ALTIVECPEM/D
11/1998
Rev. 0.1

AltiVec™ Technology Programming Environments Manual

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

CONTENTS

Paragraph Number	Title	Page Number

About This Book

Chapter 1 Overview

Chapter 2 AltiVec Register Set

CONTENTS

Paragraph Number	Title	Page Number

CONTENTS

Paragraph Number	Title	Page Number

Chapter 4
Addressing Modes and Instruction Set Summary

CONTENTS

Paragraph Number	Title	Page Number

ILLUSTRATIONS

Figure Number	Title	Page Number

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

Figure Number	Title	Page Number

TABLES

Table Number	Title	Page Number

TABLES

About This Book

The primary objective of this manual is to help programmers provide software that is compatible with the family of PowerPC™ processors using AltiVec™ technology. This book describes how the AltiVec technology relates to both the 64- and the 32-bit portions of the PowerPC architecture.

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

AltiVec Technology Programming Environments Manual (Altivec PEM*)* is used as a reference guide for programmers. The AltiVec PEM provides a description for each instruction that includes the instruction format and figures to help in understanding how the instruction works.

Because it is important to distinguish between the levels of the PowerPC architecture in order to ensure compatibility across multiple platforms, those distinctions are shown clearly throughout this book. Most the discussions on the AltiVec technology are at the UISA level. The level of the architecture to which text refers is indicated in the outer margin, using the conventions shown in "Conventions," on page xxii.

This document stays consistent with the PowerPC architecture in referring to three levels, or programming environments, which are as follows:

U

- PowerPC user instruction set architecture (UISA)—The UISA defines the level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers.

V

- PowerPC virtual environment architecture (VEA)—The VEA, which is the smallest component of the PowerPC architecture, defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory, and defines aspects of the cache model and cache control instructions from a user-level perspective. The resources defined by the VEA are particularly useful for optimizing memory accesses and for managing resources in an environment in which other processors and other devices can access external memory.

  Implementations that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

- PowerPC operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system. The OEA defines the PowerPC memory management model, supervisor-level registers, and the exception model.

  Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

For ease in reference, this book and the processor user's manuals have arranged the architecture information into topics that build upon one another, beginning with a description and complete summary of registers and instructions (for all three environments) and progressing to more specialized topics such as the cache, exception, and memory management models. As such, chapters may include information from multiple levels of the architecture but when discussing OEA and VEA, this will be noted in the text.

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 web site at: http://www.mot.com/SPS/PowerPC/.

Audience

This manual is intended for system software and hardware developers 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, and the basic principles of RISC processing.

This book describes how the AltiVec technology interacts with both the 64- and the 32-bit portions of the PowerPC architecture

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 the features and functions of the AltiVec technology. This chapter provides an overview of how the AltiVec technology defines the register set, operand conventions, addressing modes, instruction set, cache model, and exception model.
- Chapter 2, "AltiVec Register Set," is useful for software engineers who need to understand the PowerPC programming model for the three programming environments. The chapter also discusses the functionality of the AltiVec technology registers and how they interact with the other PowerPC registers.

- Chapter 3, “Operand Conventions,” describes how the AltiVec technology interacts with the PowerPC conventions for storing data in memory, including information regarding alignment, single-precision floating-point conventions, and big- and little-endian byte ordering.
- Chapter 4, “Addressing Modes and Instruction Set Summary,” provides an overview of the AltiVec technology addressing modes and a brief description of the AltiVec technology instructions organized by function.
- Chapter 5, “Cache, Exceptions, and Memory Management,” provides a discussion of the cache and memory model defined by the VEA and aspects of the cache model that are defined by the OEA. It also describes the exception model defined in the UISA.
- Chapter 6, “AltiVec Instructions,” functions as a handbook for the AltiVec instruction set. Instructions are sorted by mnemonic. Each instruction description includes the instruction formats and figures where it helps in understanding what the instruction does.
- Appendix A, “AltiVec Instruction Set Listings,” lists all the AltiVec instructions. Instructions are grouped according to mnemonic, opcode, and form.
- 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.

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 Architecture: A Quantitative Approach,* Second Edition, by John L. Hennessy and David A. Patterson

- *Inside Macintosh: PowerPC System Software*, Addison-Wesley Publishing Company, One Jacob Way, Reading, MA, 01867; Tel. (800) 282-2732 (U.S.A.), (800) 637-0029 (Canada), (716) 871-6555 (International).
- *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).

PowerPC Documentation

The PowerPC documentation is available from the sources listed on the back cover of this manual; the document order numbers are included in parentheses for ease in ordering:

- User's manuals—These books provide details about individual PowerPC implementations and are intended to be used in conjunction with *The Programming Environments Manual*. These include the following:
  - *PowerPC 601™ RISC Microprocessor User's Manual*: MPC601UM/AD (Motorola order #)
  - *PowerPC 603e™ RISC Microprocessor User's Manual with Supplement for PowerPC 603 Microprocessor*: MPC603EUM/AD (Motorola order #)
  - *PowerPC 604™ RISC Microprocessor User's Manual*: MPC604UM/AD (Motorola order #)
- Programming environments manuals—These books provide information about resources defined by the PowerPC architecture that are common to PowerPC processors. There are two versions, one that describes the functionality of the combined 32- and 64-bit architecture models and one that describes only the 32-bit model.
  - *PowerPC Microprocessor Family: The Programming Environments*, Rev 1: MPCFPE/AD (Motorola order #)
  - *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, Rev. 1: MPCFPE32B/AD (Motorola order #)
- *Implementation Variances Relative to Rev. 1 of The Programming Environments Manual* is available via the world-wide web at http://www.motorola.com/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. These include the following:
  - *Addendum to PowerPC 603e RISC Microprocessor User's Manual: PowerPC 603e Microprocessor Supplement and User's Manual Errata:* MPC603EUMAD/AD (Motorola order #)
  - *Addendum to PowerPC 604 RISC Microprocessor User's Manual*: *PowerPC 604e™ Microprocessor Supplement and User's Manual Errata*: MPC604UMAD/AD (Motorola order #)

- 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. These include the following:
  - — *PowerPC 601 RISC Microprocessor Hardware Specifications*:
    MPC601EC/D (Motorola order #)
  - — *PowerPC 603 RISC Microprocessor Hardware Specifications*:
    MPC603EC/D (Motorola order #)
  - — *PowerPC 603e RISC Microprocessor Family: PID6-603e Hardware Specifications*:
    MPC603EEC/D (Motorola order #)
  - — *PowerPC 603e RISC Microprocessor Family: PID7V-603e Hardware Specifications*:
    MPC603E7VEC/D (Motorola order #)
  - — *PowerPC 603e RISC Microprocessor Family: PID7t-603e Hardware Specifications*:
    MPC603E7TEC/D (Motorola order #)
  - — *PowerPC 604 RISC Microprocessor Hardware Specifications*:
    MPC604EC/D (Motorola order #)
  - — *PowerPC 604e RISC Microprocessor Family: PID9V-604e Hardware Specifications*:
    MPC604E9VEC/D (Motorola order #)
- 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. Technical summaries are available for the 601, 603, 603e, 604, and 604e as well as the following:
  - — *PowerPC 620™ RISC Microprocessor Technical Summary*:
    MPC620/D (Motorola order #)
  - — *PowerPC 750 RISC Microprocessor Technical Summary*:
    MPC750/D (Motorola order #)
- *PowerPC Microprocessor Family: The Bus Interface for 32-Bit Microprocessors*: MPCBUSIF/AD (Motorola order #) provides a detailed functional description of the 60x bus interface, as implemented on the 601, 603, and 604 family of PowerPC microprocessors. This document is intended to help system and chipset developers by providing a centralized reference source to identify the bus interface presented by the 60x family of PowerPC microprocessors.
- *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.
- Documentation for support chips—These include the following:
  - *MPC105 PCI Bridge/Memory Controller User's Manual*: MPC105UM/AD (Motorola order #)
  - *MPC106 PCI Bridge/Memory Controller User's Manual*: MPC106UM/AD (Motorola order #)

Additional literature on PowerPC implementations is being released as new processors become available. For a current list of PowerPC documentation, refer to the world-wide web at http://www.motorola.com/PowerPC/.

Conventions

Throughout the documentation when a register or bit is "set" it means the register or bit is set to 1, and when a register is "cleared" it means the register or bit is set to 0.

This document uses the following notational conventions:

mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics indicate variable command parameters, for example, **bcctr***x*. Book titles in text are set in italics.
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify a source GPR
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
REG[FIELD]	Abbreviations or acronyms for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
vA, **v**B, **v**C	Instruction syntax used to identify a source VR
vD	Instruction syntax used to identify a destination VR
x	In certain contexts, such as a signal encoding, this indicates a don't care.

n	Used to express an undefined numerical value
¬	NOT logical operator
&	AND logical operator
|	OR logical operator
U	This symbol identifies text that is relevant with respect to the PowerPC user instruction set architecture (UISA). This symbol is used both for information that can be found in the UISA specification as well as for explanatory information related to that programming environment.
V	This symbol identifies text that is relevant with respect to the PowerPC virtual environment architecture (VEA). This symbol is used both for information that can be found in the VEA specification as well as for explanatory information related to that programming environment.
O	This symbol identifies text that is relevant with respect to the PowerPC operating environment architecture (OEA). This symbol is used both for information that can be found in the OEA specification as well as for explanatory information related to that programming environment.
0 0 0 0	Indicates reserved bits or bit fields in a register. Although these bits may be written to as either ones or zeros, they are always read as zeros.

Additional conventions used with instruction encodings are described in Section 6.1, “Instruction Formats.”

Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this document. Note that the meanings for some acronyms (such as SDR1 and XER) are historical, and the words for which an acronym stands may not be intuitively obvious.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
ALU	Arithmetic logic unit
ASR	Address space register
BAT	Block address translation
BPU	Branch processing unit
CR	Condition register
CTR	Count register
DAR	Data address register

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
DEC	Decrementer register
DSISR	Register used for determining the source of a DSI exception
EA	Effective address
ECC	Error checking and correction
FPR	Floating-point register
FPSCR	Floating-point status and control register
FPU	Floating-point unit
GPR	General-purpose register
IEEE	Institute of Electrical and Electronics Engineers
ITLB	Instruction translation lookaside buffer
IU	Integer unit
L2	Secondary cache
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
LSQ	Least-significant quad-word
lsq	Least-significant quad-word
MERSI	Modified/exclusive/reserved/shared/invalid—cache coherency protocol
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSQ	Most-significant quad-word
msq	Most-significant quad-word
MSR	Machine state register
NaN	Not a number
NIA	Next instruction address
No-op	No operation
OEA	Operating environment architecture
PTEG	Page table entry group
RISC	Reduced instruction set computing

Table i. Acronyms and Abbreviated Terms (Continued)

Term	Meaning
RTL	Register transfer language
RWITM	Read with intent to modify
SIMM	Signed immediate value
SPR	Special-purpose register
SR	Segment register
SRR0	Machine status save/restore register 0
SRR1	Machine status save/restore register 1
STE	Segment table entry
TB	Time base register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value
UISA	User instruction set architecture
VA	Virtual address
VEA	Virtual environment architecture
VR	Vector register

Terminology Conventions

Table ii lists certain terms used in this manual that differ from the architecture terminology conventions.

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Data storage interrupt (DSI)	DSI exception
Extended mnemonics	Simplified mnemonics
Instruction storage interrupt (ISI)	ISI exception
Interrupt	Exception
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access
Swizzling	Double-word swap

Table iii describes instruction field notation conventions used in this manual.

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM
VA, VB, VT, VS	**v**A, **v**B, **v**D, **v**S (respectively)
VEC	AltiVec technology
/, //, ///	0...0 (shaded)

Chapter 1
Overview

The AltiVec™ technology provides a software model that accelerates the performance of various software applications and runs on reduced instruction set computing (RISC) microprocessors. The AltiVec technology extends the instruction set architecture (ISA) of the PowerPC architecture. AltiVec technology is a short vector parallel architecture. The AltiVec ISA is based on separate vector/SIMD-style (single instruction stream, multiple data streams) execution units that have high data parallelism. That is, the AltiVec technology operations can perform on multiple data elements in a single instruction. The term‘vector’ in this document refers to the spatial parallel processing of short, fixed-length one-dimensional matrices performed by an execution unit. It should not be confused with the temporal parallel (pipelined) processing of long, variable-length vectors performed by classical vector machines. High degrees of parallelism are achievable with simple in-order instruction dispatch and low-instruction bandwidth. However, the ISA is designed so as not to impede additional parallelism through superscalar dispatch to multiple execution units or multithreaded execution unit pipelines.

All instructions are designed to be easily pipelined with pipeline latencies no greater than scalar, double-precision, floating-point multiply-add. No instruction specifies an operation that presents a frequency limitation beyond those already imposed by existing PowerPC instructions. There are no operating mode switches which preclude fine grain interleaving of instructions with the existing floating-point and integer instructions. Parallelism with the integer and floating-point instructions is simplified by the fact that the vector unit never generates an exception and has few shared resources or communication paths that require it to be tightly synchronized with the other units. By using the SIMD parallelism, performance can be accelerated on PowerPC processors to a level that can allow concurrent real-time processing of one or more data streams.

In this document, the term ‘implementation’ refers to a hardware device (typically a microprocessor) that complies with the PowerPC architecture.

The AltiVec technology can be used as an extension to various RISC microprocessors; however, in this book it is discussed within the context of the PowerPC architecture described as follows:

- Programming model
  - Instruction set. The AltiVec instruction set specifies instructions that extend the PowerPC instruction set. These instructions are organized similar to PowerPC instructions (such as vector load/store, vector integer, and vector floating-point instructions). The specific instructions, and the forms used for encoding them, are provided in Appendix A, “Instruction Set.”
  - Register set. The AltiVec programming model defines new AltiVec registers, additions to the PowerPC register set, and how existing PowerPC registers are affected by the AltiVec technology. The model also discusses memory conventions, including details regarding the byte ordering for quad words.
- Memory model. The AltiVec technology specifies additional cache management instructions. That is, a program can execute AltiVec software instructions that indicate when a sequence of memory units (data stream/stream) are likely to be accessed.
- Exception model. To ensure efficiency, the AltiVec technology provides only an AltiVec unavailable interrupt (VUI) exception, a DSI exception, and trace exception (if implemented). There are no exceptions other than DSI exceptions on loads and stores. The AltiVec instructions can cause PowerPC exceptions.
- Memory management model. The memory model for the AltiVec technology is the same as it is implemented for the PowerPC architecture. AltiVec memory accesses are always assumed to be aligned. If an operand is unaligned, additional AltiVec instructions are used to ensure that it is correctly placed in a vector register or in memory.
- Time-keeping model—The PowerPC time-keeping model is not impacted by the AltiVec technology.

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

This chapter provides an overview of the major characteristics of the AltiVec technology in the order in which they are addressed in this book:

- Register set and programming model
- Instruction set and addressing modes
- Cache, exceptions, and memory management

1.1 Overview

The AltiVec technology’s SIMD-style extension provides an approach to accelerating the processing of data streams. Using the AltiVec instructions can provide a significant speedup for communications, multimedia, and other performance-driven applications by using data-level parallelism where available, matching scalar performance in serial sections of media applications, keeping media processing within the AltiVec unit, and minimizing bandwidth and latency memory access bottlenecks.

AltiVec technology expands the PowerPC architecture through the addition of a 128-bit vector execution unit, which operates concurrently with the existing integer- and floating-point units. A new vector execution unit provides highly parallel operations, allowing for simultaneous execution of multiple operations in a single clock cycle.

The AltiVec technology can be thought of as a set of registers and execution units that can be added to the PowerPC architecture in a manner analogous to the addition of floating-point units. Floating-point units were added to provide support for high-precision scientific calculations and the AltiVec technology is added to the PowerPC architecture to accelerate the next level of performance-driven, high-bandwidth communications and computing applications. Figure 1-1 provides the high level structural overview for PowerPC with the AltiVec technology.

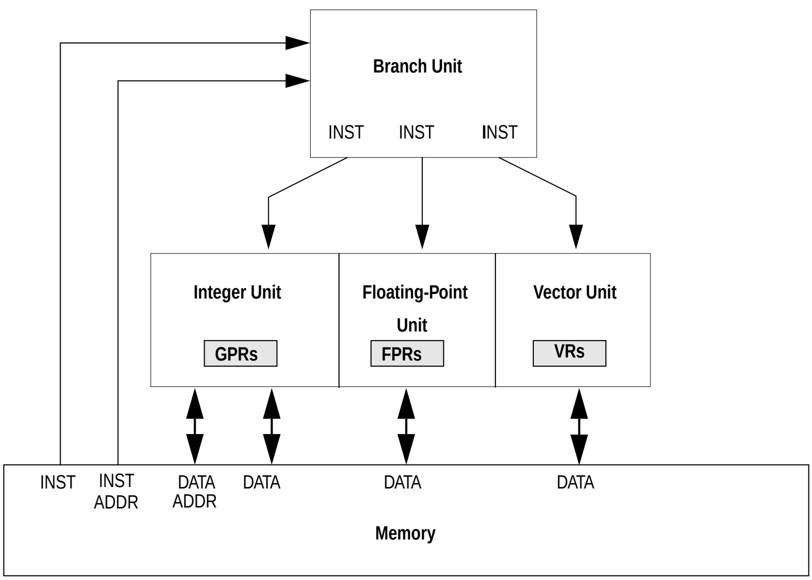


Figure 1-1. High Level Structural Overview of PowerPC with AltiVec Technology

The AltiVec technology is purposefully simple such that there are no exceptions other than DSI exceptions on loads and stores, no hardware unaligned access support, and no complex functions. The AltiVec technology is scaled down to only the necessary pieces in order to facilitate efficient cycle time, latency, and throughput on hardware implementations.

The AltiVec technology defines the following:

- Fixed 128-bit wide vector length that can be subdivided into sixteen 8-bit bytes, eight 16-bit half words, or four 32-bit words
- Vector register file (VRF) architecturally separate from floating-point registers (FPRs) and general-purpose registers (GPRs)
- Vector integer and floating-point arithmetic
- Four operands for most instructions (three source operands and one result)
- Saturation clamping, (that is, unsigned results are clamped to zero on underflow and to the maximum positive integer value (2^n-1, for example, 255 for byte fields) on overflow. For signed results, saturation clamps results to the smallest representable negative number (-2^{n-1}, for example, -128 for byte fields) on underflow, and to the largest representable positive number (2^{n-1}-1, for example, +127 for byte fields) on overflow)
- No mode switching that would increase the overhead of using the instructions
- Operations selected based on utility to digital signal processing algorithms (including 3D).
- AltiVec instructions provide a vector compare and select mechanism to implement conditional execution as the preferred way to control data flow in AltiVec programs
- Enhanced cache/memory interface

The AltiVec ISA supports the following:

- Voice over IP (VoIP). VoIP transmits voice as compressed digital data packets over the internet.
- Access Concentrators/DSLAMS. An access concentrator strips data traffic off of POTS lines and inserts it into the INet. Digital subscriber loop access multiplexer (DSLAM) pulls data off at a switch and immediately routes it to the Internet. This allows to concentrate ADSL digital traffic at the switch and off-load the network.
- Speech recognition. Speech processing allows voice recognition for use in applications like directory assistance and automatic dialing.
- Voice/Sound Processing (Audio decode and encode): G.711, G.721, G.723, G,729A, and AC-3. Voice processing is used to improve sound quality on lines.
- Communications
  - — Multi-channel modems
  - — Software modem: V.34, 56K
  - — Data encryption: RSA
  - — Modem banks can use the AltiVec technology to replace signal processors in DSP farms.

- 2D and 3D graphics: QuickDraw, OpenGL, VRML, Games, Entertainment, High-precision CAD
- Virtual Reality
- High-fidelity audio: 3D audio, AC-3. Hi-Fi Audio uses AltiVec's FPU.
- Image and video processing: JPEG, Filters
- Echo cancellation. The echo cancellation is used to eliminate echo build up on long landline calls.
- Array number processing
- Basestation Processing. Cellular basestation compresses digital voice data for transmission within the Internet.
- High bandwidth data communication
- Motion video decode and encode: MPEG-1, MPEG-2, MPEG-4, and H.234
- Real-time continuous speech I/O: HMM, Viterbi acceleration, Neural algorithms
- Video conferencing: H.261, H.263
- Machine Intelligence

1.1.1 The 64-Bit AltiVec Technology and the 32-Bit Subset

The AltiVec technology supports the following modes of PowerPC operations:

- 64-bit implementations/64-bit mode—The AltiVec technology defines interactions with the PowerPC 64-bit registers.
- 64-bit implementations/32-bit mode—The AltiVec technology defines interaction with the conventions for 32-bit implementations of PowerPC registers.

For further details on the 64-bit PowerPC architecture and the 32-bit subset refer to Chapter 1, "Overview," in the *PowerPC Microprocessor Family: The Programming Environments Manual*.

This book describes both the 64-bit and 32-bit PowerPC architecture modes. Instructions are described from a 64-bit perspective and in most cases, details of the 32-bit subset can easily be determined from the 64-bit descriptions. Significant differences in the 32-bit subset are highlighted and described separately as they occur.

1.1.2 The Levels of the AltiVec ISA

The AltiVec ISA follows the layering of PowerPC architecture. The PowerPC architecture has three levels defined as follows:

- PowerPC user instruction set architecture (UISA) —The UISA defines the level of the architecture to which user-level (referred to as problem state in the architecture specification) software should conform. The UISA defines the base user-level U instruction set, user-level registers, data types, floating-point memory conventions and exception model as seen by user programs, and the memory and programming models. The icon shown in the margin identifies text that is relevant to the UISA.

- PowerPC virtual environment architecture (VEA)—The VEA defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time base facility from a user-level perspective. The icon shown in the margin identifies text that is relevant to the VEA.

  Implementations that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

- PowerPC operating environment architecture (OEA)—The OEA defines supervisor-level (referred to as privileged state in the architecture specification) resources typically required by an operating system. The OEA defines the PowerPC memory management model, supervisor-level registers, synchronization requirements, and the exception model. The OEA also defines the time base feature from a supervisor-level perspective. The icon shown in the margin identifies text that is relevant to the OEA.

The AltiVec technology defines instructions at the UISA and VEA levels. The distinctions between the levels is noted in the text throughout the document

1.1.3 Features Not Defined by the AltiVec ISA

Because flexibility is an important design goal of the AltiVec technology, there are many aspects of the microprocessor design, typically relating to the hardware implementation, that the AltiVec ISA does not define, for example, the number and the nature of execution units. The AltiVec ISA is a vector/SIMD architecture, and as such makes it easier to implement pipelining instructions and parallel execution units to maximize instruction throughput. However, the AltiVec ISA does not define the internal hardware details of implementations. For example, one processor may use a simple implementation having two vector execution units whereas another may provide a bigger, faster microprocessor design with several concurrently pipelined vector arithmetic logical units (ALUs) with separate load/store units (LSUs) and prefetch units.

1.2 The AltiVec Architectural Model

This section provides overviews of aspects defined by the AltiVec ISA, following the same order as the rest of this book. The topics are as follows:

- Registers and programming model
- Operand conventions
- Instruction set and addressing modes
- Cache model, exceptions, and memory management

1.2.1 AltiVec Registers and Programming Model

In the AltiVec technology, the ALU operates on from one to three source vectors and produces a single result/destination vector on each instruction. The ALU is a SIMD-style arithmetic unit that performs the same operation on all the data elements that comprise each vector. This scheme allows efficient code scheduling in a highly parallel processor. Load and store instructions are the only instructions that transfer data between registers and memory. The vector unit and vector register file are shown in Figure 1-2.

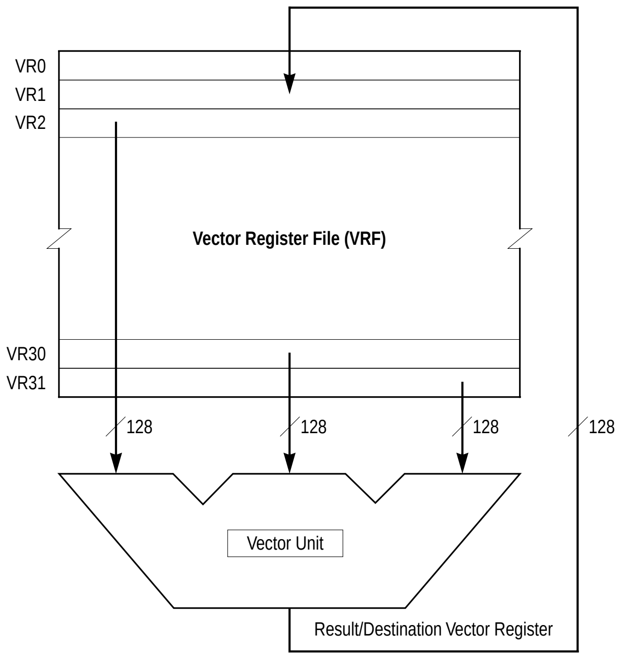


Figure 1-2. AltiVec Top-Level Diagram

The vector unit is a SIMD-style unit in which an instruction performs operations in parallel on the data elements that comprise each vector. Architecturally, the vector register file (VRF) is separate from the GPRs and FPRs. The AltiVec programming model incorporates the 32 registers of the VRF, each register is 128 bits wide.

1.2.2 Operand Conventions

Operand conventions define how data is stored in vector registers and memory.

1.2.2.1 Byte Ordering

The default mapping for AltiVec ISA is PowerPC big-endian, but AltiVec ISA provides the option of operating in either big- or little-endian mode. 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 1-3.

Quad Word															
Word 0				Word 1				Word 2				Word 3			
High Order Half Word 0		Low Order Half Word 1		Half Word 2		Half Word 3		Half Word 4		Half Word 5		Half Word 6		Half Word 7	
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 127

MSB (High Order) — LSB (Low Order)

Figure 1-3. Big-Endian Byte Ordering for a Vector Register

As shown in Figure 1-3, the elements in vector registers 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. That is, in Figure 1-3 the high-order half word for word 0 (bits 0–15), 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).

In big-endian mode, an AltiVec quad word load instruction for which the effective address (EA) is quad-word aligned places the byte addressed by EA into byte element 0 of the target vector register. The byte addressed by EA + 1 is placed in byte element 1, and so forth. Similarly, an AltiVec quad word store instruction for which the EA is quad word-aligned places byte element 0 of the source vector register into the byte addressed by EA. Byte element 1 is placed into the byte addressed by EA + 1, and so forth.

1.2.2.2 Floating-Point Conventions

The AltiVec ISA basically has two modes for floating-point, that is a Java-/IEEE-/C9X-compliant mode or a possibly faster non-Java/non-IEEE mode. AltiVec ISA conforms to the Java Language Specification 1 (hereafter referred to as Java), that is a subset of the default environment specified by the IEEE standard (ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic). For aspects of floating-point behavior that are not defined by Java but are defined by the IEEE standard, AltiVec ISA conforms to the IEEE standard. For aspects of floating-point behavior that are defined neither by Java nor by the IEEE standard but are defined by the C9X Floating-Point Proposal,WG14/N546 X3J11/96-010 (Draft 2/26/96) (hereafter referred to as C9X), AltiVec ISA conforms to C9X when in Java-compliant mode.

