vec_cmpgt, vec_cmplt
vcmpgtfp.	—	vec_all_gt, vec_all_lt, vec_all_ngt, vec_all_nlt, vec_any_gt, vec_any_lt, vec_any_ngt, vec_any_nlt
vcmpgtsbx	vec_vcmpgtsbx	vec_cmpgt, vec_cmplt
vcmpgtsb.	—	vec_all_ge, vec_all_gt, vec_all_le, vec_all_lt, vec_any_ge, vec_any_gt, vec_any_le, vec_any_lt
vcmpgtshx	vec_vcmpgtshx	vec_cmpgt, vec_cmplt
vcmpgtsh.	—	vec_all_ge, vec_all_gt, vec_all_le, vec_all_lt, vec_any_ge, vec_any_gt, vec_any_le, vec_any_lt
vcmpgtswx	vec_vcmpgtswx	vec_cmpgt, vec_cmplt
vcmpgtsw.	—	vec_all_ge, vec_all_gt, vec_all_le, vec_all_lt, vec_any_ge, vec_any_gt, vec_any_le, vec_any_lt
vcmpgtubx	vec_vcmpgtubx	vec_cmpgt, vec_cmplt
vcmpgtub.	—	vec_all_ge, vec_all_gt, vec_all_le, vec_all_lt, vec_any_ge, vec_any_gt, vec_any_le, vec_any_lt
vcmpgtuhx	vec_vcmpgtuhx	vec_cmpgt, vec_cmplt
vcmpgtuh.	—	vec_all_ge, vec_all_gt, vec_all_le, vec_all_lt, vec_any_ge, vec_any_gt, vec_any_le, vec_any_lt
vcmpgtuwx	vec_vcmpgtuwx	vec_cmpgt, vec_cmplt
vcmpgtuw.	—	vec_all_ge, vec_all_gt, vec_all_le, vec_all_lt, vec_any_ge, vec_any_gt, vec_any_le, vec_any_lt
vctsxs	vec_vctsxs	vec_cts
vctuxs	vec_vctuxs	vec_ctu
vexptefp	vec_vexptefp	vec_expte

Table A-1. Instructions to Operations/Predicates Cross-Reference (Continued)

AltiVec Instruction	Specific Operation	Generic Operation/Predicate
vlogefp	vec_vlogefp	vec_loge
vmaddfp	vec_vmaddfp	vec_madd
vmaxfp	vec_vmaxfp	vec_max
vmaxsb	vec_vmaxsb	vec_max
vmaxsh	vec_vmaxsh	vec_max
vmaxsw	vec_vmaxsw	vec_max
vmaxub	vec_vmaxub	vec_max
vmaxuh	vec_vmaxuh	vec_max
vmaxuw	vec_vmaxuw	vec_max
vmhaddshs	vec_vmhaddshs	vec_madds
vmhraddshs	vec_vmhraddshs	vec_mradds
vminfp	vec_vminfp	vec_min
vminsb	vec_vminsb	vec_min
vminsh	vec_vminsh	vec_min
vminsw	vec_vminsw	vec_min
vminub	vec_vminub	vec_min
vminuh	vec_vminuh	vec_min
vminuw	vec_vminuw	vec_min
vmladduhm	vec_vmladduhm	vec_mladd
vmrghb	vec_vmrghb	vec_mergeh
vmrghh	vec_vmrghh	vec_mergeh
vmrghw	vec_vmrghw	vec_mergeh
vmrglb	vec_vmrglb	vec_mergel
vmrglh	vec_vmrglh	vec_mergel
vmrglw	vec_vmrglw	vec_mergel
vmsummbm	vec_vmsummbm	vec_msum
vmsumshm	vec_vmsumshm	vec_msum
vmsumshs	vec_vmsumshs	vec_msums
vmsumubm	vec_vmsumubm	vec_msum
vmsumuhm	vec_vmsumuhm	vec_msum
vmsumuhs	vec_vmsumuhs	vec_msums
vmulesb	vec_vmulesb	vec_mule