1.2.3 AltiVec Addressing Modes

As with PowerPC instructions, AltiVec instructions are encoded as single-word (32-bit) instructions. Instruction formats are consistent among all instruction types, permitting decoding to be parallel with operand accesses. This fixed instruction length and consistent format simplifies instruction pipelining. AltiVec load, store, and stream prefetch instructions use secondary opcodes in primary opcode 31 (0b011111). AltiVec ALU-type instructions use primary opcode point 4 (0b000100).

AltiVec ISA supports both intraelement and interelement operations. In an intraelement operation, elements work in parallel on the corresponding elements from multiple source operand registers and place the results in the corresponding fields in the destination operand register. An example of an intraelement operation is the Vector Add Signed Word Saturate (**vaddsws)** instruction shown in Figure 1-4.

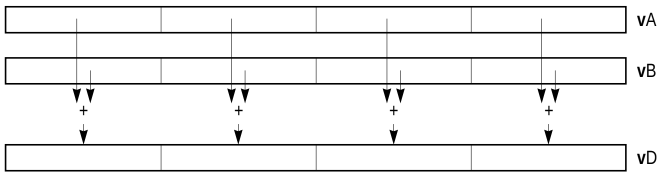


Figure 1-4. Intraelement Example, vaddsws

In this example, the four signed integer (32 bits) elements in register **v**A are added to the corresponding four signed integer (32 bits) elements in register **v**B and the four results are placed in the corresponding elements in register **v**D.

In interelement operations data paths cross over. That is, different elements from each source operand are used in the resulting destination operand. An example of an interelement operation is the Vector Permute **(vperm**) instruction shown in Figure 1-5.

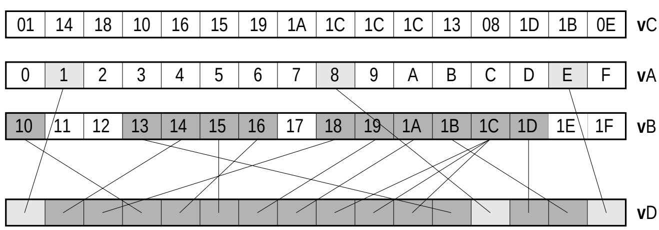


Figure 1-5. Interelement Example, vperm

In this example, **vperm** allows any byte in two source vector registers (**v**A and **v**B) to be copied to any byte in the destination vector register, **v**D. The bytes in a third source vector register (**v**C) specify from which byte in the first two source vector registers the corresponding target byte is to be copied. In this case the elements from the source vector registers do not have corresponding elements that operate on the destination register.

Most arithmetic and logical instructions are intraelement operations. The data paths for the ALU run primarily north and south with little crossover. The crossover data paths have been restricted as much as possible to the interelement manipulation instructions (unpack, pack, permute, etc.) with a vision toward implementing the ALU and shift/permute networks as separate execution units. The following list of instructions distinguishes between interelement and intraelement instructions:

- Vector intraelement instructions
  - Vector integer instructions
    - Vector integer arithmetic instructions
    - Vector integer compare instructions
    - Vector integer rotate and shift instructions
  - Vector floating-point instructions
    - Vector floating-point arithmetic instructions
    - Vector floating-point rounding and conversion instructions
    - Vector floating-point compare instruction
    - Vector floating-point estimate instructions
  - Vector memory access instructions
- Vector interelement instructions
  - Vector alignment support instructions
  - Vector permutation and formatting instructions
    - Vector pack instructions
    - Vector unpack instructions
    - Vector merge instructions
    - Vector splat instructions
    - Vector permute instructions
    - Vector shift left/right instructions

1.2.4 AltiVec Instruction Set

U Although these categories are not defined by the AltiVec ISA, the AltiVec instructions can be grouped as follows:

V
- Vector integer arithmetic instructions—These instructions are defined by the UISA. They include computational, logical, rotate, and shift instructions.
  - Vector integer arithmetic instructions
  - Vector integer compare instructions
  - Vector integer logical instructions
  - Vector integer rotate and shift instructions

- Vector floating-point arithmetic instructions—These include floating-point arithmetic instructions defined by the UISA.
  - Vector floating-point arithmetic instructions
  - Vector floating-point multiply/add instructions
  - Vector floating-point rounding and conversion instructions
  - Vector floating-point compare instruction
  - Vector floating-point estimate instructions
- Vector load and store instructions—These include load and store instructions for vector registers defined by the UISA.
- Vector permutation and formatting instructions—These instructions are defined by the UISA.
  - Vector pack instructions
  - Vector unpack instructions
  - Vector merge instructions
  - Vector splat instructions
  - Vector permute instructions
  - Vector select instructions
  - Vector shift instructions
- Processor control instructions—These instructions are used to read and write from the AltiVec status and control register (VSCR). These instructions are defined by the UISA.
- Memory control instructions—These instructions are used for managing of caches (user level and supervisor level). The instructions are defined by VEA. V

1.2.5 AltiVec Cache Model U

The AltiVec ISA defines several instructions for enhancements to cache management. These instructions allow software to indicate to the cache hardware how it should prefetch and prioritize writeback of data. The AltiVec ISA does not define hardware aspects of cache implementations. V

1.2.6 AltiVec Exception Model

The AltiVec vector unit never generates an exception. Data stream instructions will never cause an exception themselves. Therefore, on any event that would cause an exception on a normal load or store, such as a page fault or protection violation, the data stream instruction does not take a DSI exception; instead, it simply aborts and is ignored. Most AltiVec instructions do not generate any kind of exception. Vector load and store instructions that attempt to access a direct-store segment will cause a DSI exception.

The AltiVec unit does not report IEEE exceptions; there are no status flags and the unit has no architecturally visible traps. Default results are produced for all exception conditions as specified first by the Java specification. If no default exists, the IEEE standard's default is used. Then, if no default exists, the C9X default is used.

1.2.7 Memory Management Model

In a PowerPC processor the MMU's primary functions are to translate logical (effective) addresses to physical addresses for memory accesses and I/O accesses (most I/O accesses are assumed to be memory-mapped) and to provide access protection on a block or page basis. Some protection is also available even if translation is disabled. Typically, it is not programmable. The AltiVec ISA does not provide any additional instructions to the PowerPC memory management model, but the AltiVec instructions have options to ensure that an operand is correctly placed in a vector register or in memory.

Chapter 2
AltiVec Register Set

This chapter describes the register organization defined by the AltiVec technology. It also describes how AltiVec instructions affect some of the PowerPC registers. The AltiVec ISA defines register-to-register operations for all computational instructions. Source data for these instructions is accessed from the on-chip vector registers (VRs) or are provided as immediate values embedded in the opcode. Architecturally, the VRs are separate from the general-purpose registers (GPRs) and floating-point registers (FPRs). Data is transferred between memory and vector registers with explicit AltiVec load and store instructions only.

Note that the handling of reserved bits in any register is implementation-dependent. Software is permitted to write any value to a reserved bit in a register. However, a subsequent reading of the reserved bit returns 0 if the value last written to the bit was 0 and returns an undefined value (may be 0 or 1) otherwise. This means that even if the last value written to a reserved bit was 1, reading that bit may return 0.

2.1 AltiVec Register File (VRF)

The VRF, shown in Figure 2-1, has 32 registers, each is 128 bits wide. Each vector register can hold sixteen 8-bit elements, eight 16-bit elements, or four 32-bit elements.

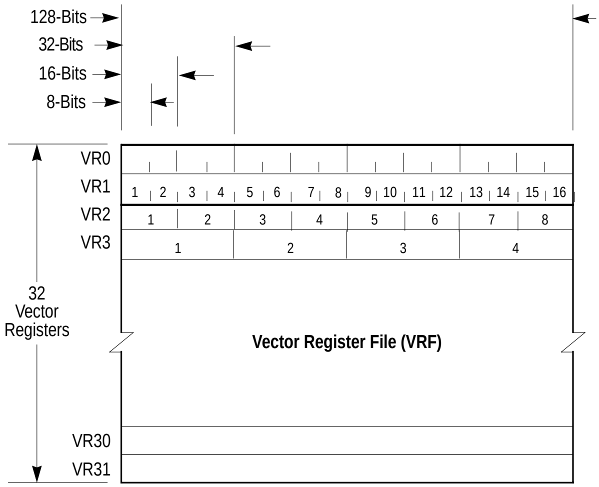


Figure 2-1. AltiVec Register File

The vector registers are accessed as vector instruction operands. Access to registers are explicit as part of the execution of an instruction.

2.1.1 The Vector Status and Control Register (VSCR)

The vector status and control register (VSCR) is a special 32-bit vector register (not an SPR) that is read and written in a manner similar to the FPSCR in the PowerPC scalar floating-point unit. The VSCR is shown in Figure 2-2.

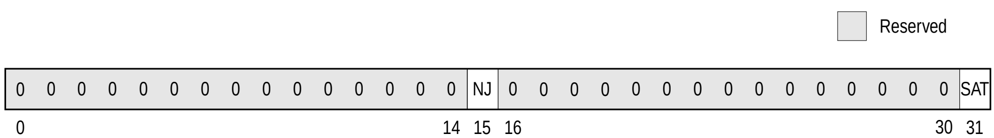


Figure 2-2. Vector Status and Control Register (VSCR)

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.

Special instructions Move from Vector Status and Control Register (**mfvscr)** and Move to Vector Status and Control Register (**mtvscr**) are provided to move the VSCR from and to a vector register. When moved to or from a vector register, the 32-bit VSCR is right-justified in the 128-bit vector register. When moved to a vector register, 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 2-3.

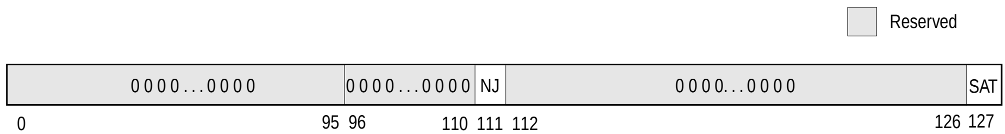


Figure 2-3. VSCR Moved to a Vector Register

VSCR bit settings are shown in Table 2-1.

Table 2-1. VSCR Field Descriptions

Bits	Name	Description
0–14	—	Reserved. The handling of reserved bits is the same as the normal PowerPC implementation, that is, system registers such as XER and FPSCR are 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.
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 floating–point overview Section 3.2.1, “Floating-Point Modes.”
16–30	—	Reserved. The handling of reserved bits is the same as the normal PowerPC implementation, that is, system registers such as XER and FPSCR are 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.
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 to 1 until it is cleared to 0 by an **mtvscr** instruction. For further discussion refer to Section 4.2.1.1, “Saturation Detection.” 1 The AltiVec saturate instruction implicitly sets when saturation has occurred on the results one of the AltiVec instructions having saturate in its name: Move To VSCR (**mtvscr**) Vector Add Integer with Saturation (**vaddubs**, **vadduhs**, **vadduws**, **vaddsbs**, **vaddshs**, **vaddsws**) Vector Subtract Integer with Saturation (**vsububs**, **vsubuh**s, **vsubuws**, **vsubsbs**, **vsubshs**, **vsubsws**) Vector Multiply-Add Integer with Saturation (**vmhaddshs**, **vmhraddshs**) Vector Multiply-Sum with Saturation (**vmsumuhs**, **vmsumshs**, **vsumsws**) Vector Sum-Across with Saturation (**vsumsws**, **vsum2sws**, **vsum4sbs**, **vsum4shs**, **vsum4ubs**) Vector Pack with Saturation (**vpkuhus**, **vpkuwus**, **vpkshus**, **vpkswus**, **vpkshss**, **vpkswss**) Vector Convert to Fixed-Point with Saturation (**vctuxs**, **vctsxs**) 0 Indicates no saturation occurred, **mtvscr** can explicitly clear this bit.

The **mtvscr** is context synchronizing. This implies that all AltiVec instructions logically preceding an **mtvscr** in the program flow will execute in the architectural context (NJ mode) that existed prior to completion of the **mtvscr**, and that all instructions logically following the **mtvscr** will execute in the new context (NJ mode) established by the **mtvscr**.

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

2.1.2 VRSAVE Register (VRSAVE)

The VRSAVE register is a separate register used to assist in application and operating system software in saving and restoring the architectural state across process context-switched events. VRSAVE is a new user-mode accessible 32-bit special-purpose register (SPR 256) that is added to the PowerPC architecture to assist software in providing efficient save and restore operations. The VRSAVE register (VRSAVE) is entirely maintained and managed by software, VRSAVE is shown in Figure 2-4.

VR0	VR1	VR2	VR3	VR4	VR5	VR6	VR7	VR8	VR9	VR10	VR11	VR12	VR13	VR14	VR15	VR16	VR17	VR18	VR19	VR20	VR21	VR22	VR23	VR24	VR25	VR26	VR27	VR28	VR29	VR30	VR31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure 2-4. Saving/Restoring the AltiVec Context Register (VRSAVE)

VRSAVE bit settings are shown in Table 2-2.

Table 2-2. VRSAVE Bit Settings

Bits	Name	Description
0-31	VR*n*	1 VR*n* is live, it is using VR0 of the VRF for the current running process 0 VR*n* is dead, it is not being used for the current running process

The VRSAVE register is read or written only as the direct result of a **mfspr** or **mtspr** instruction, respectively. The recommended usage of VRSAVE is for each bit in this register to correspond to one of the vector registers and its values indicate whether the corresponding register is currently live (1) or dead (0). A live register contains data that is currently in use by the executing process, a dead register does not contain data. If VRSAVE is used to indicate which vector registers (VRs) are being used by a program, the operating system could save only those VRs when an exception occurs, and could restore only those VRs when resuming from the exception. If this approach is taken it must be applied rigorously; if a program fails to indicate that a given VR is in use, software errors may occur that will be difficult to detect and correct because they are timing-dependent. Some operating systems save and restore VRSAVE only for programs that also use other AltiVec registers.

2.1.3 PowerPC Condition Register

The PowerPC condition register (CR) is a 32-bit register that reflects the result of certain operations and provides a mechanism for testing and branching. For AltiVec ISA, the CR6 field can optionally be used, that is if an AltiVec instruction field’s record bit (Rc) is set in a vector compare instruction. The CR6 field is updated. The bits in the PowerPC CR are grouped into eight 4-bit fields, CR0–CR7, as shown in Figure 2-5.

CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
0 3	4 7	8 11	12 15	16 19	20 23	24 27	28 31

Figure 2-5. Condition Register (CR)

For more details on the CR see Chapter 2, “PowerPC Register Set,” in *PowerPC: The Programming Environments Manual.*

To control program flow based on vector data, all vector compare instructions can optionally update CR6. If the instruction field’s record bit (Rc) is set in a vector compare instruction, the CR6 field is updated according to Table 2-3.

Table 2-3. CR6 Field Bit Settings for Vector Compare Instructions

CR Bit	CR6 Field Bit	Vector Compare	Vector Compare Bounds
24	0	1 Relation is true for all element pairs	0
25	1	0	0
26	2	1 Relation is false for all element pairs 0 All fields were in bounds	1 All fields are in bounds for the **vcmpbfp** instruction so the result code of all fields is 0b00 0 One of the fields is out of bounds for the **vcmpbfp** instruction
27	3	0	0

The Rc bit should be used sparingly. As for other PowerPC instructions, in some implementations instructions with Rc bit = 1 could have somewhat longer latency or be more disruptive to instruction pipeline flow than instructions with Rc bit = 0. Therefore techniques of accumulating results and testing infrequently are advised.

2.1.4 AltiVec Bit in the PowerPC Machine State Register (MSR)

An AltiVec Available bit is added to the PowerPC machine state register (MSR). In 64-bit implementations, the MSR is 64 bits wide as shown in Figure 2-6.

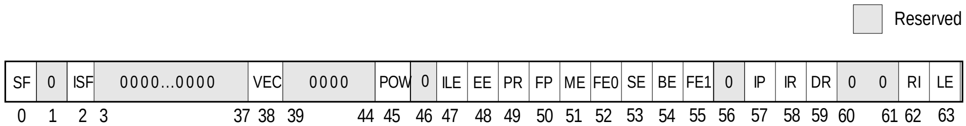


Figure 2-6. Machine State Register (MSR)—64-Bit Implementation

In 32-bit PowerPC implementations, the MSR is 32 bits wide as shown in Figure 2-7. Note that the 32-bit implementation of the MSR is comprised of the 32 least-significant bits of the 64-bit MSR.

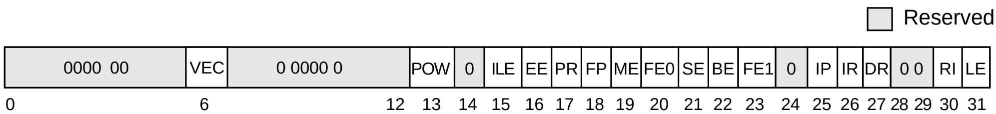


Figure 2-7. Machine State Register (MSR)—32-Bit Implementation

In 32-bit PowerPC implementations, bit 6, VEC, is added to the MSR as shown in Figure 2-7. Also AltiVec data stream prefetching instructions will be suspended and resumed based on MSR[PR] and MSR[DR]. The Data Stream Touch (**dst**) and Data Stream Touch for Store (**dstst**) instructions are supported whenever MSR[DR] = 1. If either instruction is executed when MSR[DR] = 0 (real addressing mode), the results are boundedly undefined. For each existing data stream, prefetching is enabled if the MSR[DR] = 1 and MSR[PR] bit has the value it had when the **dst** or **dstst** instruction that specified the data stream was executed. Otherwise prefetching for the data stream is suspended. In particular, the occurrence of an exception suspends all data stream prefetching.

Table 2-4 shows the AltiVec bit definitions for the MSR as well as how the PR and DR bits are affected by the AltiVec data stream instructions.

Table 2-4. MSR Bit Settings Affected by AltiVec

Bits		Name	Description
64 Bit	32 Bit		
38	6	VEC	AltiVec Available 0 When the bit is cleared to zero, the processor executes an “AltiVec Unavailable Exception” when any attempt to execute a vector instruction that accesses the vector register file (VRF) or VSCR register. 1 The VRF and VSCR registers are accessible to vector instructions. Note: the VRSAVE register is not protected by MSR [VEC]. The data streaming family of instructions (**dst**, **dstt**, **dstst**, **dststt**, **dss**, and **dssall**) are not affected by the MSR[VEC], that is, the VRF and VSCR registers are available to the data streaming instructions even when the MSR[VEC] is cleared.
49	17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions. Note: Care should be taken if data-stream prefetching is used in privileged state (MSR[PR] = 0). For each existing data stream, prefetching is enabled if (a) MSR[DR] = 1 and (b) MSR[PR] has the value it had when the **dst** or **dstst** instruction that specified the data stream was executed. Otherwise, prefetching for the data stream is suspended.
59	27	DR	Data address translation 0 Data address translation is disabled. If data stream touch (**dst**) and data stream touch for store (**dstst**) instructions are executed whenever DR = 0, the results are boundedly undefined. 1 Data address translation is enabled. Data stream touch (**dst**) and data stream touch for store (**dstst**) instructions are supported whenever DR = 1.

For more detailed information including the other bit settings for MSR, refer to Chapter 2, “PowerPC Register Set,” in *PowerPC Microprocessor Family: The Programming Environments Manual*.

2.1.5 Machine Status Save/Restore Registers (SRR)

The machine status save/restore (SRR) registers are part of the PowerPC OEA supervisor-level registers. The SRR0 and SRR1 registers are used to save machine status on exceptions and to restore machine status when an **rfid** (or **rfi)** instruction is executed. For more detailed information, refer to Chapter 2, “PowerPC Register Set,” in *PowerPC: The Programming Environments Manual*.

2.1.5.1 Machine Status Save/Restore Register 0 (SRR0)

The SRR0 is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. SRR0 is used to save machine status on exceptions and restore machine status when an **rfid** (or **rfi**) instruction is executed. For the AltiVec ISA, it holds the effective address (EA) for the instruction that caused the AltiVec unavailable exception. The AltiVec unavailable exception occurs when no higher priority exception exists, and an attempt is made to execute an AltiVec instruction when MSR[VEC] = 0. The format of SRR0 is shown in Figure 2-8.

For 32-bit implementations, the format of SRR0 is that of the low-order bits (32–63) of Figure 2-8.

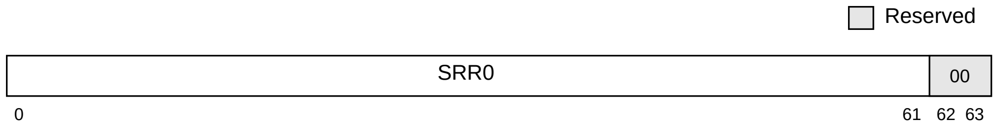


Figure 2-8. Machine Status Save/Restore Register 0 (SRR0)

2.1.5.2 Machine Status Save/Restore Register 1 (SRR1)

The SRR1 is a 64-bit register in 64-bit implementations and a 32-bit register in 32-bit implementations. SRR1 is used to save machine status on exceptions and to restore machine status when an **rfid** (or **rfi**) instruction is executed. The format of SRR1 is shown in Figure 2-9.

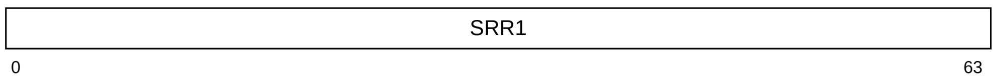


Figure 2-9. Machine Status Save/Restore Register 1 (SRR1)

In 64-bit implementations, when an AltiVec unavailable exception occurs, SRR1[33–36] and SRR1[42–47] are cleared to zero and bits MSR[0], MSR[48–55], MSR[57–59], and MSR[62–63] are placed into the corresponding bit positions of SRR1 as it was just prior to the exception. For 32-bit implementations, when an AltiVec unavailable exception occurs, SRR1[1–4] and SRR[10–15] are cleared and MSR[16–23], MSR[25–27], and MSR[30–31] are placed into the corresponding bit positions of SRR1 as they were before the exception.

2.2 PowerPC Register Set

The addition of the AltiVec technology adds some additional new registers as well as affecting bit settings in some of the PowerPC registers when AltiVec instructions are executed. Figure 2-10 shows a graphic representation of the entire PowerPC register set and how the AltiVec register set resides within the PowerPC architecture. The PowerPC registers affected by AltiVec instructions are shaded.

SUPERVISOR MODEL - OEA

USER MODEL UISA

General-Purpose Registers
- GPR0 (64/32)
- GPR1 (64/32)
- ⋮
- GPR31 (64/32)

Floating-Point Registers
- FPR0 (64)
- FPR1 (64)
- ⋮
- FPR31 (64)

Condition Register [1]
- CR (32)

Floating-Point Status and Control Register [1]
- FPSCR (32)

XER Register [1]
- XER (32) SPR 1

Link Register
- LR (64/32) SPR 8

Count Register
- CTR (64/32) SPR 9

Vector Save/Restore Register
- VRSAVE SPR 256

Vector Status and Control Register
- VSCR (32)

Vector Registers
- VR0 (64)
- VR1 (64)
- VR2 (64)
- ⋮
- VR31 (64)

Configuration Registers

Machine State Register
- MSR (64/32)

Processor Version Register [1]
- PVR (32) SPR 287

Memory Management Registers

Instruction BAT Registers
- IBAT0U (64/32) SPR 528
- IBAT0L (64/32) SPR 529
- IBAT1U (64/32) SPR 530
- IBAT1L (64/32) SPR 531
- IBAT2U (64/32) SPR 532
- IBAT2L (64/32) SPR 533
- IBAT3U (64/32) SPR 534
- IBAT3L (64/32) SPR 535

Data BAT Registers
- DBAT0U (64/32) SPR 536
- DBAT0L (64/32) SPR 537
- DBAT1U (64/32) SPR 538
- DBAT1L (64/32) SPR 539
- DBAT2U (64/32) SPR 540
- DBAT2L (64/32) SPR 541
- DBAT3U (64/32) SPR 542
- DBAT3L (64/32) SPR 543

SDR1
- SDR1 (64/32) SPR 25

Address Space Register [3]
- ASR (64) SPR 280

Segment Registers [1, 2]
- SR0 (32)
- SR1 (32)
- ⋮
- SR15 (32)

Exception Handling Registers

Data Address Register
- DAR (64) SPR 19

DSISR [1]
- DSISR (32) SPR 18

SPRGs
- SPRG0 (64/32) SPR 272
- SPRG1 (64/32) SPR 273
- SPRG2 (64/32) SPR 274
- SPRG3 (64/32) SPR 275

Save and Restore Registers
- SRR0 (64/32) SPR 26
- SRR1 (64/32) SPR 27

Floating-Point Exception Cause Register (Optional)
- FPECR SPR 1022

Miscellaneous Registers

Time Base Facility [1] (For Writing)
- TBL (32) SPR 284
- TBU (32) SPR 285

Decrementer [1]
- DEC (32) SPR 22

Processor Identification Register (Optional)
- PIR SPR 1023

Data Address Breakpoint Register (Optional)
- DABR (64/32) SPR 1013

External Access Register (Optional) [1]
- EAR (32) SPR 282

USER MODEL VEA

Time Base Facility [1] (For Reading)
- TBL (32) TBL 268[4]
- TBU (32) TBU 269

[1] These registers are 32-bit registers only.
[2] These registers are on 32-bit implementations only.
[3] These registers are on 64-bit implementations only.
[4] In 64-bit implementations, TBR268 is read as a 64-bit value.

Figure 2-10. Programming Model—All Registers

Chapter 3
Operand Conventions

This section describes the operand conventions as they are represented in the AltiVec technology at the UISA level. Detailed descriptions are provided of conventions used for transferring data between vector registers and memory, and representing data in these vector registers using both big- and little-endian byte ordering. Additionally, the floating-point default conditions for exceptions are described.

3.1 Data Organization in Memory

U

The AltiVec instruction set architecture (ISA) follows the same data organization as the PowerPC architecture UISA with a few extensions. In addition to supporting byte, half-word and word operands, as defined in the PowerPC architecture UISA, AltiVec ISA supports quad-word (128-bit) operands.

The following sections describe the concepts of alignment and byte ordering of data for quad words, otherwise alignment is the same as described in Chapter 3, “Operand Conventions,” in the *PowerPC Microprocessor Family: The Programming Environments Manual.*

3.1.1 Aligned and Misaligned Accesses

Vectors are accessed from memory with instructions such as Vector Load Indexed (**lvx**) and Store Vector Indexed (**stvx**) instructions. The operand of a vector register to memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. AltiVec instructions are four bytes long and word-aligned like PowerPC instructions.

Operands for vector register to memory access instructions have the characteristics shown in Table 3-1.