Table A-1. Instructions to Operations/Predicates Cross-Reference (Continued)

AltiVec Instruction	Specific Operation	Generic Operation/Predicate
vmulesh	vec_vmulesh	vec_mule
vmuleub	vec_vmuleub	vec_mule
vmuleuh	vec_vmuleuh	vec_mule
vmulosb	vec_vmulosb	vec_mulo
vmulosh	vec_vmulosh	vec_mulo
vmuloub	vec_vmuloub	vec_mulo
vmulouh	vec_vmulouh	vec_mulo
vnmsubfp	vec_vnmsubfp	vec_nmsub
vnor	vec_vnor	vec_nor
vor	vec_vor	vec_or
vperm	vec_vperm	vec_perm
vpkpx	vec_vpkpx	vec_packpx
vpkshss	vpkshss	vec_packs
vpkshus	vec_vpkshus	vec_packsu
vpkswss	vec_vpkswss	vec_packs
vpkswus	vec_vpkswus	vec_packsu
vpkuhum	vec_vpkuhum	vec_pack
vpkuhus	vec_vpkuhus	vec_packs, vec_packsu
vpkuwum	vec_vpkuwum	vec_pack
vpkuwus	vec_vpkuwus	vec_packs, vec_packsu
vrefp	vec_vrefp	vec_re
vrfim	vec_vrfim	vec_floor
vrfin	vec_vrfin	vec_round
vrfip	vec_vrfip	vec_ceil
vrfiz	vec_vrfiz	vec_trunc
vrlb	vec_vrlb	vec_rl
vrlh	vec_vrlh	vec_rl
vrlw	vec_vrlw	vec_rl
vrsqrtefp	vec_vrsqrtefp	vec_rsqrte
vsel	vec_vsel	vec_sel
vsl	vec_vsl	vec_sll
vslb	vec_vslb	vec_sl

Table A-1. Instructions to Operations/Predicates Cross-Reference (Continued)

AltiVec Instruction	Specific Operation	Generic Operation/Predicate
vsldoi	vec_vsldoi	vec_sld
vslh	vec_vslh	vec_sl
vslo	vec_vslo	vec_slo
vslw	vec_vslw	vec_sl
vspltb	vec_vspltb	vec_splat
vsplth	vec_vsplth	vec_splat
vspltisb	vec_vspltisb	vec_splat_s8, vec_splat_u8
vspltish	vec_vspltish	vec_splat_s16, vec_splat_u16
vspltisw	vec_vspltisw	vec_splat_s32, vec_splat_u32
vspltw	vec_vspltw	vec_splat
vsr	vec_vsr	vec_srl
vsrab	vec_vsrab	vec_sra
vsrah	vec_vsrah	vec_sra
vsraw	vec_vsraw	vec_sra
vsrb	vec_vsrb	vec_sr
vsrh	vec_vsrh	vec_sr
vsro	vec_vsro	vec_sro
vsrw	vec_vsrw	vec_sr
vsubcuw	vec_vsubcuw	vec_subc
vsubfp	vec_vsubfp	vec_sub
vsubsbs	vec_vsubsbs	vec_subs
vsubshs	vec_vsubshs	vec_subs
vsubsws	vec_vsubsws	vec_subs
vsububm	vec_vsububm	vec_sub
vsububs	vec_vsububs	vec_subs
vsubuhm	vec_vsubuhm	vec_sub
vsubuhs	vec_vsubuhs	vec_subs
vsubuwm	vec_vsubuwm	vec_sub
vsubuws	vec_vsubuws	vec_subs
vsumsws	vec_vsumsws	vec_sums
vsum2sws	vec_vsum2sws	vec_sum2s
vsum4sbs	vec_vsum4sbs	vec_sum4s