Table 3-1. Memory Operand Alignment

Operand	Length	32-bit Aligned Address (28-31)	64-bit Aligned Address (60-63)
Byte	8 bits (1 byte)	xxxx	xxxx
Half word	2 bytes	xxx0	xxx0
Word	4 bytes	xx00	xx00
Quad word	16 bytes	0000	0000

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

The concept of alignment is also applied more generally to data in memory. For example, an 8-byte data item is said to be half-word–aligned if its address is a multiple of two; that is, the effective address (EA) points to the next effective address that is 2 bytes (a half word) past the current effective address, that would be the EA + 2 bytes, and then the next being the EA + 4 bytes, and effective address would continue skipping every 2 bytes (2 bytes = 1 half word). This ensures that the effective address is half-word aligned as it points to each successive half word in memory.

It is important to understand that AltiVec memory operands are assumed to be aligned, and AltiVec memory accesses are performed as if the appropriate number of low-order bits of the specified effective address were zero. This assumption is different from PowerPC integer and floating-point memory access instructions where alignment is not always assumed. So for AltiVec ISA, the low-order bit of the effective address is ignored for half-word AltiVec memory access instructions, and the low-order four bits of the effective address are ignored for quad-word AltiVec memory access instructions. The effect is to load or store the memory operand of the specified length that contains the byte addressed by the effective address.

If a memory operand is misaligned, additional instructions must be used to correctly place the operand in a vector register or in memory. AltiVec technology provides instructions to shift and merge the contents of two vector registers. These instructions facilitate copying misaligned quad-word operands between memory and the vector registers.

3.1.2 AltiVec Byte Ordering

For PowerPC and AltiVec processors, the smallest addressable memory unit is the byte (8 bits), and scalars are composed of one or more sequential bytes. The AltiVec ISA supports both big- and little-endian byte ordering. The default byte ordering is big-endian. However, the code sequence used to switch from big- to little-endian mode may differ among processors.

The PowerPC architecture uses the machine state register (MSR) for specifying byte ordering—little-endian mode (LE). The MSR[LE] specifies the endian mode in which the processor is currently operating. A value of 0 specifies big-endian mode and a value of 1 specifies little-endian mode. For further details on PowerPC byte ordering, refer to Chapter 3, “Operand Conventions,” in the *PowerPC Microprocessor Family: The Programming Environments Manual*

AltiVec ISA follows the endian support of PowerPC for elements up to double words. AltiVec ISA also supports quad words and additional support is provided for this. In AltiVec ISA when a 64-bit scalar is moved from a register to memory, it occupies eight consecutive bytes in memory and a decision must be made regarding byte ordering in these eight addresses.

The default byte ordering for AltiVec ISA is big-endian.

3.1.2.1 Big-Endian Byte Ordering

For big-endian scalars, the most-significant byte (MSB) is stored at the lowest (or starting) address while the least-significant byte (LSB) is stored at the highest (or ending) address. This is called big-endian because the big end of the scalar comes first in memory.

3.1.2.2 Little-Endian Byte Ordering

For little-endian scalars, the LSB is stored at the lowest (or starting) address while the MSB is stored at the highest (or ending) address. This is called little-endian because the little end of the scalar comes first in memory.

3.1.3 Quad Word Byte Ordering Example

The idea of big- and little-endian byte ordering is best illustrated in an example of a quad word such as 0x2021_2223_2425_2627_2829_2A2B_2C2D_2E2F located in memory. This quad word is used throughout this section to demonstrate how the bytes that comprise a quad word are mapped into memory.

The quad word (0x2021_2223_2425_2627_2829_2A2B_2C2D_2E2F) is shown in big-endian mapping in Figure 3-1. A hexadecimal representation is used for showing address values and the values in the contents of each byte. The address is shown below each byte’s contents. The big-endian model addresses the quad word at address 0x00, which is the MSB (0x20), proceeding to the address 0x0F, which contains the LSB (0x2F).

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Quad Word															
Contents	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	↑ MSB															↑ LSB

Figure 3-1. Big-Endian Mapping of a Quad Word

Figure 3-2 shows the same quad word using little-endian mapping. In the little-endian model, the quad word's 0x00 address specifies the LSB (0x2F) and proceeds to address 0x0F which contains its MSB (0x20).

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Quad Word															
Contents	2F	2E	2D	2C	2B	2A	29	28	27	26	25	24	23	22	21	20
Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	↑ LSB															↑ MSB

Figure 3-2. Little-Endian Mapping of a Quad Word

Figure 3-3 shows the sequence of bytes laid out with addresses increasing from left to right. Programmers familiar with little-endian byte ordering may be more accustomed to viewing quad words laid out with addresses increasing from right to left, as shown in Figure 3-3.

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Quad Word															
Contents	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
Address	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
	↑ MSB															↑ LSB

Figure 3-3. Little-Endian Mapping of Quad Word—Alternate View

This allows the little-endian programmer to view each scalar in its natural byte order of MSB to LSB. This section uses both conventions based of ease of understanding for the specific example.

3.1.4 Aligned Scalars in Little-Endian Mode

The effective address (EA) calculation for the load and store instructions is described in Chapter 4, "Addressing Modes and Instruction Set Summary." For PowerPC processors in little-endian mode, the effective address is modified before being used to access memory. In PowerPC, the three low-order address bits of the effective address are exclusive-ORed (XOR) with a three-bit value that depends on the length of the operand (1, 2, 4, or 8 bytes), as shown in Table 3-2. This address modification is called munging.

Table 3-2 Effective Address Modifications

Data Width (Bytes)	EA Modification
1	XOR with 0b111
2	XOR with 0b110
4	XOR with 0b100
8	No change

The munged physical address is passed to the cache or to main memory, and the specified width of the data is transferred (in big-endian order—that is, MSB at the lowest address, LSB at the highest address) between a GPR or FPR and the addressed memory locations (as modified).

Munging makes it appear to the processor that individual aligned scalars are stored as little-endian, when in fact they are stored in big-endian order but at different byte addresses within double words. Only the address is modified, not the byte order. For further details on how to align scalars in little-endian mode see Chapter 3, “Operand Conventions,” in *PowerPC: The Programming Environments Manual*.

The PowerPC address munging is performed on double-word units. In the PowerPC architecture, little-endian mode would have the double words of a quad word appear swapped. When the quad word in memory shown at the top of Figure 3-4, loads from address 0x00, the bottom of Figure 3-4 shows how it appears to the processor as it munges the address.

Contents	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	Memory Image
Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Quad Word															
Contents	27	26	25	24	23	22	21	20	2F	2E	2D	2C	2B	2A	29	28
Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F

Figure 3-4. Quad Word Load with PowerPC Munged Little-Endian Applied

Note that double words are swapped. The byte element addressed by the quad word’s base address, 0x0F, contains 0x28, while its MSB at address 0x0 contains 0x27. This is due to the PowerPC munging being applied to offsets within double words; AltiVec ISA requires a munge within quad words.

To accommodate the quad-word operands, the PowerPC architecture can not simply be extended by munging an extra address bit. It would break existing code and/or platforms. Processors that implement AltiVec technology could not be mixed with non-AltiVec processors. Instead, AltiVec processors implement a double-word swap when moving quad words between vector registers and memory.

Figure 3-5 shows how this swapping could be implemented. This diagram represents the load path double-word swapping; the store path looks the same, except that the memory and internal boxes are reversed.

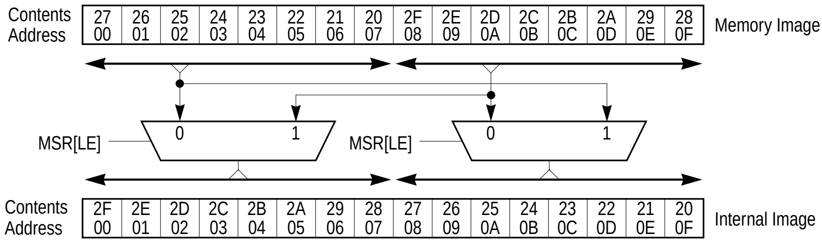


Figure 3-5. AltiVec Little Endian Double-Word Swap

In the diagram, the numbers at the bottom of the byte boxes represent the offset address of that byte; the numbers at the top are the values of the bytes at that offset.The little-endian ordering is discontinuous because the PowerPC munging is performed only on double-word units. The purpose of the double word swap within the AltiVec unit is to perform an additional swap that is not part of the PowerPC architecture.

When MSR[LE] = 1, double words are swapped and the bytes now appear in their expected ordering. When MSR[LE] = 0, no swapping is done.

To summarize, in little-endian mode, the load vector element indexed instructions (**lvebx**, **lvehx**, **lvewx**) and the store vector element indexed instructions (**stvebx**, **stvehx**, **stvewx**) have the same 3-bit address munge applied to the memory address as is specified by the PowerPC architecture for integer and floating-point loads and stores. For the quad word load vector indexed instructions (**lvx**, **lvxl**) and the store vector indexed instructions (**stvx**, **stvxl**) the two double words of the quad-word scalar data are munged and swapped as they are moved between the vector register and memory.

3.1.5 Vector Register and Memory Access Alignment

When loading an aligned byte, half word, or word memory operand into a vector register, the element that receives the data is the element that would have received the data had the entire aligned quad word containing the memory operand addressed by the effective address been loaded. Similarly, when an element in a vector register is stored into an aligned memory operand, the element selected to be stored is the element that would have been stored into the memory operand addressed by the effective address had the entire vector register been stored to the aligned quad word containing the memory operand addressed by the effective address. The position of the element in the target or source vector register depends on the endian mode, as described above. (Byte memory operands are always aligned.)

For aligned byte, half word, and word memory operands, if the corresponding element number is known when the program is written, the appropriate vector splat and vector permute instructions can be used to copy or replicate the data contained in the memory operand after loading the operand into a vector register. A vector splat instructions will take the contents of an element in a vector register and replicates that into each element in the destination vector register. A vector permute instruction is the concatenation of the contents of two vectors. An example of this is given in detail in Section 3.1.6, “Quad-Word Data Alignment.” Another example is to replicate the element across an entire vector register before storing it into an arbitrary aligned memory operand of the same length; the replication ensures that the correct data is stored regardless of the offset of the memory operand in its aligned quad word in memory.

Since vector loads and stores are size-aligned, application binary interfaces (ABIs) should specify, and programmers should take care to align data on quad-word boundaries for maximum performance.

3.1.6 Quad-Word Data Alignment

The AltiVec ISA does not provide for alignment exceptions for loading and storing data. When performing vector loads and stores, the effect is as if the low-order four bits of the address are 0x0, regardless of the actual effective address generated. Since vectors may often be misaligned due to the nature of the algorithm, AltiVec ISA provides support for post-alignment of quad-word loads and pre-alignment for quad-word stores. Note that in the following diagrams, the effect of the swapping described above is assumed and the memory diagrams will be with respect to the logical mapping of the data.

Figure 3-6 and Figure 3-7 show misaligned vectors in memory for both big- and little-endian ordering. The big-endian and little-endian examples assumes that the desired vector begins at address 0x03.

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	Quad Word HI																Quad Word LO															
Contents				20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F													
Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F

↑ MSB ↑ LSB

Figure 3-6. Misaligned Vector in Big-Endian Mode

Byte	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Quad Word HI																Quad Word LO															
Contents														2F	2E	2D	2C	2B	2A	29	28	27	26	25	24	23	22	21	20			
Address	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
	↑ MSB																															↑ LSB

Figure 3-7. Misaligned Vector in Little-Endian Addressing Mode

Figure 3-6 and Figure 3-7 show how such misaligned data causes data to be split across aligned quad words; only aligned quad words are loaded and/or stored by AltiVec load/store instructions. To align this vector, a program must load both (aligned) quad words that contain a portion of the misaligned vector data and then execute a Vector Permute (vperm) instruction to align the result.

3.1.6.1 Accessing a Misaligned Quad Word in Big-Endian Mode

Figure 3-1 shows the big-endian alignment model, using the example in Figure 3-8, **v**HI and **v**LO (HI = high order quad word, LO = low order quad word) represent vector registers that contain the misaligned quad words containing the MSBs and LSBs, respectively, of the misaligned quad word; **v**D is the target vector register.

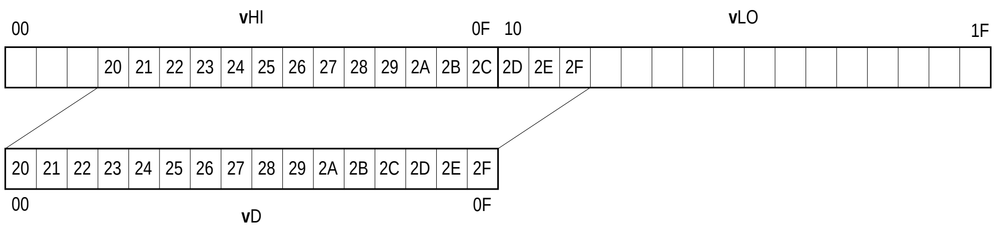


Figure 3-8. Big-Endian Quad Word Alignment

Alignment is performed by left-rotating the combined 32-byte quantity (**v**HI:**v**LO) by an amount determined by the address of the first byte of the desired data. This left-rotation is done by means of a **vperm** instruction whose control vector is generated by a Load Vector for Shift Left (**lvsl**) instruction after loading the most-significant quad word (MSQ) and least-significant quad word (LSQ) that contain the desired vector. The **lvsl** instruction uses the same address specification as the load vector indexed that loads the **v**HI component, which for big-endian ordering is the address of the desired vector.

The following instruction sequence extracts the quad word in big-endian mode:

```
lvx     vHI,rA,rB       ;# load the MSQ
lvsl    vP,rA,rB        ;# set the permute vector
addi    rB,rB,16        ;# address of LSQ
lvx     vLO,rA,rB       ;# load LSQ component
vperm   vD,vHI,vLO,vP   ;# align the data
```

Note that when streaming data is used, the overhead of generating the alignment permute vector can be spread out and the latency of the loads may be covered by loop unrolling.

The process of storing a misaligned vector is essentially the reverse of that for loading; except that the code has a read-modify-write sequence. The logical algorithm is that the vector source must be right-shifted and split into two parts, each of which is merged (via a Vector Select (**vsel**) instruction) with the current contents of its MSQ and its LSQ and stored back using a Store Vector Indexed (**svx**) instruction.

The Load Vector for Shift Right (**lvsr**) instruction is used to produce the permute control vector to be used for the right-shifting. An observation is that a single register can be used for the shifted contents if a right-rotate is done. The rotate is affected by specifying the source register for both components of the Vector Permute (**vperm**); that is, a shift of a double register with the same contents in both parts results in a rotate. In addition, the same permute control vector can be used on a sequence of ones and zeros to generate a mask for use by the **vsel** instruction to do the merging.

The complete code sequence for the store case is as follows:

```
lvx     vHI,rA,rB           ;# load current MSQ for update
lvsr    vP,rA,rB            ;# load the alignment vector
addi    rB,rB,16            ;# address of LSQ
lvx     vLO,rA,rB           ;# load the current LSQ’s data
vspltib v1s,-1              ;# generate the select mask bits
vspltib v0s,0
vperm   vMask,v0s,v1s,vP    ;# right rotate the select mask
vperm   vSrc,vSrc,vSrc,vP   ;# right rotate the data
vsel    vLO,vSrc,vLO,vMask  ;# insert LSQ component
vsel    vHI,vHI,vSrc,vMask  ;# insert MSQ component
stvx    vLO,rA,rB           ;# store LSQ
addi    rB,rB,-16           ;# address of MSQ
stvx    vHI,rA,rB           ;# store MSQ
```

When fetching a linear stream of misaligned quad words, the control vector need only be computed once. Thus the time required for aligned fetches on the ends of the stream is proportioned out. None of the data fetched internally to the stream is wasted and only gets fetched once. The average time expended for a misaligned **lvx** instruction in a long sequence approaches one **lvx** and one **vperm** instruction**.**

3.1.6.2 Accessing a Misaligned Quad Word in Little-Endian Mode

The instruction sequences used to access misaligned quad-word operands in little-endian mode are similar to those used in big-endian mode. The following instruction sequence can be used to load the misaligned quad word shown in Figure 3-7 into a vector register in little-endian mode. The load alignment case is shown in Figure 3-9. The vector register **v**HI and **v**LO receive the MSQ and LSQ respectively; **v**D is the target vector register. The **lvsr** instruction uses the same address specification as the **lvx** instruction that loads **v**LO; in little-endian byte ordering this is the address of the desired misaligned quad word.

```
lvx     vLO,rA,rB       # load the LSQ
lvsr    vP,rA,rB        # set the permute vector
addi    rB,rB,16        # address of MSQ
lvx     vHI,rA,rB       # load MSQ component
vperm   vD,vHI,vLO,vP   # align the data
```

Similarly, the following sequence of instructions stores the contents of register **v**D into a misaligned quad word in memory in little-endian mode.

```
lvx         vLO,rA,rB           # load current LSQ for update
lvsl        vP,rA,rB            # load the alignment vector
addi        rB,rB,16            # address of MSQ
lvx         vHI,rA,rB           # load the current MSQ’s data
vspltib     v1s,-1              # generate the select mask bits
vspltib     v0s,0
vperm       vMask,v0s,v1s,vP    # left rotate the select mask
vperm       vSrc,vSrc,vSrc,vP   # left rotate the data
vsel        vHI,vHI,vSrc,vMask  # insert MSQ component
vsel        vLO,vSrc,vLO,vMask  # insert LSQ component
stvx        vHI,rA,rB           # store MSQ
addi        rB,rB,-16           # address of LSQ
stvx        vLO,rA,rB           # store LSQ
```

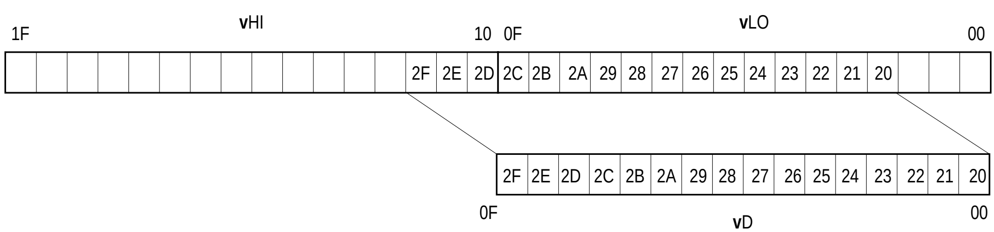


Figure 3-9. Little-Endian Alignment

3.1.6.3 Scalar Loads and Stores

No alignment is performed for scalar load or store instructions in the AltiVec ISA. If a vector load or store address is not properly size aligned, the suitable number of least significant bits are ignored, and a size aligned transfer occurs instead. Data alignment must be performed explicitly after being brought into the registers. No assistance is provided for aligning individual scalar elements that are not aligned on their natural boundary. The placement of scalar data in a vector element depends upon its address. That is, the placement of the addressed scalar is the same as if a load vector indexed instruction has been performed, except that only the addressed scalar is accessed (for cache-inhibited space); the values in the other vector elements are boundedly undefined. Also, data in the specified scalar is the same as if a store vector indexed instruction had been performed, except that only the scalar addressed is affected. No instructions are provided to assist in aligning individual scalar elements that are not aligned on their natural size boundary.

When a program knows the location of a scalar, it can perform the correct vector splats and vector permutes to move data to where it is required. For example, if a scalar is to be used as a source for a vector multiply (that is, each element multiplied by the same value), the scalar must be splatted into a vector register. Likewise, a scalar stored to an arbitrary memory location must be splatted into a vector register, and that register must be specified as the source of the store. This guarantees that the data appears in all possible positions of that scalar size for the store.

3.1.6.4 Misaligned Scalar Loads and Stores

Although no direct support of misaligned scalars is provided, the load-aligning sequence for big-endian vectors described in Section 3.1.6.1, “Accessing a Misaligned Quad Word in Big-Endian Mode” can be used to position the scalar to the left vector element, which can then be used as the source for a splat. That is, the address of a scalar is also the address of the left-most element of the quad word at that address. Similarly, the read-modify-write sequences, with the mask adjusted for the scalar size, can be used to store misaligned scalars. The same is true for little-endian mode, the load-aligning sequence for little-endian vectors described Section 3.1.6.2, “Accessing a Misaligned Quad Word in Little-Endian Mode” can be used to position the scalar to the right vector element, which can then be used as the source for a splat. That is, the address of a scalar is also the address of the right-most element of the quad word at that address.

Note that while these sequences work in cache-inhibited space, the physical accesses are not guaranteed to be atomic.

3.1.7 Mixed-Endian Systems

In many systems, the memory model is not as simple as the examples in this chapter. In particular, big-endian systems with subordinate little-endian buses (such as PCI) comprise a mixed-endian environment.

The basic mechanism to handle this is to use the Vector Permute (**vperm**) instruction to swap bytes within data elements. The value of the permute control vector depends on the size of the elements (8, 16, 32). That is, the permute control vector performs a parallel equivalent of the PowerPC Load Word Byte-Reverse Indexed (**lwbrx)** instruction, within the vector registers.

The ultimate problem is when there are misaligned, mixed-endian vectors. This can be handled by applying a vector permute of the data as required for the misaligned case, followed by the swapping vector permute on that result. Note that for streaming cases, the effect of this double permute can be accomplished by computing the swapping permute of the alignment permute vector, and then applying the resulting permute control vector to incoming data.

3.2 AltiVec Floating-Point Instructions—UISA

U There are two kinds of floating-point instructions defined for the PowerPC and AltiVec ISA—computational and noncomputational. Computational instructions consist of those operations defined by the IEEE-754 standard for 32-bit arithmetic (those that perform addition, subtraction, multiplication, and division) and the multiply-add defined by the architecture. Noncomputational floating-point instructions consist of the floating-point load and store instructions. Only the computational instructions are considered floating-point operations throughout this chapter.

The single-precision format, value representations, and computational model defined in Chapter 3, “Operand Conventions,” in *PowerPC Microprocessor Family: The Programming Environments Manual* apply to AltiVec floating-point except as follows:

- In general, no status bits are set to reflect the results of floating-point operations. The only exception is that VSCR[SAT] may be set by the Vector Convert to Fixed-Point Word instructions.
- With the exception of the two Vector Convert to Fixed-Point Word (**vctuxs**, **vctsxs**) instructions and three of the four Vector Round to Floating-Point Integer (**vrfiz**, **vrfip**, **vrfim**) instructions, all AltiVec floating-point instructions that round use the round-to-nearest rounding mode.
- Floating-point exceptions cannot cause the system error handler to be invoked.

If a function is required that is specified by the IEEE standard, is not supported by AltiVec ISA, and cannot be emulated satisfactorily using the functions that are supported by AltiVec ISA, the functions provided by the floating-point processor should be used; see Chapter 4, “Addressing Modes and Instruction Set Summary,” in *PowerPC: The Programming Environments Manual*

3.2.1 Floating-Point Modes

AltiVec ISA supports two floating-point modes of operation—a Java mode and a non-Java mode of operation that is useful in circumstances where real-time performance is more important than strict Java and IEEE-standard compliance.

When VSCR[NJ] is 0 (default), operations are performed in Java mode. When VSCR[NJ] is 1, operations are carried out in the non-Java mode.

3.2.1.1 Java Mode

Java compliance requires compliance with only a subset of the Java/IEEE/C9X standard. The Java subset helps simplify floating-point implementations, as follows:

- Reducing the number of operations that must be supported
- Eliminating exception status flags and traps
- Producing results corresponding to all disabled exceptions thus eliminating enabling control flags
- Requiring only round-to-nearest rounding mode eliminates directed rounding modes and the associated rounding control flags.

Java compliance requires the following aspects of the IEEE standard:

- Supporting denorms as inputs and results (gradual underflow) for arithmetic operations
- Providing NaN results for invalid operations
- NaNs compare unordered with respect to everything, so that the result of any comparison of any NaN to any data type is always false

In some implementations, floating-point operations in Java mode may have somewhat longer latency on normal operands and possibly much longer latency on denormalized operands than operations in non-Java mode. This means that in Java mode overall real-time response may be somewhat worse and deadline scheduling may be subject to much larger variance than non-Java mode.

3.2.1.2 Non-Java Mode

In the non-Java/non-IEEE/non-C9X mode (VSCR[NJ] = 1), gradual underflow is not performed. Instead, any instruction that would have produced a denormalized result in Java mode substitutes a correctly signed zero (±0.0) as the final result. Also, denormalized input operands are flushed to the correctly signed zero (±0.0) before being used by the instruction.

The intent of this mode is to give programmers a way to assure optimum, data-insensitive, real-time response across implementations. Another way to improved response time would be to implement denormalized operations through software emulation.

It is architecturally permitted, but strongly discouraged, for an implementation to implement only non-Java mode. In such an implementation, the VSCR[NJ] does not respond to attempts to clear it and is always read back as a 1.

No other architecturally-visible, implementation-specific deviations from this specification are permitted in either mode.

3.2.2 Floating-Point Infinities

Valid operations on infinities are processed according to the IEEE standard.

3.2.3 Floating-Point Rounding

All AltiVec floating-point arithmetic instructions use the IEEE default rounding mode, round-to-nearest. The IEEE directed rounding modes are not provided.

3.2.4 Floating-Point Exceptions

The following floating-point exceptions may occur during execution of AltiVec floating-point instructions.

- NaN operand exception
- Invalid operation exception
- Zero divide exception
- Log of zero exception
- Overflow exception
- Underflow exception

If an exception occurs, a result is placed into the corresponding target element as described in the following subsections. This result is the default result specified by Java, the IEEE standard, or C9X, as applicable. Recall that denormalized source values are treated as if they were zero when VSCR[NJ] =1. The consequences regarding exceptions are as follows:

- Exceptions that can be caused by a zero source value can be caused by a denormalized source value when VSCR[NJ] = 1.
- Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when VSCR[NJ] = 1.

3.2.4.1 NaN Operand Exception

If the exponent of a floating-point number is 255 and the fraction is non-zero, then the value is a NaN. If the most significant bit of the fraction field of a NaN is zero, then the value is a signaling NaN (SNaN), otherwise it is a quiet NaN (QNaN). In all cases the sign of a NaN is irrelevant.

A NaN operand exception occurs when a source value for any of the following instructions is a NaN.

- An AltiVec instruction that would normally produce floating-point results
- Either of the two, Vector Convert to Unsigned Fixed-Point Word Saturate (**vctuxs**) or Vector Convert to Signed Fixed-Point Word Saturate (**vctsxs**) instructions
- Any of the four vector floating-point compare instructions

The following actions are taken:

1. If the AltiVec instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is an SNaN it is converted to the corresponding QNaN (by setting the high-order bit of the fraction field to 1 before being placed into the target element).

```
if the element in register vA is a NaN
        then the result is that NaN
else if the element in register vB is a NaN
        then the result is that NaN
        else if the element in register vC is a NaN
                then the result is that NaN
```

2. If the instruction is either of the two vector convert to fixed-point word instructions (**vctuxs**, **vctsxs**), the corresponding result is 0x0000_0000. VSCR[SAT] is not affected.
3. If the instruction is Vector Compare Bounds Floating-Point (**vcmpbfp**[**.**]), the corresponding result is 0xC000_0000.
4. If the instruction is one of the other three vector floating-point compare instructions (**vcmpeqfp**[**.**], **vcmpfgefp**[**.**], **vcmpbfp**[**.**]), the corresponding result is 0x0000_0000.