Table A-1. Instructions to Operations/Predicates Cross-Reference (Continued)

AltiVec Instruction	Specific Operation	Generic Operation/Predicate
vsum4shs	vec_vsum4shs	vec_sum4s
vsum4ubs	vec_vsum4ubs	vec_sum4s
vupkhpx	vec_vupkhpx	vec_unpackh
vupkhsb	vec_vupkhsb	vec_unpackh
vupkhsh	vec_vupkhsh	vec_unpackh
vupklpx	vec_vupklpx	vec_unpackl
vupklsb	vec_vupklsb	vec_unpackl
vupklsh	vec_vupklsh	vec_unpackl
vxor	vec_vxor	vec_xor

Table A-2 lists the vector operations cross-referenced to the AltiVec instructions.

Table A-2. Operations to Instructions Cross-Reference

Specific Operation	AltiVec Instruction(s)
vec_abs	vspltisb, vsububm, vmaxsb
	vspltisb, vsubuhm, vmaxsh
	vspltisb, vsubuwm, vmaxsw
	vspltisw, vslw, vandc
vec_abss	vspltisb, vsubsbs, vmaxsb
	vspltisb, vsubshs, vmaxsh
	vspltisb, vsubsws, vmaxsw
vec_add	vaddfp
	vaddubm
	vadduhm
	vadduwm
vec_addc	vaddcuw
vec_adds	vaddsbs
	vaddshs
	vaddsws
	vaddubs
	vadduhs
	vadduws
vec_and	vand

Table A-2. Operations to Instructions Cross-Reference (Continued)

Specific Operation	AltiVec Instruction(s)
vec_andc	vandc
vec_avg	vavgsb
	vavgsh
	vavgsw
	vavgub
	vavguh
	vavguw
vec_ceil	vrfip
vec_cmpb	vcmpbfpx
vec_cmpeq	vcmpeqfx
	vcmpequbx
	vcmpequhx
	vcmpequwx
vec_cmpge	vcmpgefpx
vec_cmpgt	vcmpgtfpx
	vcmpgtsbx
	vcmpgtshx
	vcmpgtswx
	vcmpgtubx
	vcmpgtuhx
	vcmpgtuwx
vec_cmple	vcmpgefpx
vec_cmplt	vcmpgtfpx
	vcmpgtsbx
	vcmpgtshx
	vcmpgtswx
	vcmpgtubx
	vcmpgtuhx
	vcmpgtuwx
vec_ctf	vcfsx
	vcfux
vec_cts	vctsxs

Table A-2. Operations to Instructions Cross-Reference (Continued)

Specific Operation	AltiVec Instruction(s)
vec_ctu	vctuxs
vec_dss	dss
vec_dssall	dssall
vec_dst	dst
vec_dstst	dstst
vec_dststt	dststt
vec_dstt	dstt
vec_expte	vexptefp
vec_floor	vrfim
vec_ld	lvx
vec_lde	lvebx
	lvehx
	lvewx
vec_ldl	lvxl
vec_loge	vlogefp
vec_lvsl	lvsl
vec_lvsr	lvsr
vec_madd	vmaddfp
vec_madds	vmhaddshs
vec_max	vmaxfp
	vmaxsb
	vmaxsh
	vmaxsw
	vmaxub
	vmaxuh
	vmaxuw
vec_mergeh	vmrghw
	vmrghb
	vmrghh
vec_mergel	vmrglw
	vmrglb
	vmrglh

Table A-2. Operations to Instructions Cross-Reference (Continued)