3.2.4.2 Invalid Operation Exception

An invalid operation exception occurs when a source value is invalid for the specified operation. The invalid operations are as follows:

- Magnitude subtraction of infinities
- Multiplication of infinity by zero
- Vector Reciprocal Square Root Estimate Float (**vrsqrtefp**) of a negative, nonzero number or -X
- Log base 2 estimate (**vlogefp**) of a negative, nonzero number or -X

The corresponding result is the QNaN 0x7FC0_0000. This is the single-precision format analogy of the double precision format generated QNaN described in Chapter 3, “Operand Conventions,” in *PowerPC: The Programming Environments Manual.*

3.2.4.3 Zero Divide Exception

A zero divide exception occurs when a Vector Reciprocal Estimate Floating-Point (**vrefp**) or Vector Reciprocal Square Root Estimate Floating-Point (**vrsqrtefp**) instruction is executed with a source value of zero.

The corresponding result is infinity, where the sign is the sign of the source value, as follows:

- $1/+0.0 \rightarrow +\infty$
- $1/-0.0 \rightarrow -\infty$
- $1/(\sqrt{+0.0}) \rightarrow +\infty$
- $1/(\sqrt{-0.0}) \rightarrow -\infty$

3.2.4.4 Log of Zero Exception

A log of zero exception occurs when a Vector Log Base 2 Estimate Floating-Point instruction (**vlogefp**) is executed with a source value of zero. The corresponding result is infinity. The exception cases are as follows:

- **vlogefp** $\log_2(\pm 0.0) \rightarrow -\infty$
- **vlogefp** $\log_2(-x) \rightarrow$ QNaN, where $x \neq 0$

3.2.4.5 Overflow Exception

An overflow exception happens when either of the following conditions occur:

- For an AltiVec instruction that would normally produce floating-point results, the magnitude of what would have been the result if the exponent range were unbounded exceeds that of the largest finite single-precision number.
- For either of the two Vector Convert To Fixed-Point Word instructions (**vctuxs**, **vctsxs**), either a source value is an infinity or the product of a source value and 2 unsigned immediate value (UIMM) is a number too large to be represented in the target integer format.

The following actions are taken:

1. If the AltiVec instruction would normally produce floating-point results, the corresponding result is infinity, where the sign is the sign of the intermediate result.
2. If the instruction is Vector Convert to Unsigned Fixed-Point Word Saturate (**vctuxs**), the corresponding result is 0xFFFF_FFFF if the source value is a positive number or +X, and is 0x0000_0000 if the source value is a negative number or -X. VSCR[SAT] is set.
3. If the instruction is Vector Convert to Signed Fixed-Point Word Saturate (**vcfsx**), the corresponding result is 0x7FFF_FFFF if the source value is a positive number or +X, and is 0x8000_0000 if the source value is a negative number or -X. VSCR[SAT] is set.

3.2.4.6 Underflow Exception

Underflow exceptions occur only for AltiVec instructions that would normally produce floating-point results. It is detected before rounding. It occurs when a nonzero intermediate result, computed as though both the precision and the exponent range were unbounded, is less in magnitude than the smallest normalized single-precision number (2^{-126}).

The following actions are taken:

1. If VSCR[NJ] = 0, the corresponding result is the value produced by denormalizing and rounding the intermediate result.
2. If VSCR[NJ] = 1, the corresponding result is a zero, where the sign is the sign of the intermediate result.

3.2.5 Floating-Point NaNs

The AltiVec floating-point data format is compliant with the Java/IEEE/C9X single-precision format. A quantity in this format can represent a signed normalized number, a signed denormalized number, a signed zero, a signed infinity, a quiet not a number (QNaN), or a signaling NaN (SNaN).

3.2.5.1 NaN Precedence

Whenever only one source operand of an instruction that returns a floating-point result is a NaN, then that NaN is selected as the input NaN to the instruction. When more than one source operand is a NaN, the precedence order for selecting the NaN is first from **v**A then from **v**B and then from **v**C. If the selected NaN is an SNaN, it is processed as described in Section 3.2.5.2, “SNaN Arithmetic.” If the selected NaN is a QNaN, it is processed according to Section 3.2.5.3, “QNaN Arithmetic.”

3.2.5.2 SNaN Arithmetic

Whenever the input NaN to an instruction is an SNaN, a QNaN is delivered as the result, as specified by the IEEE standard when no trap occurs. The delivered QNaN is an exact copy of the original SNaN except that it is quieted; that is, the most-significant bit (msb) of the fraction is set to one (1).

3.2.5.3 QNaN Arithmetic

Whenever the input NaN to an instruction is a QNaN, it is propagated as the result according to the IEEE standard. All information in the QNaN is preserved through all arithmetic operations.

3.2.5.4 NaN Conversion to Integer

All NaNs convert to zero on conversions to integer instructions such as **vctuxs** and **vctsxs**.

3.2.5.5 NaN Production

Whenever the result of an AltiVec operation originates a NaN (for example, an invalid operation), the NaN produced is a QNaN with the sign bit = 0, exponent field = 255, msb of the fraction field = 1, and all other bits = 0.

Chapter 4
Addressing Modes and Instruction Set Summary

This chapter describes instructions and addressing modes defined by the AltiVec Instruction Set Architecture (ISA) and according to the three levels of the PowerPC architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). AltiVec instructions are primarily UISA, and if otherwise they are noted in the chapter. These instructions are divided into the following categories:

U V O

- Vector integer arithmetic instructions—These include arithmetic, logical, compare, rotate and shift instructions, described in Section 4.2.1, “Vector Integer Instructions.”
- Vector floating-point arithmetic instructions—These include floating-point arithmetic instructions, as well as a discussion on floating-point modes, described in Section 4.2.2, “Vector Floating-Point Instructions.”
- Vector load and store instructions—These include load and store instructions for vector registers, described in Section 4.2.3, “Load and Store Instructions.”
- Vector permutation and formatting instructions—These include pack, unpack, merge, splat, permute, select and shift instructions, described in Section 4.2.5, “Vector Permutation and Formatting Instructions.”
- Processor control instructions—These instructions are used to read and write from the AltiVec Status and Control Register, described in Section 4.2.6, “Processor Control Instructions—UISA.”
- Memory control instructions—These instructions are used for managing of caches (user level and supervisor level), described in Section 4.3.1, “Memory Control Instructions—VEA.”

This grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions within a processor implementation.

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision operands. The AltiVec ISA uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and word operand fetches and stores between memory and the vector registers (VRs).

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

4.1 Conventions

This section describes conventions used for the AltiVec instruction set. Descriptions of memory addressing, synchronization, and the AltiVec exception summary follow.

4.1.1 Execution Model

When used with the PowerPC instructions, AltiVec instructions can be viewed by the programmer as simply new PowerPC instructions that are freely intermixed with existing ones to provide additional features in the instruction set. PowerPC processors appear to execute instructions in program order. Some AltiVec implementations may not allow out-of-order execution and completion. Non-data dependent vector instructions may issue and execute while longer latency previously issued instructions are still in the execution stage. Register renaming is useful for AltiVec instructions to avoid stalling dispatch on false dependencies and allow maximum register name reuse in heavily unrolled loops. The execution of a sequence of instructions will not be interrupted by exceptions as the unit does not report IEEE exceptions but rather produces the default results as specified in the Java/IEEE/C9X standards. The execution of a sequence of instructions may only be interrupted by a vector load or store instruction, otherwise AltiVec instructions do not generate any exceptions.

4.1.2 Computation Modes

The AltiVec ISA supports the following PowerPC architecture types of implementations:

- 64-bit implementations, in that all general-purpose and floating-point registers, and some special-purpose registers (SPRs) are 64 bits long and effective addresses are 64 bits long. All 64-bit implementations have two modes of operation: the default 64-bit mode and 32-bit mode. The mode controls how an effective address is interpreted, how condition bits are set, and how the count register (CTR) is tested by branch conditional instructions.

  O The machine state register bit 0, MSR[SF], is used to choose between 64- and 32-bit modes. When MSR[SF] = 0, the processor runs in 32-bit mode, and when MSR[SF] = 1 the processor runs in the default 64-bit mode.

- U 32-bit implementations, in that all registers except FPRs are 32 bits long and effective addresses are 32 bits long.

Instructions defined in this chapter are provided in both 64-bit implementations and 32-bit implementations unless otherwise stated.

4.1.3 Classes of Instructions

AltiVec instructions follows the illegal instruction class defined by the PowerPC architecture in the section “Classes of Instructions” in Chapter 4, “Addressing Modes and Instruction Set Summary,” of the *PowerPC Microprocessor Family: The Programming Environments Manual*. For AltiVec ISA, all unspecified encodings within the major opcode (04) that are not defined are illegal PowerPC instructions. The only exclusion in defining an unspecified encoding is an unused bit in an immediate field or specifier field (///).

4.1.4 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, or cache instruction, and when it fetches the next sequential instruction.

4.1.4.1 Memory Operands

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

Memory operands may be bytes, half words, words, or quad words for AltiVec instructions. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The AltiVec ISA supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian; see Section 3.1.2, “AltiVec Byte Ordering,” for more information.

The natural alignment boundary of an operand of a single-register memory access instruction is equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. For a detailed discussion about memory operands, see Section 3.1, “Data Organization in Memory.”

4.1.4.2 Effective Address Calculation

An effective address (EA) is the 64- or 32-bit sum computed by the processor when executing a memory access or when fetching the next sequential instruction. For a memory access instruction, if the sum of the EA and the operand length exceeds the maximum EA, the memory operand is considered to wrap around from the maximum EA through EA 0, as described in the Chapter 4, “Addressing Modes and Instruction Set Summary,” in *PowerPC Microprocessor Family: The Programming Environments Manual*.

A zero in the **r**A field indicates the absence of the corresponding address component. For the absent component, a value of zero is used for the address. This is shown in the instruction description as (**r**A|0).

In all implementations (including 32-bit mode in 64-bit implementations), the processor can modify the three low-order bits of the calculated effective address before accessing memory if the PowerPC system is operating in little-endian mode. The double words of a quad word may be swapped as well. See Section 3.1.2, “AltiVec Byte Ordering,” for more information about little-endian mode.

AltiVec load and store operations use register indirect with index mode and boundary align to generate effective addresses. For further details see Section 4.2.3.2, “Load and Store Address Generation.”

4.2 AltiVec UISA Instructions

AltiVec instructions can provide additional supporting instructions to the PowerPC architecture. This section discusses the instructions defined in the AltiVec user instruction set architecture (UISA).

4.2.1 Vector Integer Instructions

The following are categories for vector integer instructions:

- Vector integer arithmetic instructions
- Vector integer compare instructions
- Vector integer logical instructions
- Vector integer rotate and shift instructions

Integer instructions use the content of the vector registers (VRs) as source operands and place results into VRs as well. Setting the Rc bit of a vector compare instruction causes the PowerPC condition register (CR) to be updated.

The AltiVec integer instructions treat source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation. For example, Vector Add Unsigned Word Modulo (**vadduwm**) and Vector Multiply Odd Unsigned Byte (**vmuloub**) instructions interpret both operands as unsigned integers.

4.2.1.1 Saturation Detection

Most integer instructions have both signed and unsigned versions and many have both modulo (wrap-around) and saturating clamping modes. Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero on underflow and to the maximum positive integer value (2^n-1, for example, 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number (-2^{n-1}, for example, -128 for byte fields) on underflow, and to the largest representable positive number (2^{n-1}-1, for example, +127 for byte fields) on overflow. When a modulo instruction is used, the resultant number truncates overflow or underflow for the length (byte, half word, word, quad word) and type of operand (unsigned, signed). The AltiVec ISA provides a way to detect saturation and sets the SAT bit in the Vector Status and Control Register (VSCR[SAT]) in a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned 0+0 → 0 and unsigned byte 1+254 → 255, are not considered saturation conditions and do not cause VSCR[SAT] to be set.

The VSCR[SAT] can be set by the following types of integer, floating-point, and formatting instructions:

- Move to VSCR (**mtvscr**)
- Vector add integer with saturation (**vaddubs**, **vadduhs**, **vadduws**, **vaddsbs**, **vaddshs**, **vaddsws**)
- Vector subtract integer with saturation (**vsububs**, **vsubuh**s, **vsubuws**, **vsubsbs**, **vsubshs**, **vsubsws**)
- Vector multiply-add integer with saturation (**vmhaddshs**, **vmhraddshs**)
- Vector multiply-sum with saturation (**vmsumuhs**, **vmsumshs**, **vsumsws**)
- Vector sum-across with saturation (**vsumsws**, **vsum2sws**, **vsum4sbs**, **vsum4shs**, **vsum4ubs**)
- Vector pack with saturation (**vpkuhus**, **vpkuwus**, **vpkshus**, **vpkswus**, **vpkshss**, **vpkswss**)
- Vector convert to fixed-point with saturation (**vctuxs**, **vctsxs**)

Note that only instructions that explicitly call for saturation can set VSCR[SAT]. Modulo integer instructions and floating-point arithmetic instructions never set VSCR[SAT]. For further details see Section 2.1.1, "The Vector Status and Control Register (VSCR)."

4.2.1.2 Vector Integer Arithmetic Instructions

Table 4-1 lists the integer arithmetic instructions for the PowerPC processors.

Table 4-1. Vector Integer Arithmetic Instructions

Name	Mnemonic	Syntax	Operation
Vector Add Unsigned Integer [b,h,w] Modulo	**vaddubm** **vadduhm** **vadduwm**	**v**D,**v**A,**v**B	Place the sum (**v**A[unsigned integer elements]) + (**v**B[unsigned integer elements]) into **v**D[unsigned integer elements] using modulo arithmetic. For **b**, byte, integer length = 8 bits =1 byte, add 16 unsigned integers from **v**A to the corresponding 16 unsigned integers from **v**B For **h**, half word, integer length =16 bits = 2 bytes, add 8 unsigned integers from **v**A to the corresponding 8 unsigned integers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, add 4 unsigned integers from **v**A to the corresponding 4 unsigned integers from **v**B Note: unsigned or signed integers can be used with these instructions

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Add Unsigned Integer [b,h,w] Saturate	**vaddubs** **vadduhs** **vadduws**	**v**D,**v**A,**v**B	Place the sum (**v**A[unsigned integer elements]) + (**v**B[unsigned integer elements]) into **v**D[unsigned integer elements] using saturate clamping mode. Saturate clamping mode means if the resulting sum is >(2^n-1) saturate to (2^n-1), where n = **b**,**h**,**w.** Fo**r b**, byte, integer length = 8 bits = 1 byte, add 16 unsigned integers from **v**A to the corresponding 16 unsigned integers from vB For **h**, half word, integer length = 16 bits = 2 bytes, add 8 unsigned integers from **v**A to the corresponding 8 unsigned integers formable For **w**, word, integer length = 32 bits = 4 bytes, add 4 unsigned integers from **v**A to the corresponding 4 unsigned integers from **v**B If the result saturates, VSCR[SAT] is set.
Vector Add Signed Integer[b.h.w] Saturate	**vaddsbs** **vaddshs** **vddsws**	**v**D**,v**A**,v**B	Place the sum (**v**A[signed integer elements]) + (**v**B[signed integer elements]) into **v**D[signed integer elements] using saturate clamping mode. Saturate clamping mode means: if the sum is >(2^{n-1}-1) saturate to (2^{n-1}-1) and if < (- 2^{n-1}) saturate to (-2^{n-1}), where n = **b**,**h**,**w.** For **b**, byte, integer length = 8 bits = byte, add 16 signed integers from **v**A to the corresponding 16 signed integers from **v**B For **h**, half word, integer length = 16 bits = 2 bytes, add 8 signed integers from **v**A to the corresponding 8 signed integers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, add 4 signed integers from **v**A to the corresponding 4 signed integers from **v**B If the result saturates, VSCR[SAT] is set.
Vector Add and Write Carry-out Unsigned Word	**vaddcuw**	**v**D**,v**A**,v**B	Take the carry out of summing (**v**A) + (**v**B) and place it into **v**D. For **w**, word, integer length = 32 bits = 2 bytes, add 4 unsigned integers from **v**A to the corresponding 4 unsigned integers from **v**B and the resulting carry outs are correspondingly placed in **v**D.
Vector Subtract Unsigned Integer Modulo	**vsububm** **vsubuhm** **vsubuwm**	**v**D**,v**A**,v**B	Place the unsigned integer sum (**v**A) - (**v**B) into **v**D using modulo arithmetic. For **b**, byte, integer length = 8 bits =1 byte, subtract 16 unsigned integers in **v**B from the corresponding 16 unsigned integers in **v**A For **h**, half word, integer length = 16 bits = 2 bytes, subtract 8 unsigned integers in **v**B from the corresponding 8 unsigned integers in **v**A For **w**, word, integer length = 32 bits = 4 bytes, subtract 4 unsigned integers in **v**B from the corresponding 4 unsigned integers in **v**A Note that unsigned or signed integers can be used with these instructions

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Subtract Unsigned Integer Saturate	**vsububs** **vsubuhs** **vsubuws**	**v**D**,v**A**,v**B	Place the unsigned integer sum **v**A - **v**B into **v**D using saturate clamping mode, that is, if the sum < 0, it saturates to 0 corresponding to **b**,**h**,**w**. For **b**, byte, integer length = 8 bits = 1 byte, subtract 16 unsigned integers in **v**B from the corresponding 16 unsigned integers in **v**A For **h**, half word, integer length =16 bits = 2 bytes, subtract 8 unsigned integers in **v**B from the corresponding 8 unsigned integers in **v**A For **w**, word, integer length = 32 bits = 4 bytes, subtract 4 unsigned integers in **v**B from the corresponding 4 unsigned integers in **v**A If the result saturates, VSCR[SAT] is set.
Vector Subtract Signed Integer Saturate	**vsububs** **vsubuhs** **vsubuws**	**v**D**,v**A**,v**B	Place the signed integer sum (**v**A) - (**v**B) into **v**D using saturate clamping mode. Saturate clamping mode means: if the sum is >(2^{n-1}-1) saturate to (2^{n-1}-1) and if < (- 2^{n-1}) saturate to (-2^{n-1}), where n= **b**,**h**,**w.** For **b**, byte, integer length = 8 bits = 1 byte, subtract 16 signed integers in **v**B from the corresponding 16 signed integers in **v**A For **h**, half word, integer length = 16 bits = 2 bytes, subtract 8 signed integers in **v**B from the corresponding 8 signed integers in **v**A For **w**, word, integer length = 32 bits = 4 bytes, subtract 4 signed integers in **v**B from the corresponding 4 signed integers in **v**A
Vector Subtract and Write Carry-out Unsigned Word	**vasubcuw**	**v**D**,v**A**,v**B	Take the carry out of the sum (**v**A) - (**v**B) and place it into **v**D. For **w**, word, integer length = 32 bits = 2 bytes, subtract 4 unsigned integers in **v**B from the corresponding 4 unsigned integers in **v**A and place the resulting carry outs into **v**D.
Vector Multiply Odd Unsigned Integer [b,h] Modulo	**vmuloub** **vmulouh**	**v**D,**v**A,**v**B	Place the unsigned integer products of (**v**A) * (**v**B) into **v**D using modulo arithmetic mode. For **b**, byte, integer length = 8 bits =1 byte, multiply 8 odd-numbered unsigned integer byte elements from **v**A to the corresponding 8 odd-numbered unsigned integer byte elements from **v**B resulting in 8 unsigned integer half-word products in **v**D. For **h**, half word, integer length =16 bits = 2 bytes, multiply 4 odd-numbered unsigned integer half word elements from **v**A to the corresponding 4 odd numbered unsigned integer half-word elements from **v**B resulting in 4 unsigned integer word products in v**D**.

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Multiply Odd Signed Integer [b,h] Modulo	**vmulosb** **vmulosh**	**v**D,**v**A,**v**B	Place the signed integer product of (**v**A) * (**v**B) into **v**D using modulo arithmetic mode. For **b**, byte, integer length = 8 bits = 1 byte, multiply 8 odd-numbered signed integer byte elements from **v**A to 8 odd-numbered signed integer byte elements from **v**B resulting in 8 signed integer half-word products in v**D**. For **h**, half word, integer length = 16 bits = 2 bytes, multiply 4 odd-numbered signed integer half word elements from **v**A to 4 odd-numbered signed integer half word elements from **v**B resulting in 4 signed integer word products in v**D**.
Vector Multiply Even Unsigned Integer [b,h] Modulo	**vmuleub** **vmuleuh**	**v**D,**v**A,**v**B	Place the unsigned integer products of (**v**A) * (**v**B) into **v**D using modulo arithmetic mode. For **b**, byte, integer length = 8 bits =1 byte, multiply 8 even-numbered unsigned integer byte elements from **v**A to 8 even-numbered unsigned integer byte elements from **v**B resulting in 8 unsigned integer half-word products in v**D**. For **h**, half word, integer length = 16 bits = 2 bytes, multiply 4 even-numbered unsigned integer half-word elements from **v**A to 4 even numbered unsigned integer half- word elements from **v**B resulting in 4 unsigned integer word products in v**D**
Vector Multiply Even Signed Integer [b,h] Modulo	**vmulesb** **vmulesh**	**v**D,**v**A,**v**B	Place the signed integer product of (**v**A) * (**v**B) into **v**D using modulo arithmetic mode. For **b**, byte, integer length = 8 bits = 1 byte, multiply 8 even-numbered signed integer byte elements from **v**A to 8 even-numbered signed integer byte elements from **v**B resulting in 8 signed integer half-word products in v**D**. For **h**, half word, integer length = 16 bits = 2 bytes, multiply 4 even-numbered signed integer half-word elements from **v**A to 4 even-numbered signed integer half-word elements from **v**B resulting in 4 signed integer word products in v**D**.
Vector Multiply-High and Add Signed Half-Word Saturate	**vmhaddshs**	**v**D,**v**A,**v**B, **v**C	The 17 most significant bits (msb's)of the product of (**v**A) * (**v**B) adds to sign-extended **v**C and places the result into **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, multiply the 8 signed half words from **v**A with the corresponding 8 signed half words from **v**B to produce a 32-bit intermediate product and then take the 17 msb's (bits 0–16) of the 8 intermediate products and add them to the 8 sign-extended half words in **v**C, place the 8 half-word saturated results in **v**D. If the intermediate product is as follows: > $(2^{15}–1)$ saturate to $(2^{15}–1)$ and if < $–2^{15}$ saturate to $–2^{15}$. If the results saturates, VSCR[SAT] is set.

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Multiply-High Round and Add Signed Half-Word Saturate	**vmhraddshs**	**v**D,**v**A,**v**B,**v**C	Add the rounded product of (**v**A) * (**v**B) to sign-extended **v**C and place the result into **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, multiply the 8 signed integers from **v**A to the corresponding 8 signed integers from **v**B and then round the 8 immediate products by adding the value 0x0000_4000 to it. Then add the most significant bits (msb's), bits 0–16, of the 8 rounded immediate products to the 8 sign-extended values in **v**C and place the 8 signed half-word saturated results into **v**D. If the intermediate product is: > $(2^{15}–1)$ saturate to $(2^{15}–1)$ and if < $–2^{15}$ saturate to $–2^{15}$. If the result saturates, VSCR[SAT] is set.
Vector Multiply-Low and Add Unsigned Half-Word Modulo	**vmladduhm**	**v**D,**v**A,**v**B,**v**C	Add the product of (**v**A) * (**v**B) to zero-extended **v**C and place into **v**D. For **h**, half word, integer length =16 bits = 2 bytes, multiply the 8 signed integers from **v**A to the corresponding 8 signed integers from **v**B to produce a 32-bit intermediate product. The 16-bit value in **v**C is zero-extended to 32 bits and added to the intermediate product and the lower 16 bits of the sum (bit 16–31) is placed in **v**D. Note that unsigned or signed integers can be used with these instructions
Vector Multiply-Sum Unsigned Integer [b,h] Modulo	**vmsumubm** **vmsumuhm**	**v**D,**v**A,**v**B,**v**C	The product of (**v**A) * (**v**B) is added to zero-extended **v**C and placed into **v**D using modulo arithmetic. For **b**, byte, integer length = 8 bits = 1 byte, multiply 4 unsigned integer bytes from a word element in **v**A by the corresponding 4 unsigned integer bytes in a word element in **v**B and the sum of these products are added to the zero-extended unsigned integer word element in **v**C and then placed the unsigned integer word result into **v**D, following this process for each 4-word element in **v**A and **v**B. For **h**, half word, integer length = 16 bits = 2 bytes, multiply 2 unsigned integer half words from a word element in **v**A by the corresponding 2 unsigned integer half words in a word element in **v**B and the sum of these products are added to zero-extended unsigned integer word element in **v**C and then place the unsigned integer word result into **v**D, following this process for each 4 word element in **v**A and **v**B.
Vector Multiply-Sum Signed Half-Word Saturate	**vmsumshs**	**v**D,**v**A,**v**B,**v**C	Add the product of (**v**A) * (**v**B) to **v**C and place the result into **v**D using saturate clamping mode. For **h**, half word, integer length = 16 bits = 2 bytes, multiply 2 signed integer half words from a word element in **v**A by the corresponding 2 signed integer half words in a word element in **v**B. Add the sum of these products to the signed integer word element in **v**C and then place the signed integer word result into **v**D, (following this process for each 4-word element in **v**A and **v**B). If the intermediate result is > $(2^{31}–1)$, saturate to $(2^{31}–1)$ and if the result is < -2^{31}, saturate to -2^{31}. If the result saturates, VSCR[SAT] is set.

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Multiply-Sum Unsigned Half-Word Saturate	**vmsumuhs**	**v**D,**v**A,**v**B,**v**C	Add the product of (**v**A) * (**v**B) to zero-extended **v**C and place the result into **v**D using saturate clamping mode. For **h**, half word, integer length = 16 bits = 2 bytes, multiply 2 unsigned integer half words from a word element in **v**A by the corresponding 2 unsigned integer half words in a word element in **v**B. Add the sum of these products to the zero-extended unsigned integer word element in **v**C and then place the unsigned integer word result into **v**D, (following this process for each 4-word element in **v**A and **v**B). If the intermediate result is > (2^{32}–1) saturate to (2^{32}–1). If the result saturates, VSCR[SAT] is set.
Vector Multiply-Sum Mixed Byte Modulo	**vmsummbm**	**v**D,**v**A,**v**B,**v**C	Add the product of (**v**A) * (**v**B) to **v**C and place into **v**D using modulo arithmetic. For **b**, byte, integer length = 8 bits = 1 byte, multiply 4 signed integer bytes from a word element in **v**A by the corresponding 4 unsigned integer bytes from a word element in **v**B. Add the sum of these 4 signed products to the signed integer word element in **v**C and then place the signed integer word result into **v**D, following this process for each 4-word element in **v**A and **v**B.
Vector Multiply-Sum Signed Half-Word Modulo	**vmsumshm**	**v**D,**v**A,**v**B,**v**C	Add the product of (**v**A) * (**v**B) to **v**C and place into **v**D using modulo arithmetic. For **h**, half word, integer length = 16 bits = 2 bytes, multiply 2 signed integer half words from a word element in **v**A by the corresponding 2 signed integer half words in a word element in **v**B. Add the sum of these 2 products to the signed integer word element in **v**C and then place the signed integer word result into **v**D, following this process for each 4-word element in **v**A and **v**B.
Vector Sum Across Signed Word Saturate	**vsumsws**	**v**D,**v**A,**v**B	Place the sum of signed word elements in **v**A and the word in **v**B[96–127] into **v**D. For **w**, word, integer length = 32 bits = 4 bytes, add the sum of the 4 signed integer word elements in vA to the word element in vB[96-127]. If the intermediate product is > (2^{31}–1) saturate to (2^{31}–1) and if < -2^{31} saturate to -2^{31}. Place the signed integer result in **v**D[96-127],**v**D[0-95] are cleared.

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Sum Across Partial (1/2) Signed Word Saturate	**vsum2sws**	**v**D,**v**A,**v**B	Add **v**A[word 0 + word 1] + **v**B[word 1] and place in **v**D[word 1]. Repeat only add **v**A[word 2 + word 3] + **v**B[word 3] and place in **v**D[word 3]. word 0 = Bits 0–31 word 1 = Bits 32-63 word 2 = Bits 64-95 word 3 = Bits 96-127, See Figure1-2, "Big-Endian Byte Ordering for a Vector Register" for a picture of what the word elements would look like in a vector register. Add the sum of word 0 and word 1 of **v**A to word 1 of **v**B using saturate clamping mode and place the result is into word 1of **v**D. Then add the sum of word 2 and word 3 of (**v**A) to word 3 of **v**B using saturate clamping mode and place those results into word 3 in **v**D. If the intermediate result for either calculation is > $(2^{31}–1)$ then saturate to $(2^{31}–1)$ and if < $–2^{31}$ then saturate to $–2^{31}$.
Vector Sum Across Partial (1/4) Unsigned Byte Saturate	**vsum4ubs**	**v**D,**v**A,**v**B	Add **v**A[sum of 4 byte elements in word] and **v**B[word element] then place in **v**D[word element] using saturate clamping mode. For **b**, byte, integer length = 8 bits = 1 byte, for each word element in **v**B, add the sum of 4 unsigned bytes in the word in vA to the unsigned word element in **v**B and then place the results into the corresponding unsigned word element in **v**D. If the intermediate result for is > $(2^{32}–1)$ it saturates to $(2^{32}–1)$. If the result saturates, VSCR[SAT] is set.
Vector Sum Across Partial (1/4) Signed Integer Saturate	**vsum4sbs** **vsum4shs**	**v**D,**v**A,**v**B	Add **v**A[sum of signed integer elements in word] and **v**B[word element] then place in **v**D[word element] using saturate clamping mode. For **b**, byte, integer length = 8 bits = 1 byte, for each word element in **v**B, add the sum of 4 signed bytes in the word in **v**A to the signed word element in **v**B and then place the results into the corresponding signed word element in **v**D. If the intermediate result is > $(2^{31}–1)$ then saturate to $(2^{31}–1)$ and if < $–2^{31}$ then saturate to $–2^{31}$. For **h**, half word, integer length = 16 bits = 2 bytes, for each word element in **v**B, add the sum of 2 signed half words in the word in vA to the signed word element in **v**B and then place the results into the corresponding signed word element in **v**D. If the intermediate result is > $(2^{31}–1)$ then saturate to $(2^{31}–1)$ and if < $–2^{31}$ then saturate to $–2^{31}$. If the result saturates, VSCR[SAT] is set.

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Average Unsigned Integer	**vavgub** **vavguh** **vavguw**	**v**D,**v**A,**v**B	Add the sum of (**v**A[unsigned integer elements]+ **v**B[unsigned integer elements]) +1 and place into **v**D using modulo arithmetic. For **b**, byte, integer length = 8 bits = 1 byte, add 16 unsigned integers from **v**A to 16 unsigned integers from **v**B and then add 1 to the sums and place the high order result in **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, add 8 unsigned integers from **v**A to 8 unsigned integers from **v**B and then add 1 to the sums and place the high order result in **v**D. For **w**, word, integer length = 32 bits = 4 bytes, add 4 unsigned integers from **v**A to 4 unsigned integers from **v**B and then add 1 to the sums and place the high order result in **v**D. If the result saturates, VSCR[SAT] is set.
Vector Average Signed Integer	**vavgsb** **vavgsh** **vavgsw**	**v**D,**v**A,**v**B	Add the sum of (**v**A[signed integer elements]+ **v**B[signed integer elements]) +1 and place into **v**D using modulo arithmetic. For **b**, byte, integer length = 8 bits = 1 byte, add 16 signed integers from **v**A to 16 signed integers from **v**B and then add 1 to the sums and place the high order result in **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, add 8 signed integers from **v**A to 8 signed integers from **v**B and then add 1 to the sums and place the high order result in **v**D. For **w**, word, integer length = 32 bits = 4 bytes, add 4 signed integers from **v**A to 4 signed integers from **v**B and then add 1 to the sums and place the high order result in **v**D.
Vector Maximum Unsigned Integer	**vmaxub** **vmaxuh** **vmaxuw**	**v**D,**v**A,**v**B	Compare the maximum of **v**A and **v**B unsigned integers for each integer value and which ever value is larger, place that unsigned integer value into **v**D For **b**, byte, integer length = 8 bits = 1 byte, compare 16 unsigned integers from **v**A with 16 unsigned integers from **v**B. For **h**, half word, integer length = 16 bits = 2 bytes, compare 8 unsigned integers from **v**A with 8 unsigned integers from **v**B. For **w**, word, integer length = 32 bits = 4 bytes, compare 4 unsigned integers from **v**A with 4 unsigned integers from **v**B.
Vector Maximum Signed Integer	**vmaxsb** **vmaxsh** **vmaxsw**	**v**D,**v**A,**v**B	Compare the maximum of **v**A and **v**B signed integers for each integer value and which ever value is larger, place that signed integer value into **v**D For **b**, byte, integer length = 8 bits =1 byte, compare 16 signed integers from **v**A with 16 signed integers from **v**B For **h**, half word, integer length =16 bits = 2 bytes, compare 8 signed integers from **v**A with 8 signed integers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, compare 4 signed integers from **v**A with 4 signed integers from **v**B

Table 4-1. Vector Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Minimum Unsigned Integer	**vminub** **vminuh** **vminuw**	**v**D,**v**A,**v**B	Compare the minimum of **v**A and **v**B unsigned integers for each integer value and which ever value is smaller, place that unsigned integer value into **v**D For **b**, byte, integer length = 8 bits = 1 byte, compare 16 unsigned integers from **v**A with 16 unsigned integers from **v**B For **h**, half word, integer length = 16 bits = 2 bytes, compare 8 unsigned integers from **v**A with 8 unsigned integers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, compare 4 unsigned integers from **v**A with 4 unsigned integers from **v**B
Vector Minimum Signed Integer	**vminsb** **vminsh** **vminsw**	**v**D,**v**A,**v**B	Compare the minimum of **v**A and **v**B signed integers for each integer value and which ever value is smaller, place that signed integer value into **v**D. For **b**, byte, integer length = 8 bits = 1 byte, compare 16 signed integers from **v**A with 16 signed integers from **v**B For **h**, half word, integer length = 16 bits = 2 bytes, compare 8 signed integers from **v**A with 8 signed integers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, compare 4 signed integers from **v**A with 4 signed integers from **v**B

4.2.1.3 Vector Integer Compare Instructions

The vector integer compare instructions algebraically or logically compare the contents of the elements in vector register **v**A with the contents of the elements in **v**B. Each compare result vector is comprised of TRUE (0xFF, 0xFFFF, 0xFFFFFFFF) or FALSE (0x00, 0x0000, 0x00000000) elements of the size specified by the compare source operand element (byte, half word, or word). The result vector can be directed to any vector register and can be manipulated with any of the instructions as normal data, for example, combining condition results.

Vector compares provide equal-to and greater-than predicates. Others are synthesized from these by logically combining and/or inverting result vectors.

If the record bit (Rc) is set in the integer compare instructions (shown in Table 4-3) it can optionally set the CR6 field of the PowerPC condition register. If Rc = 1 in the vector integer compare instruction, then CR6 is set to reflect the result of the comparison, as follows in Table 4-2.

Table 4-2. CR6 Field Bit Settings for Vector Integer Compare Instructions

CR Bit	CR6 Bit	Vector Compare
24	0	1 Relation is true for all element pairs (that is, **v**D is set to all ones)
25	1	0
26	2	1 Relation is false for all element pairs (that is, register **v**D is cleared)
27	3	0

Table 4-3 summarizes the vector integer compare instructions.

Table 4-3. Vector Integer Compare Instructions

Name	Mnemonic	Syntax	Operation
Vector Compare Greater than Unsigned Integer	**vcmpgtub**[.] **vcmpgtuh**[.] **vcmpgtuw**[.]	**CR06,v**D**,v**A**,v**B	Compare the value in **v**A with the value in **v**B, treating the operands as unsigned integers. Place the result of the comparison into the **v**D field specified by operand **v**D. if **v**A > **v**B then **v**D = 1's; otherwise **v**D = 0's If the record bit (Rc) is set in the vector compare instruction then **v**D == 1's, (all elements true) then CR6[0] is set **v**D == 0's, (all elements false) then CR6[2] is set For **b**, byte, integer length = 8 bits = 1 byte, compare 16 unsigned integers from **v**A to 16 unsigned integers from **v**B and place the results in the corresponding 16 elements in **v**D For **h**, half word, integer length = 16 bits = 2 bytes, compare 8 unsigned integers from **v**A to 8 unsigned integers from **v**B and place the results in the corresponding 8 elements in **v**D For **w**, word, integer length = 32 bits = 4 bytes, compare 4 unsigned integers from **v**A to 4 unsigned integers from **v**B and place the results in the corresponding 4 elements in **v**D.
Vector Compare Greater Than Signed Integer	**vcmpgtsb**[.] **vcmpgtsh**[.] **vcmpgtsw**[.]	**CR06,v**D**,v**A**,v**B	Compare the value in **v**A with the value in **v**B, treating the operands as signed integers. Place the result of the comparison into the **v**D field specified by operand **v**D if **v**A > **v**B then **v**D =1's; otherwise **v**D = 0's If the record bit (Rc) is set in the vector compare instruction then **v**D == 1's, (all elements true) then CR6[0] is set **V**D == 0's, (all elements false) then CR6[2] is set For **b**, byte, integer length = 8 bits = 1 byte, compare 16 signed integers from **v**A to 16 signed integers from **v**B and place the results in the 16 corresponding elements in **v**D For **h**, half word, integer length = 16 bits = 2 bytes, compare 8 signed integers from **v**A to 8 signed integers from **v**B and place the results in the 8 corresponding elements in **v**D For **w**, word, integer length = 32 bits = 4 bytes, compare 4 signed integers from **v**A to 4 signed integers from **v**B and place the results in the 4 corresponding elements in **v**D

Table 4-3. Vector Integer Compare Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Compare Equal To Unsigned Integer	**vcmpequb**[.] **vcmpequh**[.] **vcmpequw**[.]	**v**D**,v**A**,v**B	Compare the value in **v**A with the value in **v**B, treating the operands as unsigned integers. Place the result of the comparison into the **v**D field specified by operand **v**D. if **v**A = **v**B then **v**D =1's; otherwise **v**D = 0's If the record bit (Rc) is set in the vector compare instruction then **v**D == 1's, (all elements true) then CR6[0] is set **V**D == 0's, (all elements false) then CR6[2] is set For **b**, byte, integer length = 8 bits =1 byte, compare 16 unsigned integers from **v**A to 16 unsigned integers from **v**B and place the results in the corresponding 16 elements in **v**D For **h**, half word, integer length =16 bits = 2 bytes, compare 8 unsigned integers from **v**A to 8 unsigned integers from **v**B and place the results in the corresponding 8 elements in **v**D For **w**, word, integer length=32 bits = 4 bytes, compare 4 unsigned integers from **v**A to 4 unsigned integers from **v**B and place the results in the corresponding 4 elements in **v**D. Note: **vcmpequb**[**.**], **vcmpequh**[**.**], and **vcmpequw**[**.**] can use both unsigned and signed integers

4.2.1.4 Vector Integer Logical Instructions

The vector integer logical instructions shown in Table 4-4 perform bit-parallel operations on the operands.

Table 4-4. Vector Integer Logical Instructions

Name	Mnemonic	Syntax	Operation
Vector Logical AND	**vand**	**v**D,**v**A**,v**B	AND the contents of **v**A with **v**B and place the result into **v**D.
Vector Logical OR	**vor**	**v**D,**v**A**,v**B	OR the contents of **v**A with **v**B and place the result into **v**D.
Vector Logical XOR	**vxor**	**v**D,**v**A**,v**B	XOR the contents of **v**A with **v**B and place the result into **v**D.
Vector Logical AND with Complement	**vandc**	**v**D,**v**A**,v**B	AND the contents of **v**A with the complement of **v**B and place the result into **v**D.
Vector Logical NOR	**vnor**	**v**D,**v**A**,v**B	NOR the contents of **v**A a with **v**B and place the result into **v**D.

4.2.1.5 Vector Integer Rotate and Shift Instructions

The vector integer rotate instructions are summarized in Table 4-5.

Table 4-5. Vector Integer Rotate Instructions

Name	Mnemonic	Syntax	Operation
Vector Rotate Left Integer	**vrlb** **vrlh** **vrlw**	**v**D**,v**A**,v**B	Rotate each element in **v**A left by the number of bits specified in the low-order $\log_2$**(n**) bits of the corresponding element in **v**B. Place the result into the corresponding element of **v**D. For **b**, byte, integer length = 8 bits = 1 byte, use 16 integers from **v**A with 16 integers from **v**B For **h**, half word, integer length = 16 bits = 2 bytes, use 8 integers from **v**A with 8 **i**ntegers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, use 4 integers from **v**A with 4 integers from **v**B

The vector integer shift instructions are summarized in Table 4-6.

Table 4-6. Vector Integer Shift Instructions

Name	Mnemonic	Syntax	Operation
Vector Shift Left Integer	**vslb** **vslh** **vslw**	**v**D**,v**A**,v**B	Shift each element in **v**A left by the number of bits specified in the low-order $\log_2$(**n**) bits of the corresponding element in **v**B. If bits are shifted out of bit 0 of the element they are lost. Supply zeros to the vacated bits on the right. Place the result into the corresponding element of **v**D For **b**, byte, integer length = 8 bits = 1 byte, use 16 integers from **v**A with 16 integers from **v**B For **h**, half word, integer length = 16 bits = 2 bytes, use 8 integers from **v**A with 8 **i**ntegers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, use 4 integers from **v**A with 4 integers from **v**B
Vector Shift Right Integer	**vsrb** **vsrh** **vsrw**	**v**D**,v**A**,v**B	Shift each element in **v**A right by the number of bits specified in the low-order $\log_2$(**n**) bits of the corresponding element in **v**B. If bits are shifted out of bit **n**–1 of the element they are lost. Supply zeros to the vacated bits on the left. Place the result into the corresponding element of **v**D. For **b**, byte, integer length = 8 bits = 1 byte, use 16 integers from **v**A with 16 integers from **v**B For **h**, half word, integer length = 16 bits = 2 bytes, use 8 integers from **v**A with 8 **i**ntegers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, use 4 integers from **v**A with 4 integers from **v**B

Table 4-6. Vector Integer Shift Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Shift Right Algebraic Integer	**vsrab** **vsrah** **vsraw**	**v**D,**v**A,**v**B	Shift each element in **v**A right by the number of bits specified in the low-order $\log_2$(**n**) bits of the corresponding element in **v**B. If bits are shifted out of bit **n**–1 of the element they are lost. Replicate bit 0 of the element to fill the vacated bits on the left. Place the result into the corresponding element of **v**D. For **b**, byte, integer length = 8 bits = 1 byte, use 16 integers from **v**A with 16 integers from **v**B For **h**, half word, integer length = 16 bits = 2 bytes, use 8 integers from **v**A with 8 integers from **v**B For **w**, word, integer length = 32 bits = 4 bytes, use 4 integers from **v**A with 4 integers from **v**B

4.2.2 Vector Floating-Point Instructions

This section describes the vector floating-point instructions, that include the following:

- Vector floating-point arithmetic instructions
- Vector floating-point rounding and conversion instructions
- Vector floating-point compare instructions
- Vector floating-point estimate instructions

The AltiVec floating-point data format complies with the ANSI/IEEE-754 standard. A quantity in this format represents: a signed normalized number, a signed denormalized number, a signed zero, a signed infinity, a quiet not a number (QNaN), or a signalling NaN (SNaN). Operations perform to a Java/IEEE/C9X-compliant subset of the IEEE standard, for further details on the Java or Non-Java mode see Section 3.2.1, “Floating-Point Modes.” The AltiVec ISA does not report IEEE exceptions but rather produces default results as specified by the Java/IEEE/C9X Standard, for further details on exceptions see Section 3.2.4, “Floating-Point Exceptions.”

4.2.2.1 Floating-Point Division and Square-Root

AltiVec instructions do not have division or square-root instructions. AltiVec ISA implements Vector Reciprocal Estimate Floating-Point (**vrefp**) and Vector Reciprocal-Square-Root Estimate Floating-Point (**vrsqrtefp**) instructions along with a Vector Negative Multiply-Subtract Floating-Point (**vnmsubfp**) instruction assisting in the Newton-Raphson refinement of the estimates. To accomplish division simply multiply the dividend ($x/y = x * 1/y$) and square-root by multiplying the original number ($\sqrt{x} = x * 1/\sqrt{x}$). In this way, the AltiVec ISA provides inexpensive divides and square-roots that are fully pipelined, sub-operation scheduled, and faster even than many hardware dividers. Methods are available to further refine these to correct IEEE results, where necessary at the cost of additional software overhead.

4.2.2.1.1 Floating-Point Division

The Newton-Raphson refinement step for the reciprocal $^1/_B$ looks like this:

```
y1 = y0 + y0*(1 - B*y0),  where y0 = recip_est(B)
```

This is implemented in the AltiVec ISA as follows:

```
y0 = vrefp(B)
 t = vnmsubfp(y0,B,1)
y1 = vmaddfp(y0,t,y0)
```

This produces a result accurate to almost 24 bits of precision (except in the case where B is a sufficiently small denormalized number that **vrefp** generates an infinity, that, if important, must be explicitly guarded against).

To get a correctly rounded IEEE quotient from the above result, a second Newton-Raphson iteration is performed to get a correctly rounded reciprocal (y2) to the required 24 bits of precision, then the residual.

```
R = A - B*Q
```

is computed with **vnmsubfp** (where A is the dividend, B the divisor, and Q an approximation of the quotient from A*y2). The correctly rounded quotient can then be obtained.

```
Q' = Q + R*y2
```

The additional accuracy provided by the fused nature of the AltiVec instruction multiply-add is essential to producing the correctly rounded quotient by this method.

The second Newton-Raphson iteration may ultimately not be needed but more work must be done to show that the absolute error after the first refinement step would always be less than 1 ulp, that is a requirement of this method.

4.2.2.1.2 Floating-Point Square-Root

The Newton-Raphson refinement step for reciprocal square root looks like the following:

```
y1 = y0 + 0.5*y0*(1 - B*y0*y0),   where y0 = recip_sqrt_est(B)
```

That can be implemented as follows:

```
y0 = vrsqrtefp(B)
t0 = vmaddfp(y0,y0,0.0)
t1 = vmaddfp(y0,0.5,0.0)
t0 = vnmsubfp(B,t0,1)
y1 = vmaddfp(t0,t1,y0)
```

Various methods can further refine a correctly rounded IEEE result—all more elaborate than the simple residual correction for division, and therefore are not presented here, but most of which also benefit from the negative multiply-subtract instruction.

4.2.2.2 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in Table 4-7.

Table 4-7. Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax	Operation
Vector Add Floating-Point	**vaddfp**	**v**D,**v**A,**v**B	Add the 4-word (32-bit) floating-point elements in **v**A to the 4-word (32-bit) floating-point elements in **v**B. Round the four intermediate results to the nearest single-precision number and placed into **v**D.
Vector Subtract Floating-Point	**vsubfp**	**v**D,**v**A,**v**B	The 4-word (32-bit) floating-point values in **v**B are subtracted from the 4 32-bit values in **v**B. The four intermediate results are rounded to the nearest single-precision floating-point and placed into **v**D.
Vector Maximum Floating-Point	**vmaxfp**	**v**D,**v**A,**v**B	Compare each of the 4 single-precision word elements in **v**A to the corresponding 4 single-precision word elements in **v**B For each of the four elements, place the larger value within each pair into **v**D. **vmaxfp** is sensitive to the sign of 0.0. When both operands are ±0.0: max(+0.0,±0.0) = max(±0.0,+0.0) ⇒ +0.0 max(-0.0,-0.0) ⇒ -0.0 max(NaN,x) ⇒ QNaN where x = any value
Vector Minimum Floating-Point	**vminfp**	**v**D,**v**A,**v**B	Compare each of the 4 single-precision word elements in **v**A to the corresponding 4 single-precision word elements in **v**B For each of the four elements, place the smaller value within each pair into **v**D. **vminfp** is sensitive to the sign of 0.0. When both operands are ±0.0: min(-0.0,±0.0) = min(±0.0,-0.0) ⇒ -0.0 min(+0.0,+0.0) ⇒ +0.0 min(NaN,x) ⇒ QNaN where x = any value

4.2.2.3 Floating-Point Multiply-Add Instructions

Vector multiply-add instructions are critically important to performance since a multiply followed by a data dependent addition is the most common idiom in DSP algorithms. In most implementations, floating-point multiply-add instructions will perform with the same latency as either a multiply or add alone, thus doubling performance in comparing to the otherwise serial multiply and adds.

AltiVec floating-point multiply-adds instructions fuse (a multiply-add fuse implies that the full product participates in the add operation without rounding, only the final result rounds). This not only simplifies the implementation and reduces latency (by eliminating the intermediate rounding) but also increases the accuracy compared to separate multiply and adds.

Be careful as Java-compliant programs can not use multiply-add instructions fused directly because Java requires both the product and sum to round separately. Thus to achieve strict Java compliance, perform the multiply and add with separate instructions.

To realize multiply in the AltiVec ISA use multiply-add instructions with a zero addend (for example, **vmaddfp v**D,**v**A,**v**C,**v**B where (**v**B = 0.0).

Note that in order to use multiply-add instructions to perform an IEEE or Java-compliant multiply, the addend must be -0.0. This is necessary to insure that the sign of a zero result is correct when the product is either +0.0 or -0.0 (+0.0 + -0.0 ⇒ +0.0, and -0.0 + -0.0 ⇒ -0.0). When the sign of a resulting 0.0 is not important, then use +0.0 as the addend that may, in some cases, avoiding the need for a second register to hold a -0.0 in addition to the integer 0/floating-point +0.0 that may already be available.

The floating-point multiply-add instructions are summarized in Table 4-8.

Table 4-8. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Syntax	Operation
Vector Multiply-Add Floating-Point	**vmaddfp**	**v**D,**v**A,**v**C,**v**B	Multiply the four word floating-point elements in **v**A by the corresponding four word elements in **v**C. Add the four word elements in **v**B to the four intermediate products. Round the results to the nearest single-precision numbers and place the corresponding word elements into **v**D.
Vector Negative Multiply-Subtract Floating-Point	**vmmsubfp**	**v**D,**v**A,**v**C,**v**B	Multiply the four word floating-point elements in **v**A by the corresponding four word elements in **v**C. Subtract the four word floating-point elements in **v**B from the four intermediate products and invert the sign of the difference. Round the results to the nearest single-precision numbers and place the corresponding word elements into **v**D.

4.2.2.4 Floating-Point Rounding and Conversion Instructions

All AltiVec floating-point arithmetic instructions use the IEEE default rounding mode, round-to-nearest. The AltiVec ISA does not provide the IEEE directed rounding modes.

The AltiVec ISA provides separate instructions for converting floating-point numbers to integral floatin
g-point values for all IEEE rounding modes as follows:

- Round-to-nearest (**vrfin**) (round)
- Round-toward-zero (**vrfiz**) (truncate)
- Round-toward-minus-infinity (**vrfim**) (floor)
- Round-toward-positive-infinity (**vrfip**) (ceiling).

Floating-point conversions to integers (**vctuxs**, **vctsxs**) use round-toward-zero (truncate). The floating-point rounding instructions are shown in Table 4-9.

Table 4-9. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Syntax	Operation
Vector Round to Floating-Point Integer Nearest	**fvrfin**	**v**D,**v**B	Round to the nearest the four word floating-point elements in **v**B and place the four corresponding word elements into **v**D.

Table 4-9. Floating-Point Rounding and Conversion Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Round to Floating-Point Integer toward Zero	**fvrfiz**	**v**D,**v**B	Round towards zero the four word floating-point elements in **v**B and place the four corresponding word elements into **v**D.
Vector Round to Floating-Point Integer toward Positive Infinity	**fvrfip**	**v**D,**v**B	Round towards +Infinity the four word floating-point elements in **v**B and place the four corresponding word elements into **v**D.
Vector Round to Floating-Point Integer toward Minus Infinity	**fvrfim**	**v**D,**v**B	Round towards -Infinity the four word floating-point elements in **v**B and place the four corresponding word elements into **v**D.
Vector Convert from Unsigned Fixed-Point Word	**vcfux**	**v**D,**v**B, **UIMM**	Convert each of the four unsigned fixed-point integer word elements in **v**B to the nearest single-precision value. Divide the result by $2^{\mathbf{UIMM}}$ and place into the corresponding word element of **v**D.
Vector Convert from Signed Fixed-Point Word	**vcfsx**	**v**D,**v**B, **UIMM**	Convert each signed fixed-point integer word element in **v**B to the nearest single-precision value. Divide the result by $2^{\mathbf{UIMM}}$ and place into the corresponding word element of **v**D.
Vector Convert to Unsigned Fixed-Point Word Saturate	**vctuxs**	**v**D,**v**B, **UIMM**	Multiply each of the four single-precision word elements in **v**B by $2^{\mathbf{UIMM}}$. The products are converted to unsigned fixed-point integers using the Round toward Zero mode. If the intermediate results are > 2^{32}–1 saturate to 2^{32}–1 and if it is < 0 saturate to 0. Place the unsigned integer results into the corresponding word elements of **v**D.
Vector Convert to Signed Fixed-Point Word Saturate	**vctsxs**	**v**D,**v**B, **UIMM**	Multiply each of the four single-precision word elements in **v**B by $2^{\mathbf{UIMM}}$. The products are converted to signed fixed-point integers using Round toward Zero mode. If the intermediate results are > 2^{32}–1 saturate to 2^{32}–1 and if it is < –2^{31} saturate to –2^{31}. Place the unsigned integer results into the corresponding word elements of **v**D.