Specific Operation	AltiVec Instruction(s)
vec_mfvscr	mfvscr
vec_min	vminfp
	vminsb
	vminsh
	vminsw
	vminub
	vminuh
	vminuw
vec_mladd	vmladduhm
vec_mradds	vmhraddshs
vec_msum	vmsummbm
	vmsumshm
	vmsumubm
	vmsumuhm
vec_msums	vmsumshs
vec_msums	vmsumuhs
vec_mtvscr	mtvscr
vec_mule	vmulesb
	vmulesh
	vmuleub
	vmuleuh
vec_mulo	vmulosb
	vmulosh
	vmuloub
	vmulouh
vec_nmsub	vnmsubfp
vec_nor	vnor
vec_or	vor
vec_pack	vpkuhum
	vpkuwum
vec_packpx	vpkpx

Table A-2. Operations to Instructions Cross-Reference (Continued)

Specific Operation	AltiVec Instruction(s)
vec_packs	vpkshss
	vpkswss
	vpkuhus
	vpkuwus
vec_packsu	vpkuhus
	vpkuwus
	vpkshus
	vpkswus
vec_perm	vperm
vec_re	vrefp
vec_rl	vrlb
	vrlh
	vrlw
vec_round	vrfin
vec_rsqrte	vrsqrtefp
vec_sel	vsel
vec_sl	vslb
	vslh
	vslw
vec_sld	vsldoi
vec_sll	vsl
vec_slo	vslo
vec_splat	vspltb
	vsplth
	vspltw
vec_splat_s16	vspltish
vec_splat_s32	vspltisw
vec_splat_s8	vspltisb
vec_splat_u16	vspltish
vec_splat_u32	vspltisw
vec_splat_u8	vspltisb

Table A-2. Operations to Instructions Cross-Reference (Continued)

Specific Operation	AltiVec Instruction(s)
vec_sr	vsrb
	vsrh
	vsrw
vec_sra	vsrab
	vsrah
	vsraw
vec_srl	vsr
vec_sro	vsro
vec_st	stvx
vec_ste	stvebx
	stvehx
	stvewx
vec_stl	stvxl
vec_sub	vsubfp
	vsububm
	vsubuhm
	vsubuwm
vec_subc	vsubcuw
vec_subs	vsubsbs
	vsubshs
	vsubsws
	vsububs
	vsubuhs
	vsubuws
vec_sum2s	vsum2sws
vec_sum4s	vsum4sbs
	vsum4shs
	vsum4ubs
vec_sums	vsumsws
vec_trunc	vrfiz

Table A-2. Operations to Instructions Cross-Reference (Continued)

Specific Operation	AltiVec Instruction(s)
vec_unpackh	vupkhpx
	vupkhsb
	vupkhsh
vec_unpackl	vupklpx
	vupklsb
	vupklsh
vec_xor	vxor

Table A-3 lists the predicates cross-referenced to the AltiVec instructions.

Table A-3. Predicate to Instruction Cross-Reference

Predicate	AltiVec Instruction
vec_all_eq	vcmpeqfp.
	vcmpequb.
	vcmpequh.
	vcmpequw.
vec_all_ge	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgefp.
vec_all_gt	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgtfp.
vec_all_in	vcmpbfp.
vec_all_le	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgefp.

Table A-3. Predicate to Instruction Cross-Reference (Continued)

Predicate	AltiVec Instruction
vec_all_lt	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgtfp.
vec_all_nan	vcmpeqfp.
vec_all_ne	vcmpeqfp.
	vcmpequb.
	vcmpequh.
	vcmpequw.
vec_all_nge	vcmpgefp.
vec_all_ngt	vcmpgtfp.
vec_all_nle	vcmpgefp.
vec_all_nlt	vcmpgtfp.
vec_all_numeric	vcmpeqfp.
vec_any_eq	vcmpeqfp.
	vcmpequb.
	vcmpequh.
	vcmpequw.
vec_any_ge	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgefp.