4.2.2.5 Floating-Point Compare Instructions

This section describes floating-point unordered compare instructions.

All AltiVec floating-point compare instructions (**vcmpeqfp**, **vcmpgtfp**, **vcmpgefp**, and **vcmpbfp**) return FALSE if either operand is a NaN. Not equal-to, not greater-than, not greater-than-or-equal-to, and not-in-bounds NaNs compare to everything, including themselves.

Compares always return a Boolean mask (TRUE = 0x_FFFF_FFFF, FALSE = 0x_0000_0000) and never return a NaN. The **vcmpeqfp** instruction is recommended as the Isnan(**v**X) test. No explicit unordered compare instructions or traps are provided. However, the greater-than-or-equal-to predicate (≥) (**vcmpgefp**) is provided—in addition to the > and = predicates available for integer comparison—specifically to enable IEEE unordered comparison that would not be possible with just the > and = predicates. Table 4-10 lists the six common mathematical predicates and how they would be realized in AltiVec code.

Table 4-10. Common Mathematical Predicates

Case	Mathematical Predicate	AltiVec Realization	Relations			
			a>b	a<b	a=b	?
1	a = b	a = b	F	F	T	F
2	a ≠ b (?<>)	¬ (a = b)	T	T	F	T
3	a > b	a > b	T	F	F	F
4	a < b	b > a	F	T	F	F
5	a ≥ b	¬ (b > a)	T	F	T	*T
6	a ≤ b	¬ (a > b)	F	T	T	*T
5a	a ≥ b	a ≥ b	T	F	T	F
6a	a ≤ b	b ≥ a	F	T	T	F

*** Note**: cases 5 and 6 implemented with greater-than (**vcmpgtfp** and **vnor**) would not yield the correct IEEE result when the relation is unordered.

Table 4-11 shows the remaining eight useful predicates and how they might be realized in AltiVec code.

Table 4-11. Other Useful Predicates

Case	Predicate	AltiVec Realization	Relations			
			a>b	a<b	a=b	?
7	a ? b	¬ ((a=b) ∨ (b>a) ∨ (a>b))	F	F	F	T
8	a <> b	(a ≥ b) ⊕ (b ≥ a)	T	T	F	F
9	a <=> b	(a ≥ b) ∨ (b ≥ a)	T	T	T	F
10	a ?> b	¬ (b ≥ a)	T	F	F	T
11	a ?>= b	¬ (b > a)	T	F	T	T
12	a ?< b	¬ (a ≥ b)	F	T	F	T
13	a ?<= b	¬ (a > b)	F	T	T	T
14	a ?= b	¬ ((a > b) ∨ (b > a))	F	F	T	T

The vector floating-point compare instructions compares the elements in two vector registers word-by-word, interpreting the elements as single-precision numbers. With the exception of the Vector Compare Bounds Floating-Point (**vcmpbfp**) instruction they set the target vector register, and CR[6] if Rc = 1, in the same manner as do the vector integer compare instructions.

The Vector Compare Bounds Floating-Point (**vcmpbfp**) instruction sets the target vector register, and CR[6] if Rc = 1, to indicate whether the elements in **v**A are within the bounds specified by the corresponding element in **v**B, as explained in the instruction description. A single-precision value x is said to be within the bounds specified by a single-precision value y if ($-y \leq x \leq y$).

The floating-point compare instructions are summarized in Table 4-12.

Table 4-12. Floating-Point Compare Instructions

Name	Mnemonic	Syntax	Operation
Vector Compare Greater Than Floating-Point [Record]	**vcmpgtfp**[.]	**CR6,** **v**D**,v**A**,v**B	Compare each of the 4 single-precision word elements in **v**A to the corresponding four single-precision word elements in **v**B For each element, if **v**A > **v**B then set the corresponding element in **v**D to all 1's otherwise clear the element in **v**D to all 0's If the record bit (Rc = 1) is set in the vector compare instruction, then **v**D ==1, (all elements true) then CR6[0] is set **v**D == 0, (all elements false) then CR6[2] is set
Vector Compare Equal to Floating-Point [Record]	**vcmpeqfp**[.]	**CR6,v**D**,v**A**,v**B	Compare each of the 4 single-precision word elements in **v**A to the corresponding 4 single-precision word elements in **v**B. For each element, if **v**A = **v**B then set the corresponding element in **v**D to all 1's otherwise clear the element in **v**D to all 0's If the record bit (Rc = 1) is set in the vector compare instruction then **v**D ==1, (all elements true) then CR6[0] is set **v**D == 0, (all elements false) then CR6[2] is set
Vector Compare Greater Than or Equal to Floating-Point [Record]	**vcmpgeqfp**[.]	**CR6,v**D**,v**A**,v**B	Compare each of the 4 single-precision word elements in **v**A to the corresponding 4 single-precision word elements in **v**B. For each element, if **v**A >= **v**B then set the corresponding element in **v**D to all 1's otherwise clear the element in **v**D to all 0's If the record bit (Rc = 1) is set in the vector compare instruction then **v**D ==1, (all elements true) then CR6[0] is set **v**D == 0, (all elements false) then CR6[2] is set
Vector Compare Bounds Floating-Point [Record]	**vcmpbfp**[.]	**CR6,v**D**,v**A**,v**B	Compare each of the 4 single-precision word elements in **v**A to the corresponding single-precision word elements in **v**B. A 2-bit value is formed that indicates whether the element in **v**A is within the bounds specified by the element in **v**B, as follows. Bit 0 of the two-bit value is cleared if the element in **v**A is <= to the element in **v**B, and is set otherwise. Bit 1 of the two-bit value is cleared if the element in **v**A is >= to the negation of the element in **v**B, and is set otherwise. The two-bit value is placed into the high-order two bits of the corresponding word element of **v**D and the remaining bits of the element are cleared to 0. If Rc=1, CR6[2] is set when all four elements in **v**A are within the bounds specified by the corresponding element in **v**B

4.2.2.6 Floating-Point Estimate Instructions

The floating-point estimate instructions are summarized in Table 4-13.

Table 4-13. Floating-Point Estimate Instructions

Name	Mnemonic	Syntax	Operation
Vector Reciprocal Estimate Floating-Point	**vrefp**	**v**D,**v**B	Place estimates of the reciprocal of each of the four word floating-point source elements in **v**B in the corresponding four word elements in **v**D.
Vector Reciprocal Square Root Estimate Floating-Point	**vrsqrtefp**	**v**D,**v**B	Place estimates of the reciprocal square-root of each of the four word source elements in **v**B in the corresponding four word elements in **v**D.
Vector Log2 Estimate Floating-Point	**vlogefp**	**v**D,**v**B	Place estimates of the base 2 logarithm of each of the four word source elements in **v**B in the corresponding four word elements in **v**D.
Vector 2 Raised to the Exponent Estimate Floating-Point	**vexptefp**	**v**D,**v**B	Place estimates of 2 raised to the power of each of the four word source elements in **v**B in the corresponding four word elements in **v**D.

4.2.3 Load and Store Instructions

Only very basic load and store operations are provided in the AltiVec ISA. This keeps the circuitry in the memory path fast so the latency of memory operations will be low. Instead, a powerful set of field manipulation instructions are provided to manipulate data into the desired alignment and arrangement after the data has been brought into the vector registers.

Load vector indexed (**lvx**, **lvxl**) and store vector indexed (**stvx**, **stvxl**) instructions transfer an aligned quad-word vector between memory and vector registers. Load vector element indexed (**lvebx**, **lvehx**, **lvewx**) and store vector element indexed instructions (**stvebx**, **stvehx**, **stvewx**) transfer byte, half-word, and word scalar elements between memory and vector registers.

All vector loads and vector stores use the index (**r**A|0 + **r**B) addressing mode to specify the target memory address. The AltiVec ISA does not provide any update forms. An **lvebx**, **lvehx**, or **lvewx** instruction transfers a scalar data element from memory into the destination vector register, leaving other elements in the vector with boundedly-undefined values. A **stvebx**, **stvehx**, or **stvewx** instruction transfers a scalar data element from the source vector register to memory leaving other elements in the quad word unchanged. No data alignment occurs, that is, all scalar data elements are transferred directly on their natural memory byte-lanes to or from the corresponding element in the vector register. Quad word memory accesses made by **lvx**, **lvxl**, **stvx**, and **stvxl** instructions are not guaranteed to be atomic. Direct-store segments (T=1) are not supported. Any vector load or store that attempts to access a direct-store segment will cause a DSI exception.

4.2.3.1 Alignment

All memory references must be size aligned. If a vector load or store address is not properly size aligned, the suitable number of least significant bits are ignored, and a size aligned transfer occurs instead. Data alignment must be performed explicitly after being brought into the registers. No assistance is provided to assist in aligning individual scalar elements that are not aligned on their natural size boundary. However, assistance is provided for justifying non-size-aligned vectors. This is provided through the special Load Vector for Shift Left (**lvsl**) and Load Vector for Shift Right (**lvsr**) instructions that compute the proper Vector Permute (**vperm**) control vector from the misaligned memory address. For details on how to use these instructions to align data see Section 3.1.6, "Quad-Word Data Alignment."

The **lvx**, **lvxl**, **stvx**, and **stvxl** instructions can be used to move all sorts of data, not just multimedia data, in typical PowerPC environments. Therefore, because vector loads and stores are size-aligned, care should be taken to align data on even quad-word boundaries for maximum performance.

4.2.3.2 Load and Store Address Generation

Vector load and store operations generate effective addresses using register indirect with index mode.

All AltiVec load and store instructions use register indirect with index addressing mode that cause the contents of two general-purpose registers (specified as operands **r**A and **r**B) to be added in the generation of the effective address (EA). A zero in place of the **r**A operand causes a zero to be added to the contents of the GPR specified in **r**B. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0). If the address becomes unaligned, for a half word, word, or quad word, when combining addresses (**r**A|0 + **r**B), the effective address is ANDed with the appropriate zero values to boundary align the address and is summarized in Table 4-14.

Table 4-14. Effective Address Alignment

Operand	Effective Address Bit	Setting
Indexed Half word	EA[63]	0b0
Indexed Word	EA[62–63]	0b00
Indexed Quad word	EA[60–63]	0b0000

Figure 4-1 shows how an effective address is generated when using register indirect with index addressing.

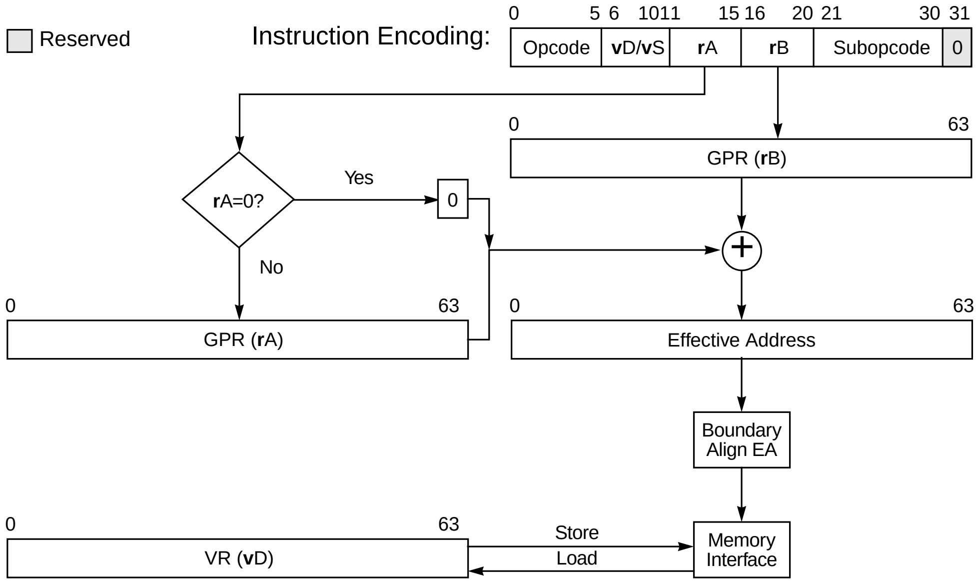


Figure 4-1. Register Indirect with Index Addressing for Loads/Stores

4.2.3.3 Vector Load Instructions

For vector load instructions, the byte, half word, or word addressed by the EA (effective address) is loaded into **r**D.

The default byte and bit ordering is big-endian as in the PowerPC architecture; see Section 3.1.2, “AltiVec Byte Ordering,” for information about little-endian byte ordering.

Table 4-15 summarizes the vector load instructions.

Table 4-15. Integer Load Instructions

Name	Mnemonic	Syntax	Operation
Load Vector Element Integer Indexed	**lvebx** **lvehx** **lvewx**	**v**D**,r**A**,r**B	The EA is the sum (**r**A|0) + (**r**B). Load the byte, half word, or word in memory addressed by the EA into the low-order bits of **v**D. The remaining bits in vD are set to boundedly undefined values. Because memory must stay aligned, the EA is set to default to alignment: For **b**, byte, integer length = 8 bits = 1 byte, For **h**, half word, integer length = 16 bits = 2 bytes, EA[62-63] is set to 0b0 For **w**, word, integer length= 32 bits = 4 bytes, EA[61-63] is set to 0b00
Load Vector Element Indexed	**lvx**	**v**D**,r**A**,r**B	The EA is the sum (**r**A|0) + (**r**B). Load the double word in memory addressed by the EA into **v**D. Because memory needs to stay aligned, the EA is set to default to alignment: For **q**, quad word, integer length =128 bits = 8 bytes, the EA[60-63] is set to 0b0000 LRU = 0 If the processor is in little-endian mode, load the double word in memory addressed by EA into **v**D[6–127] and load the double word in memory addressed by EA+8 into **v**D[0–63].
Load Vector Element Indexed LRU	**lvxl**	**v**D**,r**A**,r**B	The EA is the sum (**r**A|0) + (**r**B). Load the double word in memory addressed by the EA into **v**D. For **d**, double word, integer length=64 bits = 4 bytes, the EA[60-63] is set to 0b0000 LRU =1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future. If the processor is in little-endian mode, load the double word in memory addressed by EA into **v**D[64–127] and load the double word in memory addressed by EA+8 into **v**D[0–63].

The **lvsl** and **lvsr** instructions can be used to create the permute control vector to be used by a subsequent **vperm** instruction. Let X and Y be the contents of **v**A and **v**B specified by **vperm**. The control vector created by **lvsl** causes the **vperm** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes (sh = the value in EA[60-63]). The control vector created by l**vsr** causes the **vperm** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

These instructions can also be used to rotate or shift the contents of a vector register left **lvsl** or right **lvsr** by sh bytes. For rotating, the vector register to be rotated should be specified as both the **v**A and the **v**B register for **vperm**. For shifting left, the **v**B register for **vperm** should be a register containing all zeros and **v**A should contain the value to be shifted, and vice versa for shifting right. For further examples on how to align the data see Section 3.1.6, “Quad-Word Data Alignment.” The default byte and bit ordering is big-endian as in the PowerPC architecture; see Section 3.1.2.2, “Little-Endian Byte Ordering,” for information about little-endian byte ordering.

Table 4-16 summarizes the vector alignment instructions.

Table 4-16. Vector Load Instructions Supporting Alignment

Name	Mnemonic	Syntax	Operation
Load Vector for Shift Left	**lvsl**	**v**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The EA[60–63] = sh, then based on a table lookup place the value in **v**D if sh = 0x0 then (**v**D)0:127 <- 0x000102030405060708090A0B0C0D0E0F if sh = 0x1 then (**v**D)0:127 <- 0x0102030405060708090A0B0C0D0E0F10 if sh = 0x2 then (**v**D)0:127 <- 0x02030405060708090A0B0C0D0E0F1011 if sh = 0x3 then (**v**D)0:127 <- 0x030405060708090A0B0C0D0E0F101112 if sh = 0x4 then (**v**D)0:127 <- 0x0405060708090A0B0C0D0E0F10111213 if sh = 0x5 then **v**D)0:127 <- 0x05060708090A0B0C0D0E0F1011121314 if sh = 0x6 then (**v**D)0:127 <- 0x060708090A0B0C0D0E0F101112131415 if sh = 0x7 then (**v**D)0:127 <- 0x0708090A0B0C0D0E0F10111213141516 if sh = 0x8 then (**v**D)0:127 <- 0x08090A0B0C0D0E0F1011121314151617 if sh = 0x9 then (**v**D)0:127 <- 0x090A0B0C0D0E0F101112131415161718 if sh = 0xA then (**v**D)0:127 <- 0x0A0B0C0D0E0F10111213141516171819 if sh = 0xB then (**v**D)0:127 <- 0x0B0C0D0E0F101112131415161718191A if sh = 0xC then (**v**D)0:127 <- 0x0C0D0E0F101112131415161718191A1B if sh = 0xD then (**v**D)0:127 <- 0x0D0E0F101112131415161718191A1B1C if sh = 0xE then (**v**D)0:127 <- 0x0E0F101112131415161718191A1B1C1D if sh = 0xF then (**v**D)0:127 <- 0x0F101112131415161718191A1B1C1D1E

Table 4-16. Vector Load Instructions Supporting Alignment (Continued)

Load Vector for Shift Right	**lvsr**	**v**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The EA[60–63] = sh, then based on the table lookup below place the value in **v**D if sh = 0x0 then (**v**D)0:127 <- 0x101112131415161718191A1B1C1D1E1F if sh = 0x1 then (**v**D)0:127 <- 0x0F101112131415161718191A1B1C1D1E if sh = 0x2 then (**v**D)0:127 <- 0x0E0F101112131415161718191A1B1C1D if sh = 0x3 then (**v**D)0:127 <- 0x0D0E0F101112131415161718191A1B1C if sh = 0x4 then (**v**D)0:127 <- 0x0C0D0E0F101112131415161718191A1B if sh = 0x5 then (**v**D)0:127 <- 0x0B0C0D0E0F101112131415161718191A if sh = 0x6 then (**v**D)0:127 <- 0x0A0B0C0D0E0F10111213141516171819 if sh = 0x7 then (**v**D)0:127 <- 0x090A0B0C0D0E0F101112131415161718 if sh = 0x8 then (**v**D)0:127 <- 0x08090A0B0C0D0E0F1011121314151617 if sh = 0x9 then (**v**D)0:127 <- 0x0708090A0B0C0D0E0F10111213141516 if sh = 0xA then (**v**D)0:127 <- 0x060708090A0B0C0D0E0F101112131415 if sh = 0xB then (**v**D)0:127 <- 0x05060708090A0B0C0D0E0F1011121314 if sh = 0xC then (**v**D)0:127 <- 0x0405060708090A0B0C0D0E0F10111213 if sh = 0xD then (**v**D)0:127 <- 0x030405060708090A0B0C0D0E0F101112 if sh = 0xE then (**v**D)0:127 <- 0x02030405060708090A0B0C0D0E0F1011 if sh = 0xF then (**v**D)0:127 <- 0x0102030405060708090A0B0C0D0E0F10

4.2.3.4 Vector Store Instructions

For vector store instructions, the contents of vector register used as a source (**v**S) are stored into the byte, half word, word or quad word in memory addressed by the effective address (EA). Table 4-17 provides a summary of the vector store instructions.

Table 4-17. Integer Store Instructions

Name	Mnemonic	Syntax	Operation
Store Vector Element Integer Indexed	**svetbx** **svethx** **svetwx**	**v**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). Store the contents of the low-order bits of **v**S into the integer in memory addressed by the EA. Because memory needs to stay aligned, the EA is set to default to alignment: For **b**, byte, integer length = 8 bits =1 byte, For **h**, half word, integer length = 16 bits = 2 bytes, EA[62–63] is set to 0b0 For **w**, word, integer length = 32 bits = 4 bytes, EA[61–63] is set to 0b00
Store Vector Element Indexed	**stvx**	**v**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). Store the contents of **v**S into the quad word in memory addressed by the EA. For **q**, quad word, integer length = 64 bits = 4 bytes, the EA[60–63] is set to 0b0000 LRU = 0 If the processor is in little-endian mode, store the contents of **v**S[64–127] into the double word in memory addressed by EA, and store the contents of **v**S[0–63] into the double word in memory addressed by EA+8.
Store Vector Element Indexed LRU	**stvxl**	**v**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). Store the contents of **v**S into the quad word in memory addressed by the EA. For **d**, double word, integer length=64 bits = 4 bytes, the EA[60–63] is set to 0b0000 LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future. If the processor is in little-endian mode, store the contents of **v**S[64–127] into the double word in memory addressed by EA, and store the contents of **v**S[0–63] into the double word in memory addressed by EA+8.

4.2.4 Control Flow

AltiVec instructions can be freely intermixed with existing PowerPC instructions to form a complete program. AltiVec instructions do provide a vector compare and select mechanism to implement conditional execution as the preferred mechanism to control data flow in AltiVec programs. And AltiVec vector compare instructions can update the condition register thus providing the communication from AltiVec execution units to PowerPC branch instructions necessary to modify program flow based on vector data.

4.2.5 Vector Permutation and Formatting Instructions

Vector pack, unpack, merge, splat, permute, and select can be used to accelerate various vector math and vector formatting. Details of the various instructions follow.

4.2.5.1 Vector Pack Instructions

Half-word vector pack instructions (**vpkuhum**, **vpkuhus**, **vpkshus**, **vpkshss**) truncate the sixteen half words from two concatenated source operands producing a single result of sixteen bytes (quad word) using either modulo(2^8), 8-bit signed-saturation, or 8-bit unsigned-saturation to perform the truncation. Similarly, word vector pack instructions (**vpkuwum**, **vpkuwus**, **vpkswus**, **vpksws**) truncate the eight words from two concatenated source operands producing a single result of eight half words using modulo(2^16), 16-bit signed-saturation, or 16-bit unsigned-saturation to perform the truncation.

One special purpose form of Vector Pack Pixel (**vpkpx**) instruction is provided that packs eight 32-bit (8/8/8/8) pixels from two concatenated source operands into a single result of eight 16-bit 1/5/5/5 αRGB pixels. The least significant bit of the first 8-bit element becomes the 1-bit α field, and each of the three 8-bit R, G, and B fields are reduced to 5 bits by discarding the 3 lsbs.

Table 4-18 describes the vector pack instructions.

Table 4-18. Vector Pack Instructions

Name	Mnemonic	Syntax	Operation
Vector Pack Unsigned Integer [h,w] Unsigned Modulo	**vpkuhum** **vpkuwum**	**v**D, **v**A, **v**B	Concatenate the low-order unsigned integers of **v**A and the low-order unsigned integers of **v**B and place into **v**D using unsigned modulo arithmetic. **v**A is placed in the lower order double word of **v**D and **v**B is placed into the higher order double word of **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 8 unsigned integers, in other words the 8 low-order bytes of the half words from **v**A and **v**B For **w**, word, integer length = 32 bits = 4 bytes,4 unsigned integers, in other words the 4 low-order half words of the words from **v**A and **v**B
Vector Pack Unsigned Integer [h,w] Unsigned Saturate	**vpkuhus** **vpkuwus**	**v**D, **v**A, **v**B	Concatenate he low-order unsigned integers of **v**A and the low-order unsigned integers of **v**B and place into **v**D using unsigned saturate clamping mode. **v**A is placed in the lower order double word of **v**D and **v**B is placed into the higher order double word of **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 8 unsigned integers, in other words the 8 low-order bytes of the half words from **v**A and **v**B For **w**, word, integer length = 32 bits = 4 bytes,4 unsigned integers, in other words the 4 low-order words of the half words from **v**A and **v**B
Vector Pack Signed Integer [h,w] Unsigned Saturate	**vpkshus** **vpkswus**	**v**D, **v**A, **v**B	Concatenate the low-order signed integers of **v**A and the low-order signed integers of **v**B and place into **v**D using unsigned saturate clamping mode. **v**A is placed in the lower order double word of **v**D and **v**B is placed into the higher order double word of **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 8 signed integers, in other words the 8 low-order bytes of the half word from **v**A and **v**B For **w**, word, integer length = 32 bits = 4 bytes, 4 signed integers, in other words the 4 low-order half words of the words from **v**A and **v**B

Table 4-18. Vector Pack Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector Pack Signed Integer [h,w] Unsigned Saturate	**vpkshss** **vpkswss**	**v**D**, v**A**, v**B	Concatenate the low-order signed integers of **v**A and the low-order signed integers of **v**B are concatenated and place into **v**D using signed saturate clamping mode. **v**A is placed in the lower order double word of **v**D and **v**B is placed into the higher order double word of **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 8 signed integers, in other words the 8 low-order bytes of the half word from **v**A and **v**B For **w**, word, integer length = 32 bits = 4 bytes, 4 signed integers, in other words the 4 low-order half words of the words from **v**A and **v**B
Vector Pack Pixel	**vpkpx**	**v**D**, v**A**, v**B	Each word element in **v**A and **v**B is packed to 16 bits and the half word is placed into **v**D. Each word from **v**A and **v**B is packed to 16 bits in the following order: [bit 7 of the first byte (bit 7 of the word)] [bits 0–4 of the second byte (bits 8–12 of the word) [bits 0–4 of the third byte (bits 16–20 of the word)] [bits 0–4 of the fourth byte (bits 24–28 of the word)] **v**A half words are placed in the lower order double word of **v**D and **v**B half words are placed into the higher order double word of **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 8 signed integers, in other words the 8 low-order bytes of the half word from **v**A and **v**B For **w**, word, integer length = 32 bits = 4 bytes, 4 signed integers, in other words the 4 low-order half words of the words from **v**A and **v**B

4.2.5.2 Vector Unpack Instructions

Byte vector unpack instructions unpack the 8 low bytes (or 8 high bytes) of one source operand into 8 half words using sign extension to fill the MSBs. Half word vector unpack instructions unpack the 4 low half words (or 4 high half words) of one source operand into 4 words using sign extension to fill the MSbs.

A special purpose form of vector unpack is provided, the Vector Unpack Low Pixel (**vupklpx**) and the Vector Unpack High Pixel (**vupkhpx**) instructions for 1/5/5/5 αRGB pixels. The 1/5/5/5 pixel vector unpack, unpacks the four low 1/5/5/5 pixels (or four 1/5/5/5 high pixels) into four 32-bit (8/8/8/8) pixels. The 1-bit α element in each pixel is sign extended to 8 bits, and the 5-bit R, G, and B elements are each zero extended to 8 bits.

Table 4-19 describes the unpack instructions.

Table 4-19. Vector Unpack Instructions

Name	Mnemonic	Syntax	Operation
Vector Unpack High Signed Integer	**vupkhsb** **vupkhsh**	**v**D, **v**B	Each signed integer element in the high order double word of v**B** is sign extended to fill the MSBs in a signed integer and then is placed into **v**D. For **b**, byte, integer length = 8 bits = 1 byte, 8 signed bytes from the high order double word of **v**B are unpacked and sign extended to 8 half words into **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 8 signed half words from the high order double word of **v**B are unpacked and sign extended to 4 words into **v**D
Vector Unpack High Pixel	**vupkhpx**	**v**D, **v**B	Each half-word element in the high order double word of v**B** is unpacked to produce a 32-bit word that is then placed in the same order into **v**D. A half-word element is unpacked to 32 bits by concatenating, in order, the results of the following operations. sign-extend bit 0 of the half word to 8 bits zero-extend bits 1–5 of the half word to 8 bits zero-extend bits 6–10 of the half word to 8 bits zero-extend bits 11–15 of the half word to 8 bits
Vector Unpack Low Signed Integer	**vupklsb** **vupklsh**	**v**D, **v**B	Each signed integer element in the low-order double word of v**B** is sign extended to fill the MSBs in a signed integer and then is placed into **v**D. For **b**, byte, integer length = 8 bits = 1 byte, 8 signed bytes from the low-order double word of **v**B are unpacked and sign extended to 8 half words into **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 8 signed half words from the low-order double word of **v**B are unpacked and sign extended into 4 words in **v**D
Vector Unpack Low Pixel	**vupklpx**	**v**D, **v**B	Each half-word element in the low-order double word of v**B** is unpacked to produce a 32-bit word that is then placed in the same order into **v**D. A half-word element is unpacked to 32 bits by concatenating, in order, the results of the following operations. sign-extend bit 0 of the half word to 8 bits zero-extend bits 1–5 of the half word to 8 bits zero-extend bits 6–10 of the half word to 8 bits zero-extend bits 11–15 of the half word to 8 bits

4.2.5.3 Vector Merge Instructions

Byte vector merge instructions interleave the 8 low bytes (or 8 high bytes) from two source operands producing a result of 16 bytes. Similarly, half-word vector merge instructions interleave the 4 low half words (or 4 high half words) of two source operands producing a result of 8 half words, and word vector merge instructions interleave the 2 low words (or 2 high words) from two source operands producing a result of 4 words. The vector merge instruction has many uses, notable among them is a way to efficiently transpose SIMD vectors. Table 4-20 describes the merge instructions.

Table 4-20. Vector Merge Instructions

Name	Mnemonic	Syntax	Operation
Vector Merge High Integer	**vmrghb** **vmrghh** **vmrghw**	**v**D, **v**A, **v**B	Each integer element in the high order double word of **v**A is placed into the low-order integer element in **v**D. Each integer element in the high order double word of **v**B is placed into the high order integer element in **v**D. For **b**, byte, integer length = 8 bits = 1 byte, 8 bytes from the high order double word of **v**A are placed into the low-order byte of each half word in **v**D and 8 bytes from the high order double word of **v**B are placed into the high order byte of each half word in **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 4 half words from the high order double word of **v**A are placed into the low-order half word of each word in **v**D and 4 half words from the high order double word of **v**B are placed into the high order half word of each word in **v**D. For **w**, word, integer length = 32 bits = 4 bytes, 2 words from the high order double word of **v**A are placed into the low-order word of each double word in **v**D and 2 words from the high order double word of **v**B are placed into the high order word of each double word in **v**D.
Vector Merge Low Integer	**vmrglb** **vmrglh** **vmrglw**	**v**D, **v**A, **v**B	Each integer element in the low-order double word of **v**A is placed into the low-order integer element in **v**D. Each integer element in the low-order double word of **v**B is placed into the high order integer element in **v**D. For **b**, byte, integer length = 8 bits = 1 byte, 8 bytes from the low-order double word of **v**A are placed into the low-order byte of each half word in **v**D and 8 bytes from the low-order double word of **v**B are placed into the high order byte of each half word in **v**D. For **h**, half word, integer length = 16 bits = 2 bytes, 4 half words from the low-order double word of **v**A are placed into the low-order half word of each word in **v**D and 4 half words from the low-order double word of **v**B are placed into the high order half word of each word in **v**D. For **w**, word, integer length = 32 bits = 4 bytes, 2 words from the low-order double word of **v**A are placed into the low-order word of each double word in **v**D and 2 words from the low-order double word of **v**B are placed into the high order word of each double word in **v**D.

4.2.5.4 Vector Splat Instructions

When a program needs to perform arithmetic vector, the vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a Vector Register by a constant). Vector splat instructions can be used to move data where it is required. For example to multiply all elements of a vector register by a constant, the vector splat instructions can be used to splat the scalar into the vector register. Likewise, when storing a scalar into an arbitrary memory location, it must be splatted into a vector register, and that register specified as the source of the store. This will guarantee that the data appears in all possible positions of that scalar size for the store. Table 4-21 describes the vector splat instructions.

Table 4-21. Vector Splat Instructions

Name	Mnemonic	Syntax	Operation
Vector Splat Integer	**vspltb** **vsplth** **vspltw**	**v**D, **v**B, **UIMM**	Replicate the contents of element UIMM in **v**B and place into each element in **v**D. For **b**, byte, integer length = 8 bits = 1 byte, each element is a byte. For **h**, half word, integer length =16 bits = 2 bytes, each element is a half word. For **w**, word, integer length = 32 bits = 4 bytes, 2 words each element is a word.
Vector Splat Immediate Signed Integer	**vspltisb** **vspltish** **vspltisw**	**v**D, **SIMM**	Sign-extend the value of the SIMM field to the length of the element and replicate that value and place into each element in **v**D. For **b**, byte, integer length = 8 bits = 1 byte, each element is a byte. For **h**, half word, integer length =16 bits = 2 bytes, each element is a half word. For **w**, word, integer length = 32 bits = 4 bytes, 2 words each element is a word.

4.2.5.5 Vector Permute Instructions

Permute instructions allow any byte in any two source vector registers to be directed to any byte in the destination vector. The fields in a third source operand specify from which field in the source operands the corresponding destination field will be taken. The Vector Permute (**vperm**) instruction is a very powerful one that provides many useful functions. For example, it provides a good way to perform table-lookups and data alignment operations. An example of how to use the command in aligning data see Section 3.1.6, “Quad-Word Data Alignment.” Table 4-22 describes the vector permute instruction.

Table 4-22. Vector Permute Instruction

Name	Mnemonic	Syntax	Operation
Vector Permute	**vperm**	**v**D**, v**A**,v**B**,v**C	**v**C specifies which bytes from **v**A and **v**B are to be copied and placed into the byte elements in **v**D.

4.2.5.6 Vector Select Instruction

Data flow in the vector unit can be controlled without branching by using a vector compare and the vector select (**vsel**) instructions. In this use, the compare result vector is used directly as a mask operand to vector select instructions.The **vsel** instruction selects one field from one or the other of two source operands under control of its mask operand. Use of the TRUE/FALSE compare result vector with select in this manner produces a two instruction equivalent of conditional execution on a per-field basis. Table 4-23 describes the **vsel** instruction.

Table 4-23. Vector Select Instruction

Name	Mnemonic	Syntax	Operation
Vector Select	**vsel**	**v**D,**v**A,**v**B,**v**C	For each bit, compare the value in **v**C to the value 0b0 and if it equals 0b0 then load **v**D with **v**A's corresponding bit value otherwise compare the value in **v**C to the value 0b1 and if it equals 0b1 then load **v**D with **v**B's corresponding bit value.

4.2.5.7 Vector Shift Instructions

The vector shift instructions shift the contents of a vector register or of a pair of vector registers left or right by a specified number of bytes (**vslo**, **vsro**, **vsldoi**) or bits (**vsl**, **vsr**). Depending on the instruction, this shift count is specified either by low-order bits of a vector register or by an immediate field in the instruction. In the former case the low-order 7 bits of the shift count register give the shift count in bits ($0 \leq \text{count} \leq 127$). Of these 7 bits, the high-order 4 bits give the number of complete bytes by which to shift and are used by **vslo** and **vsro**; the low-order 3 bits give the number of remaining bits by which to shift and are used by **vsl** and **vsr**.

There are two methods of specifying an inter-element shift or rotate of two source vector registers, extracting 16 bytes as the result vector. There is also a method for shifting a single source vector register left or right by any number of bits.

Table 4-24 describes the various vector shift instructions.

Table 4-24. Vector Shift Instructions

Name	Mnemonic	Syntax	Operation
Vector Shift Left	**vsl**	**v**D,**v**A,**v**B	Shift **v**A left by the 3 lsbs of **v**B, and place the result into **v**D If **v**B value in invalid, the default result is boundely undefined
Vector Shift Left Double by Octet Immediate	**vsldoi**	**v**D,**v**A,**v**B,SH	Shift **v**B left by the 3 lsbs of SH value and then OR with **v**A, place the result is into **v**D If **v**B value in invalid, the default result is 0
Vector Shift Left by Octet	**vslo**	**v**D,**v**A,**v**B	Shift **v**A left by the 3 lsbs of **v**B, and place the result into **v**D If **v**B value in invalid, the default result is 0b000
Vector Shift Right by Octet	**vsro**	**v**D,**v**A,**v**B	Shift **v**A right by the 3 lsbs of **v**B, and place the result into **v**D If **v**B value in invalid, the default result is 0b000

4.2.5.7.1 Immediate Interelement Shifts/Rotates

The Vector Shift Left Double by Octet Immediate (**vsidoi**) instruction provides the basic mechanism that can be used to provide inter-element shifts and/or rotates. This instruction is like a **vperm**, except that the shift count is specified as a literal in the instruction rather than as a control vector in another vector register, as is required by **vperm**. The result vector consists of the left-most 16 bytes of the rotated 32-byte concatenation of **v**A:**v**B, where shift (SH) is the rotate count. Table 4-25 below enumerates how various shift functions can be achieved using the **vsidoi** instruction.

Table 4-25. Coding Various Shifts and Rotates with the vsidoi Instruction

To Get This:		Code This:			
Operation	**sh**	**Instruction**	**Immediate**	**vA**	**vB**
rotate left double	0–15	**vsidoi**	0–15	MSV	LSV
rotate left double	16–31	**vsidoi**	mod16(SH)	LSV	MSV
rotate right double	0–15	**vsidoi**	16–sh	MSV	LSV
rotate right double	16–31	**vsidoi**	16–mod16(SH)	LSV	MSV
shift left single, zero fill	0–15	**vsidoi**	0–15	MSV	0x0
shift right single, zero fill	0–15	**vsidoi**	16–SH	0x0	MSV
rotate left single	0–15	**vsidoi**	0–15	MSV	=VA
rotate right single	0–15	**vsidoi**	16–SH	MSV	=VA

4.2.5.7.2 Computed Interelement Shifts/Rotates

The Load Vector for Shift Left (**lvsl**) instruction and Load Vector for Shift Right (**lvsr**) instruction are supplied to assist in shifting and/or rotating vector registers by an amount determined at run time. The input specifications have the same form as the vector load and store instructions, that is, it uses register indirect with index addressing mode(**r**A|0 +**r**B). This is because one of their primary purposes is to compute the permute control vector necessary for post-load and pre-store shifting necessary for dealing with unaligned vectors.

This **lvsl** instruction can be used to align a big-endian unaligned vector after loading the (aligned) vectors that contain its pieces. The **lvsl** instruction can be used to unalign a vector register for use in a read-modify-write sequence that will store an unaligned little-endian vector.

The **lvsr** instruction can be used to align a little-endian unaligned vector after loading the (aligned) vectors that contain its pieces. The **lvsl** instruction can be used to unalign a vector register for use in a read-modify-write sequence that will store an unaligned big-endian vector.

For an example on how the **lvsl** instruction is used to align a vector in big-endian mode see Section 3.1.6.1, “Accessing a Misaligned Quad Word in Big-Endian Mode.” For an example on how **lvsr** is used to align a vector in little-endian mode see Section 3.1.6.2, “Accessing a Misaligned Quad Word in Little-Endian Mode.”

4.2.5.7.3 Variable Interelement Shifts

A vector register may be shifted left or right by a number of bits specified in a vector register. This operation is supported with four instructions, two for right shift and two for left shift.

The Vector Shift Left by Octet (**vslo**) and Vector Shift Right by Octet (**vsro**) instructions shift a vector register from 0 to 15 bytes as specified in bits 121–124 of another vector register. The Vector Shift Left (**vsl**) and Vector Shift Right (**vsr**) instructions shift a vector register from 0 to 7 bits as specified in another vector register (the shift count must be specified in the three lsbs of each byte in the vector and must be identical in all bytes or the result is boundedly undefined). In all of these instructions, zeros are shifted into vacated element and bit positions.

Used sequentially with the same shift-count vector register, these instructions will shift a vector register left or right from 0 to 127 bits as specified in bits 121–127 of the shift-count vector register. For example:

```
vslo      VZ, VX, VY
vspltb    VY, VY, 15
vsl       VZ, VZ, VY
```

will shift **v**X by the number of bits specified in **v**Y and place the results in **v**Z.

With these instructions a full double-register shift can be performed in seven instructions. The following code will shift **v**W||**v**X left by the number of bits specified in **v**Y placing the result in **v**Z:

```
vslo      t1, VW, VY    ; shift the most significant. register left
vspltb    VY, VY, 15
vsl       t1, t1, VY
vsububm   VY, V0, VY    ; adjust count for right shift (V0=0)
vsro      t2, VX, VY    ; right shift least sign. register
vsr       t2, t2, VY
vor       VZ, t1, t2    ; merge to get the final result
```

4.2.6 Processor Control Instructions—UISA

Processor control instructions are used to read from and write to the PowerPC condition register (CR), machine state register (MSR), and special-purpose registers (SPRs). See Chapter 4, “Addressing Mode and Instruction Set Summary,” in *PowerPC: The Programming Environments Manual*, for information about the instructions used for reading from and writing to the MSR and SPRs.

4.2.6.1 AltiVec Status and Control Register Instructions

Table 4-26 summarizes the instructions for reading from or writing to the AltiVec status and control register (VSCR). For more information on VSCR see section in Section 2.1.1, “The Vector Status and Control Register (VSCR).”

Table 4-26. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax	Operation
Move to AltiVec Status and Control Register	**mtvscr**	CRM,**r**S	Place the contents of **v**B into VSCR.
Move from AltiVec Status and Control Register	**mfvscr**	**v**B	Place the contents of VSCR into **v**B.

4.2.7 Recommended Simplified Mnemonics

To simplify assembly language programs, a set of simplified mnemonics is provided for some of the most frequently used operations (such as no-op, load immediate, load address, move register, and complement register). Assemblers should provide the simplified mnemonics listed below. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in this document.

Simplified mnemonics are provided for the Data Stream Touch (**dst**) and Data Stream Touch for Store (**dstst**) instructions so that they can be coded with the transient indicator as part of the mnemonic rather than as a numeric operand. Similarly, simplified mnemonics are provided for the Data Stream Stop (**dss)** instruction so that it can be coded with the all streams indicator as part of the mnemonic. These are shown as examples with the instructions in Table 4-27.

Table 4-27. Simplified Mnemonics for Data Stream Touch (dst)

Operation	Simplified Mnemonic	Equivalent to
Data Stream Touch (non-transient)	**dst r**A**, r**B**,** STRM	**dst r**A**, r**B**,** STRM,**0**
Data Stream Touch Transient	**dstt r**A**, r**B**,** STRM	**dst r**A**, r**B**,** STRM,1
Data Stream Touch for Store (non-transient)	**dstst r**A**, r**B**,** STRM	**dstst r**A**, r**B**,** STRM,0
Data Stream Touch for Transient	**dststt r**A**, r**B**,** STRM	**dststt r**A**, r**B**,** STRM,1
Data Stream Stop (one stream)	**dss** STRM	**dss** STRM,**0**
Data Stream Stop All	**dssall**	**dss 0**,**1**

4.3 AltiVec VEA Instructions

U V O The PowerPC virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache-control instructions, address aliasing, and other related issues. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA. For further details see Chapter 4, “Addressing Mode and Instruction Set Summary,” in *PowerPC: The Programming Environments Manual.*

This section describes the additional instructions that are provided by the AltiVec ISA for the VEA.

4.3.1 Memory Control Instructions—VEA

V O Memory control instructions include the following types:

- Cache management instructions (user-level and supervisor-level)
- Segment register manipulation instructions
- Segment lookaside buffer management instructions
- Translation lookaside buffer (TLB) management instructions

This section describes the user-level cache management instructions defined by the VEA. See Chapter 4, “Addressing Mode and Instruction Set Summary,” in *PowerPC: The Programming Environments Manual* for more information about supervisor-level cache, segment register manipulation, and TLB management instructions.

4.3.2 User-Level Cache Instructions—VEA

V The instructions summarized in this section provide user-level programs the ability to manage on-chip caches if they are implemented. See Chapter 5, “Cache Model and Memory Coherency,” in *PowerPC: The Programming Environments Manual* for more information about cache topics.

Bandwidth between the processor and memory is managed explicitly by the programmer through the use of cache management instructions. These instructions provide a way for software to communicate to the cache hardware how it should prefetch and prioritize writeback of data. The principal instruction for this purpose is a software directed cache prefetch instruction called Data Stream Touch (**dst**). Other related instructions are provided for complete control of the software directed cache prefetch mechanism.

Table 4-28 summarizes the directed prefetch cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs. V

Table 4-28. User-Level Cache Instructions

Name	Mnemonic	Syntax	Operation
Data Stream Touch	**dst**	**r**A,**r**B,STRM,T	This instruction associates the data stream specified by the contents of **r**A and **r**B with the stream ID specified by STRM. This instruction is a hint that performance will probably be improved if the cache blocks containing the specified data stream are fetched into the data cache, because the program will probably soon load from the stream, and that prefetching from any data stream that was previously associated with the specified stream ID is no longer needed. The hint is ignored for blocks that are Caching Inhibited. The specified data stream is defined by the following. EA: (**r**A), where **r**A ^= 0 unit size: (**r**B)[35–39 {3–7 for 32-bit implementations}] if (**r**B)[35-39 {3–7for 32-bit implementations)}] ^= 0; otherwise 32 count: (**r**B)[40–47 {8–15for 32-bit implementations}] if (**r**B)[40–47 {8–15for 32-bit implementations}] ^= 0; otherwise 256 stride: (**r**B)[48–63 {16–31for 32-bit implementations}] if (**r**B)[48–63 {16–31for 32-bit implementations}] ^= 0; otherwise 32768 The T bit of the instruction indicates whether the data stream is likely to be stored into fairly frequently in the near future (T=0) or to be transient (T=1). If **r**A=0, the instruction form is invalid.

Table 4-28. User-Level Cache Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Data Stream Touch	**dstt**	**r**A,**r**B,STRM,T	This instruction associates the data stream specified by the contents of registers **r**A and **r**B with the stream ID specified by STRM. This instruction is a hint that performance will probably be improved if the cache blocks containing the specified data stream are not fetched into the data cache, because the program will probably not load from the stream.That is, the data stream will be relatively transient in nature. That is, it will have poor locality and is likely to be referenced a very few times or over a very short period of time. The memory subsystem can use this persistent/transient knowledge to manage the data as is most appropriate for the specific design of the cache/memory hierarchy of the processor on which the program is executing. An implementation is free to ignore **dstt**, in that case it should simply be executed as a **dst.** However, software should always attempt to use the correct form of **dst** or **dstt** regardless of whether the intended processor implements **dstt** or not. In this way the program will automatically benefit when run on processors that do support **dstt**. The specified data stream is defined by the following. EA: (**r**A), where **r**A ^= 0 unit size: (**r**B)[35–39 {3–7 for 32-bit implementations}] if (**r**B)[35-39 {3–7for 32-bit implementations)}] ^= 0; otherwise 32 count: (**r**B)[40–47 {8–15for 32-bit implementations}] if (**r**B)[40–47 {8–15for 32-bit implementations}] ^= 0; otherwise 256 stride: (**r**B)[48–63 {16–31for 32-bit implementations}] if (**r**B)[48–63 {16–31for 32-bit implementations}] ^= 0; otherwise 32768 The T bit of the instruction indicates whether the data stream is likely to be accessed into fairly frequently in the near future (T=0) or to be transient (T=1). If **r**A=0, the instruction form is invalid.
Data Stream Touch for Store (non-transient)	**dstst**	**r**A,**r**B,STRM,T	This instruction associates the data stream specified by the contents of registers **r**A and **r**B with the stream ID specified by STRM. This instruction is a hint that performance will probably be improved if the cache blocks containing the specified data stream are fetched into the data cache, because the program will probably soon access into the stream, and that prefetching from any data stream that was previously associated with the specified stream ID is no longer needed. The hint is ignored for blocks that are caching inhibited. The specified data stream is defined by the following. EA: (**r**A), where **r**A ^= 0 unit size: (**r**B)[35–39 {3–7 for 32-bit implementations}] if (**r**B)[35-39 {3-7for 32-bit implementations)}] ^= 0; otherwise 32 count: (**r**B)[40–47 {8–15for 32-bit implementations}] if (**r**B)[40–47 {8–15for 32-bit implementations}] ^= 0; otherwise 256 stride: (**r**B)[48–63 {16–31for 32-bit implementations}] if (**r**B)[48–63 {16–31for 32-bit implementations}] ^= 0; otherwise 32768 The T bit of the instruction indicates whether the data stream is likely to be stored into fairly frequently in the near future (T=0) or to be transient (T=1). If **r**A=0, the instruction form is invalid

Table 4-28. User-Level Cache Instructions (Continued)

Name	Mnemonic	Syntax	Operation
Data Stream Touch for Store	**dststt**	**r**A,**r**B,STRM,T	This instruction associates the data stream specified by the contents of **r**A and **r**B with the stream ID specified by STRM. This instruction is a hint that performance will probably not be improved if the cache blocks containing the specified data stream are fetched into the data cache, because the program will probably not access the stream. That is, the data stream will be relatively transient in nature. That is, it will have poor locality and is likely to be referenced a very few times or over a very short period of time. The memory subsystem can use this persistent/transient knowledge to manage the data as is most appropriate for the specific design of the cache/memory hierarchy of the processor on which the program is executing. The specified data stream is defined by the following. EA: (**r**A), where **r**A ^= 0 unit size: (**r**B)[35–39 {3–7 for 32-bit implementations}] if (**r**B)[35-39 {3-7for 32-bit implementations)}] ^= 0; otherwise 32 count: (**r**B)[40–47 {8–15for 32-bit implementations}] if (**r**B)[40–47 {8–15for 32-bit implementations}] ^= 0; otherwise 256 stride: (**r**B)[48–63 {16–31for 32-bit implementations}] if (**r**B)[48–63 {16–31for 32-bit implementations}] ^= 0; otherwise 32768 The T bit of the instruction indicates whether the data stream is likely to be stored into fairly frequently in the near future (T=0) or to be transient (T=1). If **r**A=0, the instruction form is invalid
Data Stream Stop	**dss**	STRM,A	If A = 0 and a data stream associated with the stream ID specified by STRM exists, this instruction terminates prefetching of that data stream. If A = 1, this instruction terminates prefetching of all existing data streams. (The STRM field is ignored.) In addition, executing a **dss** instruction ensures that all memory accesses associated with data stream prefetching caused by preceding dst and dstst instructions that specified the same stream ID as that specified by the **dss** instruction (A = 0), or by all preceding dst and dstst instructions (A = 1), will be in group G1 with respect to the memory barrier created by a subsequent **sync** instruction. **dss** serves as both a basic and an extended mnemonic. The assembler will recognize a **dss** mnemonic with two operands as the basic form, and a **dss** mnemonic with one operand as the extended form. Execution of a **dss** instruction causes address translation for the specified data stream(s) to cease. Prefetch requests for which the effective address has already been translated may complete and may place the corresponding data into the data cache
Data Stream Stop All	**dssall**		Terminates prefetching of all existing data streams. All active streams may be stopped. If the optional data stream prefetch facility is implemented, **dssall** (extended mnemonic for **dss**), to terminate any data stream prefetching requested by the interrupted program, in order to avoid prefetching data in the wrong context, consuming memory bandwidth fetching data that are not likely to be needed by the other program, and interfering with data cache use by the other program. The **dssall** must be followed by a **sync**, and additional software synchronization may be required.

Chapter 5
Cache, Exceptions, and Memory Management

This chapter summarizes details of the AltiVec™ technology definition that pertain to cache and memory management models. Note that the AltiVec technology defines most of its instructions at the user-level (UISA). Because most AltiVec instructions are computational there is little effect on the VEA and OEA portions of the PowerPC architecture definition.

Because the AltiVec instruction set architecture (ISA) uses 128-bit operands, additional instructions are provided to optimize cache and memory bus use.

5.1 PowerPC Shared Memory

In order to fully understand the data stream prefect instructions for the AltiVec, one needs a knowledge on the PowerPC architecture for shared memory . The following provides updated details on the PowerPC architecture for shared memory.

The PowerPC architecture supports the sharing of memory between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to memory by one or more programs using different effective addresses. All these cases are considered memory sharing. Memory is shared in blocks that are an integral number of pages.

When the same memory has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

5.1.1 PowerPC Memory Access Ordering

The memory model for the ordering of memory accesses is weakly consistent. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when necessary for the correct execution of the program. The order in which the processor performs memory accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main memory may all be different.

Several means of enforcing an ordering of memory accesses are provided to allow programs to share memory with other programs, or with mechanisms such as I/O devices:

- If two Store instructions specify memory locations that are both caching inhibited and guarded, the corresponding memory accesses are performed in program order with respect to any processor or mechanism.
- If a load instruction depends on the value returned by a preceding load instruction (because the value is used to compute the effective address specified by the second load), the corresponding memory accesses are performed in program order with respect to any processor or mechanism to the extent required by the memory coherence required attributes associated with the access, if any. This applies even if the dependency has no effect on program logic (for example, the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load).
- When a processor (P1) executes a **sync** or **eieio** instruction a memory barrier is created, which separates applicable memory accesses into two groups, G1 and G2. G1 includes all applicable memory accesses associated with instructions preceding the barrier-creating instruction, and G2 includes all applicable memory accesses associated with instructions following the barrier-creating instruction. The memory barrier ensures that all memory accesses in G1 wi l l be performed with respect to any processor or mechanism, to the extent required by the memory coherence required attributes associated with the access, if any, before any memory accesses in G2 are performed with respect to that processor or mechanism.

The ordering done by a memory barrier is said to be “cumulative” if it also orders memory accesses that are performed by processors and mechanisms other than P1, as follows:

- G1 includes all applicable memory accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- G2 includes all applicable memory accesses by any such processor or mechanism that are performed after a load instruction executed by that processor or mechanism has returned the value accessed by a store that is in G2.

The memory barrier created by **sync** is cumulative, and applies to all memory accesses except those associated with fetching instructions following the **sync** instruction. See the description of **eieio** instruction in the *PowerPC Microprocessor Family: The Programming Environments Manual* for a description of the corresponding properties of the memory barrier created by that instruction.