Table A-3. Predicate to Instruction Cross-Reference (Continued)

Predicate	AltiVec Instruction
vec_any_gt	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgtfp.
vec_any_le	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgefp.
vec_any_lt	vcmpgtsb.
	vcmpgtsh.
	vcmpgtsw.
	vcmpgtub.
	vcmpgtuh.
	vcmpgtuw.
	vcmpgtfp.
vec_any_nan	vcmpeqfp.
vec_any_ne	vcmpeqfp.
	vcmpequb.
	vcmpequh.
	vcmpequw.
vec_any_nge	vcmpgefp.
vec_any_ngt	vcmpgtfp.
vec_any_nle	vcmpgefp.
vec_any_nlt	vcmpgtfp.
vec_any_numeric	vcmpeqfp.
vec_any_out	vcmpbfp.

Glossary of Terms and Abbreviations

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from *IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

Note that some terms are defined in the context of how they are used in this book.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

B

Biased exponent. An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address n of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most-significant byte. *See* Little-endian.

C

Cache. High-speed memory component containing recently-accessed data and/or instructions (subset of main memory).

Cast. A cast expression consists of a left parenthesis, a type name, a right parenthesis, and an operand expression. The cast causes the operand value to be converted to the type name within the parentheses.

D

Denormalized number. A nonzero floating-point number whose *exponent* has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

E

Effective address (EA). The 32- or 64-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. *See also* Biased exponent.

F

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs and store instructions move data from FPRs to memory. The FPRs are 64 bits wide and store floating-point vlaues in double-precision format.

Fraction. In the binary representation of a floating-point number, the field of the *significand* that lies to the right of its implied binary point.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

L

Least-significant bit (lsb). The bit of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. *See* Big-endian.

M

Mnemonic. The abbreviated name of an instruction used for coding.

Modulo. A value *v* which lies outside the range of numbers representable by an n-bit wide destination type is replaced by the low-order n bits of the two's complement representation of *v*.

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for 'Not a Number'; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs (SNaNs) and quiet NaNs (QNaNs).

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

Overflow. An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits.

Q

Quad word. A group of 16 contiguous locations starting at an address divisible by 16.

Quiet NaN. A type of *NaN* that can propagate through most arithmetic operations without signaling exceptions. A quiet NaN is used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. *See* Signaling NaN.

R

Record bit. Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation. Its presence is denoted by a "." following the mnemonic.

Reserved field. In a register, a reserved field is one that is not assigned a function. A reserved field may be a single bit. The handling of reserved bits is *implementation-dependent*. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

RISC (reduced instruction set computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Saturate. A value *v* which lies outside the range of numbers representable by a destination type is replaced by the representable number closest to *v*.

Signaling NaN. A type of *NaN* that generates an invalid operation program exception when it is specified as arithmetic operands. *See* Quiet NaN.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Splat. A splat instruction will take one element and replicate (splat) that value into a vector register.

Sticky bit. A bit that when *set* must be cleared explicitly.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

T

Tiny. A floating-point value that is too small to be represented for a particular precision format, including *denormalized* numbers; they do not include ±0.

U

Underflow. An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or mantissa than the single-precision format can provide. In other words, the result is too small to be represented accurately.

User mode. The unprivileged operating state of a processor used typically by application software. In user mode, software can only access certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

Vector Literal. A vector literal is a constant expression with a value that is taken as a vector type. See Section 2.5.1, “Vector Literals” for details.

Vector Register (VR). Any of the 32 registers in the vector register file. Each vector register is 128 bits wide. These registers can provide the source operands and destination results for AltiVec instructions.

W

Word. A 32-bit data element.

INDEX

INDEX

INDEX

INDEX

INDEX

W

X

Attention!

This book is a companion to the *PowerPC Microprocessor Family: The Programming Environments*, referred to as *The Programming Environments Manual*. Note that the companion *Programming Environments Manual* exists in two versions. See the Preface for a description of the following two versions:

- *PowerPC Microprocessor Family: The Programming Environments,* Rev 1
  Order #: MPCFPE/AD
- *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, Rev 1*
  Order #: MPCFPE32B/AD

Call the Motorola LDC at 1-800-441-2447 or contact your local sales office to obtain copies.