No ordering should be assumed among the memory accesses caused by a single instruction (that is, by an instruction for which the access is not atomic), and no means are provided for controlling that order.

5.2 AltiVec Memory Bandwidth Management

The AltiVec ISA provides a way for software to speculatively load larger blocks of data from memory. That is, you can use bandwidth that would otherwise be idle which permits the software to take advantage of locality and reduces the number of system memory accesses.

5.2.1 Software-Directed Prefetch

Bandwidth between the processor and memory is managed explicitly by the programmer through use of cache management instructions. These instructions let software indicate to the cache hardware how to prefetch and prioritize writeback of data. The principle instruction for this purpose is a software-directed cache prefetch instruction, Data Stream Touch (**dst**), described in the following section.

5.2.1.1 Data Stream Touch (dst)

The data stream prefetch facility permits a program to indicate that a sequence of units of memory is likely to be accessed soon by memory access instructions. Such a sequence is called a data stream or, when the context is clear, simply a stream. A data stream is defined by the following:

- EA—The effective address of the first unit in the sequence
- Unit size—The number of quad words in each unit; 0 < unit size ≤ 32
- Count—The number of units in the sequence; 0 < count ≤ 256
- Stride—The number of bytes between the effective address of one unit in the sequence and the effective address of the next unit in the sequence (that is, the effective address of the nth unit in the sequence is EA + (n - 1) x stride); (-32768 ≤ stride < 0 or 0 < stride ≤ 32768)

The units need not be aligned on a particular memory boundary. The stride may be negative.

The **dst** instruction specifies a starting address, a block size (1–32 vectors), a number of blocks to prefetch (1–256 blocks), and a signed stride in bytes (-32,768 to +32,768 bytes), The 2-bit tag, specified as an immediate field in the opcode, identifies one of four possible touch streams. The starting address of the stream is specified in **r**A (if **r**A = 0, the instruction form is invalid). BlockSize, BlockCount, and BlockStride are specified in **r**B. Do not confuse the term ‘cache block’, the term ‘block’ always indicates a PowerPC cache block.

The format of the **r**B register is shown in Figure 5-1.

0 0 0	BlockSize	BlockCount	Signed BlockStride
0–2	3–7	8–15	16–31

Figure 5-1. Format of rB in dst Instruction

There is no zero-length block size, block count, or block stride. A BlockSize of 0 indicates 32 vectors, a BlockCount of 0 indicates 256 blocks, and a BlockStride of 0 indicates +32,768 bytes. Otherwise, these fields correspond to the numerical value of the size, count, and stride. Do not specify strides smaller than 1 block (16 bytes).

The programmer specifies block size in terms of vectors (16 bytes), regardless of the cache-block size. Hardware automatically optimizes the number of cache blocks it fetches to bring a block into the cache. The number of cache blocks fetched into the cache for each block is the fewest natural cache blocks needed to fetch the entire block, including the effects of block misalignment to cache blocks, as shown in the following:

$$\text{CacheBlocksFetched} = \text{ceiling}\left(\frac{\text{BlockSize} + \text{mod(BlockAddr,CacheBlockSize)}}{\text{CacheBlockSize}}\right)$$

The address of each block in a stream is a function of the stream's starting address, the block stride, and the block being fetched. The starting address may be any 32-bit byte address. Each block's address is computed as a full 32-bit byte address from the following:

$$\text{BlockAddr}_n = (\mathbf{r}\text{A}) + n\,(\mathbf{r}\text{B})_{16-31} \qquad \begin{array}{l}\text{where } n = \{0 \ldots (\text{BlockCount} - 1)\} \\ \text{and} \quad \text{if } ((\mathbf{r}\text{B})_{16-31} = 0) \text{ then } ((\mathbf{r}\text{B})_{16-31} \Leftarrow 32768)\end{array}$$

The address of the first cache block fetched in each block is that block's address aligned to the next lower natural cache-block boundary by ignoring $\log_2$(CacheBlockSize) least significant bits (lsbs) (for example, for 32-byte cache-blocks, the five lsbs are ignored). Cache blocks are then fetched sequentially forward until the entire block of vectors is brought into the cache. An example of a six-block data stream is shown in Figure 5-2

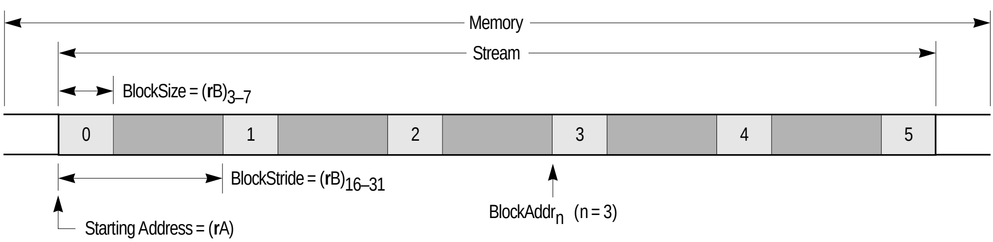


Figure 5-2. Data Stream Touch

Executing a **dst** instruction notifies the cache/memory subsystem that the program will soon need specified data. If bandwidth is available, the hardware starts loading the specified stream into the cache. To the extent that hardware can acquire the data, when the loads requiring the data finally execute, the target data will be in the cache. Executing a second **dst** to the tag of a stream in progress aborts the existing stream (at hardware's earliest convenience) and establishes a new stream with the same stream tag ID.

The **dst** instruction is a hint to hardware and has no architecturally visible effects (in the PowerPC UISA sense). The hardware is free to ignore it, to start the prefetch when it can, to abort the stream at any time, or to prioritize other memory operations ahead of it. If a stream is aborted, the program still functions properly, but subsequent loads experience the full latency of a cache miss.

The **dst** instruction does not introduce implementation problems like those of load/store multiple/string instructions. Because **dst** does not affect the architectural state, it does not cause interlock problems associated with load/store multiple/string instructions. Also, **dst** does take exceptions and requires no complex recovery mechanism.

Touch instructions should be considered strong hints. Using them in highly speculative situations could waste considerable bandwidth. Implementations that do not implement the stream mechanism treat stream instructions (**dst**, **dstt**, **dsts**, **dstst**, **dss**, and **dssall**) as no-ops. If the stream mechanism is implemented, all four streams must be provided.

5.2.1.2 Transient Streams (dstt)

The memory subsystem considers **dst** an indication that its stream data is likely to have some reasonable degree of locality and be referenced several times or over some reasonably long period. This is called persistence. The Data Stream Touch Transient instruction (**dstt**) indicates to the memory system that its stream data is transient, that is, it has poor locality and is likely to be used very few times or only for a very short time. A memory subsystem can use this knowledge to manage data for the processor’s cache/memory design. An implementation may ignore the distinction between transience and persistence, in that case **dstt** acts like **dst**. However, portable software should always use the correct form of **dst** or **dstt** regardless of whether the intended processor makes that distinction.

5.2.1.3 Storing to Streams (dstst)

A **dst** instruction brings a cache block into the cache subsystem in a state most efficient for subsequent reading of data from it (load). The companion instruction, Data Stream Touch for Store (**dstst**), brings the cache block into the cache subsystem in a state most efficient for subsequent writing to it (store). For example, in a MESI cache subsystem, a **dst** might bring a cache block in shared (S) state, whereas a **dstst** would bring the cache block in exclusive (E) state to avoid a subsequent demand-driven bus transaction to take ownership of the cache block so the store can proceed.

The **dstst** streams are the same physical streams as **dst** streams, that is, **dstst** stream tags are aliases of **dst** tags. If not implemented, **dstst** defaults to **dst**. If **dst** is not implemented, it is a no-op. The **dststt** instruction is a transient version of **dstst**.

Data stream prefetching of memory locations is not supported when bit 57 of the segment table entry or bit 0 of the segment register (SR) is set . If a **dst** or **dstst** instruction specifies a data stream containing these memory locations, results are undefined.

5.2.1.4 Stopping Streams (dss)

The **dst** instructions have a counterpart called Data Stream Stop (**dss**). A program can stop any given stream prefetch by executing **dss** with that stream's tag. This is useful when a program speculatively starts a stream prefetch but later determines that the instruction stream went the wrong way. The **dss** instruction can stop the stream so no more bandwidth is wasted. All active streams may be stopped by using **dssall**. This is useful when the operating system needs to stop all active streams (process switch) but does not know how many streams are in progress.

Because **dssall** does not specify the number of implemented streams, it should always be used instead of a sequence of **dss** instructions to stop all streams.

Neither **dss** nor **dssall** is execution synchronizing; the time between when a **dss** is issued and the stream stops is not specified. Therefore, when software must ensure that the stream is physically stopped before continuing (for example, before changing virtual memory mapping), a special sequence of synchronizing instructions is required. The sequence can differ for different situations, but the following sequence works in all contexts:

```
dssall          ; stop all streams
sync            ; insert a barrier in memory pipe
lwz     Rn,...  ; stick one more operation in memory pipe
cmpd    Rn,Rn   ;
bne-    *-4     ; make sure load data is back
isync           ; wait for all previous instructions to
                ; complete to ensure
                ; memory pipe is clear and nothing is
                ; pending in the old context
```

Data stream prefetching for a given stream is terminated by executing the appropriate **dss** instruction. The termination can be synchronized by executing a **sync** instruction after the **dss** instruction if the memory barrier created by **sync** orders all address translation effects of the subsequent context-altering instructions. Otherwise, data dependencies are also required. For example, the following instruction sequence terminates all data stream prefetching before altering the contents of an segment register (SR):

```
dssall                ; stop all data stream prefetching
sync                  ; order dssall before load
lwz     Ry,sr_y(Rx)   ; load new SR value
mtsr    y,Ry          ; alter SR y
```

The **mtsr** instruction cannot be executed until the **lwz** loads the SR value into **r**y. The memory access caused by the **lwz** cannot be performed until the **dssall** instruction takes effect (that is, until address translation stops for all data streams and all memory accesses associated with data stream prefetches for which the effective address was translated before the translation stopped are performed).

5.2.1.5 Exception Behavior of Prefetch Streams

In general, exceptions do not cancel streams. Streams are sensitive to whether the processor is in user or supervisor mode (determined by MSR[PR]) and whether data address translation is used (determined by MSR[DR]). This allows prefetch streams to behave predictably when an exception occurs.

Streams are suspended in real addressing mode (MSR[DR] = 0) and remain suspended until translation is turned back on (MSR[DR] is set). A **dst** instruction issued while data translation is off (MSR[DR] = 0) produces boundedly-undefined results.

A stream is suspended whenever the MSR[PR] is different than it was when the **dst** that established it was issued. For example, if a **dst** is issued in user mode (MSR[PR] = 1), the resulting stream is suspended when the processor enters supervisor mode (MSR[PR] = 0) and remains suspended until the processor returns to user mode. Conversely, if the **dst** were issued in supervisor mode, it is suspended if the machine enters user mode.

Because exceptions do not cancel streams automatically, the operating system must stop streams explicitly when warranted, for example when switching processes or changing virtual memory context. Care must be taken if data stream prefetching is used in supervisor-level state (MSR[PR] = 0).

After an exception, the supervisor-level program that next changes MSR[DR] from 0 to 1 cause data-stream prefetching to resume for any data streams for which the corresponding **dst** or **dstst** instruction was executed in supervisor mode; such streams are called supervisor-level data streams. This program is unlikely to be the one that executed the corresponding **dst** or **dstst** instruction and is unlikely to use the same address translation context as that in which the **dst** or **dsts**t was executed. (Suspension and resumption of data stream prefetching work more naturally for user level data streams, because the next application program to be dispatched after an exception occurs is likely to be the most recently interrupted program.) Thus, an exception handler that changes the context in which data addresses are translated may need to terminate data-stream prefetching for supervisor-level data streams and to synchronize the termination before changing MSR[DR] to 1.

Although terminating all data stream prefetching in this case would satisfy the requirements of the architecture, doing so would adversely affect the performance of applications that use data-stream prefetching. Thus, it may be better for the operating system to record stream IDs associated with any supervisor-level data streams and to terminate prefetching for those streams only.

Cache affects of supervisor-level data-stream prefetching can also adversely affect performance of applications that use data stream prefetching, as supervisor-level use of the associated stream ID can take over an applications' data stream.

Data stream instructions cannot cause exceptions directly. Therefore, any event that would cause an exception on a normal load or store, such as a page fault or protection violation, is instead aborted and ignored.

Suspension or termination of data stream prefetching for a given data stream need not cancel prefetch requests for that data stream for which the effective address has been translated and need not cause data returned by such requests to be discarded. However, to improve software's ability to pace data stream prefetching with data consumption, it may be better to limit the number of these pending requests that can exist simultaneously.

5.2.1.6 Synchronization Behavior of Streams

Streams are not affected (stopped or suspended) by execution of any PowerPC synchronization instructions (**sync**, **isync**, or **eieio**). This permits these instructions to be used for synchronizing multiple processors without disturbing background prefetch streams. Prefetch streams have no architecturally observable effects and are not affected by synchronization instructions. Synchronizing the termination of data stream prefetching is needed only by the operating system

5.2.1.7 Address Translation for Streams

Like **dcbt** and **dcbtst** instructions, **dst**, **dstst**, **dstt**, and **dststt** are treated as loads with respect to address translation, memory protection, and reference and change recording.

Unlike **dcbt** and **dcbtst** instructions, stream instructions that cause a TLB miss cause a page table search and the page descriptor to be loaded into the TLB. Conceptually, address translation and protection checking is performed on every cache-block access in the stream and proceeds normally across page boundaries and TLB misses, terminating only on page faults or protection violations that cause a DSI exception.

Stream instructions operate like normal PowerPC cache instructions (such as **dcbt**) with respect to guarded memory; they are not subject to normal restrictions against prefetching in guarded space because they are program directed. However, speculative **dst** instructions can not start a prefetch stream to guarded space.

If the effective address of a cache block within a data stream cannot be translated, or if loading from the block would violate memory protection, the processor will terminate prefetching of that stream. (Continuing to prefetch subsequent cache blocks within the stream might cause prefetching to get too far ahead of consumption of prefetched data.) If the effective address can be translated, a TLB miss can cause such termination, even on implementations for which TLBs are reloaded in software.

5.2.1.8 Stream Usage Notes

A given data stream exists if a **dst** or **dstst** instruction has been executed that specifies the stream and prefetching of the stream has neither completed, terminated, or been supplanted. Prefetching of the stream has completed, when all the memory locations within the stream that will ever be prefetched as a result of executing the **dst** or **dstst** instruction have been prefetched (for example, locations for which the effective address cannot be translated will never be prefetched). Prefetching of the stream is terminated by executing the appropriate **dss** instruction; it is supplanted by executing another **dst** or **dstst** instruction that specifies the stream ID associated with the given stream. Because there are four stream IDs, as many as four data streams may exist simultaneously.

The maximum block count of **dst** is small because of its preferred usage. It is not intended for a single **dst** instruction to prefetch an entire data stream. Instead, **dst** instructions should be issued periodically, for example on each loop iteration, for the following reasons:

- Short, frequent **dst** instructions better synchronize the stream with consumption.
- With prefetch closely synchronized just ahead of consumption, another activity is less likely to inadvertently evict prefetched data from the cache before it is needed.
- The prefetch stream is restarted automatically after an exception (that could have caused the stream to be terminated by the operating system) with no additional complex hardware mechanisms needed to restart the prefetch stream.

Issuing new **dst** instructions to stream tag IDs in progress terminates old streams—**dst** instructions cannot be queued.

For example, when multiple **dst** instructions are used to prefetch a large stream, it would be poor strategy to issue a second **dst** whose stream begins at the specified end of the first stream before it was certain that the first stream had completed. This could terminate the first stream prematurely, leaving much of the stream unprefetched.

Paradoxically, it would also be unwise to wait for the first stream to complete before issuing the second **dst**. Detecting completion of the first stream is not possible, so the program would have to introduce a pessimistic waiting period before restarting the stream and then incur the full start-up latency of the second stream.

The correct strategy is to issue the second **dst** well before the anticipated completion of the first stream and begin it at an address overlapping the first stream by an amount sufficient to cover any portion of the first stream that could not yet have been prefetched. Issuing the second **dst** too early is not a concern because blocks prefetched by the first stream hit in the cache and need not be refetched. Thus, even if issued prematurely and overlapped excessively, the second **dst** rapidly advances to the point of prefetching new blocks. This strategy allows a smooth transition from the first stream to the second without significant breaks in the prefetch stream.

For the greatest performance benefit from data-stream prefetching, use the **dst** and **dstst** (and **dss**) instructions so that the prefetched data is used soon after it is available in the data cache. Pacing data stream prefetching with consumption increases the likelihood that prefetched data is not displaced from the cache before it is used, and reduces the likelihood that prefetched data displaces other data needed by the program.

Specifying each logical data stream as a sequence of shorter data streams helps achieve the desired pacing, even in the presence of exceptions, and address translation failures. The components of a given logical data stream should have the following attributes:

- The same stream ID should be associated with each component.
- The components should partially overlap (that is, the first part of a component should consist of the same memory locations as the last part of the preceding component).
- The memory locations which do not overlap with the next component should be large enough that a substantial portion of the component is prefetched. That is, prefetch enough memory locations for the current component before it is taken over by the prefetching being done for the next component.

5.2.1.9 Stream Implementation Assumptions

Some processors can treat **dst** instructions as no-ops. However, if a processor implements **dst**, a minimum level of functionality will be provided to create as consistent a programming model across different machines as possible. Programs can assume the functionality in a dst instruction:

- Implements all four tagged streams
- Implements each tagged stream as a separate, independent stream with arbitration for memory access performed on a round-robin basis.
- Searches the table for each stream access that misses in the TLB.
- Does not abort streams on page boundary crossings
- Does not abort streams on exceptions (except DSI exceptions caused by the stream).
- Does not abort streams, or hold up execution pending completion of streams, on the PowerPC synchronization instructions **sync**, **isync**, or **eieio**.
- Does not abort streams on TLB misses that occur on loads or stores issued concurrently with running streams. However, a DSI exception from one of those loads or stores may cause streams to abort.

5.2.2 Prioritizing Cache Block Replacement

Load Vector Indexed LRU (**lvxl**) and Store Vector Indexed LRU (**stvxl**) instructions provide explicit control over cache block replacement by letting the programmer indicate whether an access is likely to be the last reference made to the cache block containing this load or store. The cache hardware can then prioritize replacement of this cache block over others with older but more useful data.

Data accessed by a normal load or store is likely to be needed more than once. Marking this data as most-recently used (MRU) indicates that it should be a low-priority candidate for replacement. However, some data, such as that used in DSP multimedia algorithms, is rarely reused and should be marked as the highest priority candidate for replacement.

Normal accesses mark data MRU. Data unlikely to be reused can be marked LRU. For example, on replacing a cache block marked LRU by one of these instructions, a processor may improve cache performance by evicting the cache block without storing it in intermediate levels of the cache hierarchy (except to maintain cache consistency).

5.2.3 Partially Executed AltiVec Instructions

The OEA permits certain instructions to be partially executed when an alignment or DSI exception occurs. In the same way that the target register may be altered when floating-point load instructions cause a DSI exception, if the AltiVec facility is implemented, the target register (**v**D) may be altered when **lvx** or **lvxl** is executed and the TLB entry is invalidated before the access completes.

Exceptions cause data stream prefetching to be suspended for all existing data streams. Prefetching for a given data stream resumes when control is returned to the interrupted program, if the stream still exists (for example, the operating system did not terminate prefetching for the stream).

5.3 DSI Exception—Data Address Breakpoint

A data address breakpoint register (DABR) match causes a DSI exception in implementations that support the data breakpoint feature. When a DABR match occurs on a non-AltiVec PowerPC processor, the DAR is set to any effective address between and including the word (for a byte, half word, or word access) or double word (for a double-word access) specified by the effective address computed by the instruction and the effective address of the last byte in the word or double word in which the match occurred. In processors that support AltiVec technology, this would include a quad-word access from an **lvx**, **lvxl**, **stvx**, or **stvxl** instruction to a segment or BAT area.

5.4 AltiVec Unavailable Exception (0x00F20)

The AltiVec facility includes an additional instruction-caused, precise exception to those defined by the OEA and discussed in Chapter 6, “Exceptions,” in the PowerPC *Programming Environments Manual*. An AltiVec unavailable exception occurs when no higher priority exception exists (see Table 5-2), an attempt is made to execute an AltiVec instruction, and MSR[VEC] = 0.

Register settings for AltiVec unavailable exceptions are described in Table 5-1.

Table 5-1. AltiVec Unavailable Exception—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the exception
SRR1	64-Bit / 32-Bit 0–32 / 0 — Loaded with equivalent bits from the MSR 33–36 / 1–4 — Cleared 37–41 / 5–9 — Loaded with equivalent bits from the MSR 42–47 / 10–15 — Cleared 48–63 / 16–31 — Loaded with equivalent bits from the MSR Note that depending on the implementation, additional bits in the MSR may be copied to SRR1.
MSR	SF [1] 1; ISF [1] —; VEC 0; POW 0; ILE — EE 0; PR 0; FP 0; ME —; FE0 0 SE 0; BE 0; FE1 0; IP —; IR 0 DR 0; RI 0; LE Set to value of ILE

[1] 64-bit implementations only

When an AltiVec unavailable exception is taken, instruction execution resumes as offset 0x00F20 from the base address determined by MSR[IP].

The **dst** and **dstst** instructions are supported if MSR[DR] = 1. If either instruction is executed when MSR[DR] = 0 (real addressing mode), results are boundedly undefined.

Conditions that cause this exception are prioritized among instruction-caused (synchronous), precise exceptions as shown in Table 5-2 (taken from the section “Exception Priorities,” in Chapter 6, “Exceptions,” in *PowerPC: The Programming Environments Manual.*

Table 5-2. Exception Priorities (Synchronous/Precise Exceptions)

Priority	Exception
3 [1]	Instruction dependent—When an instruction causes an exception, the exception mechanism waits for any instructions prior to the excepting instruction in the instruction stream to complete. Any exceptions caused by these instructions are handled first. It then generates the appropriate exception if no higher priority exception exists when the exception is to be generated. Note that a single instruction can cause multiple exceptions. When this occurs, those exceptions are ordered in priority as indicated in the following: A. Integer loads and stores a. Alignment b. DSI c. Trace (if implemented) B. Floating-point loads and stores a. Floating-point unavailable b. Alignment c. DSI d. Trace (if implemented) C. Other floating-point instructions a. Floating-point unavailable b. Program—Precise-mode floating-point enabled exception c. Floating-point assist (if implemented) d. Trace (if implemented) D. AltiVec Loads and Stores (if AltiVec facility implemented) a. AltiVec unavailable b. DSI c. Trace (if implemented) E. Other AltiVec Instructions (if AltiVec facility implemented) a. AltiVec unavailable b. Trace (if implemented) F. The **rfid** (or **rfi**) and **mtmsrd** (or **mtmsr**) a. Program—Supervisor level Instruction b. Program—Precise-mode floating-point enabled exception c. Trace (if implemented), for **mtmsrd** (or **mtmsr**) only If precise-mode IEEE floating-point enabled exceptions are enabled and the FPSCR[FEX] bit is set, a program exception occurs no later than the next synchronizing event. G. Other instructions a. These exceptions are mutually exclusive and have the same priority: — Program: Trap — System call (**sc**) — Program: Supervisor level instruction — Program: Illegal Instruction b. Trace (if implemented) F. ISI exception The ISI exception has the lowest priority in this category. It is only recognized when all instructions prior to the instruction causing this exception appear to have completed and that instruction is to be executed. The priority of this exception is specified for completeness and to ensure that it is not given more favorable treatment. An implementation can treat this exception as though it had a lower priority.

[1] The exceptions are third in priority after system reset and machine check exceptions

Chapter 6
AltiVec Instructions

This chapter lists the AltiVec instruction set in alphabetical order by mnemonic. Note that each entry includes the instruction format and a graphical representation of the instruction. All the instructions are 32 bit and a description of the instruction fields and pseudocode conventions are also provided. For more information on the AltiVec instruction set, refer to Chapter 4 “Addressing Modes and Instruction Set Summary,” for more information on the PowerPC instruction set, refer to Chapter 8, “Instruction Set,” in *The PowerPC Microprocessor Family: The Programming Environments Manual*.

6.1 Instruction Formats

AltiVec instructions are four bytes (32 bits) long and are word-aligned. AltiVec instruction set architecture (ISA) has 4 operands, three source vectors and one result vector. Bits 0–5 always specify the primary opcode for AltiVec instructions. AltiVec ALU type instructions specify the primary opcode point 4(0b000100). AltiVec load, store, and stream prefetch instructions use secondary opcode in primary opcode 31 (0b011111).

Within a vector register, a byte, half-word, or word element, are referred to as follows:

- Byte elements, each byte = 8 bits, so in the pseudocode, n = 8 and there would be a total of 16 elements
- Half-word elements, each byte = 16 bits, so in the pseudocode, n = 16 and there would be a total of 8 elements
- Word elements, each byte = 32 bits, so in the pseudocode, n = 32 and there would be a total of 4 elements

Refer to Figure 1-3, for an example of how elements are placed in a vector register.

6.1.1 Instruction Fields

Table 6-1 describes the instruction fields used in the various instruction formats.

Table 6-1. Instruction Syntax Conventions

Field	Description
OPCD (0–5)	Primary opcode field
rA, A(11–15)	Specifies a GPR to be used as a source or destination.
rB, B(16–20)	Specifies a GPR to be used as a source.
Rc (31)	Record bit. 0 Does not update the condition register (CR). 1 For the optional AltiVec facility, set CR field 6 to control program flow as described in Section 2.1.3, “PowerPC Condition Register”
vA (11–15)	Specifies a vector register to be used as a source
vB (16–20)	Specifies a vector register to be used as a source.
vC (21–25)	Specifies a vector register to be used as a source.
vD (6–10)	Specifies a vector register to be used as a destination.
vS (6–10)	Specifies a vector register to be used as a source.
SHB (22–25)	Specifies a shift amount in bytes.
SIMM (11–15)	This immediate field is used to specify a (5 bit) signed integer.
UIMM (11–15)	This immediate field is used to specify a 4-,8-,12-, or 16-bit unsigned integer.
XO	Extended Opcode Field.

6.1.2 Notation and Conventions

The operation of some instructions is described by a semiformal language (pseudocode). See Table 6-2 for a list of additional pseudocode notation and conventions used throughout this section.

Table 6-2. Notation and Conventions

Notation/Convention	Meaning
←	Assignment
¬	NOT logical operator
do i=X to Y by Z	Do the following starting at X and iterating to Y by Z
$+_{int}$	2’s complement integer add
$-_{int}$	2’s complement integer subtract
$\mathbf{+_{ui}}$	Unsigned integer add
$-_{ui}$	Unsigned integer subtract
$*_{ui}$	Unsigned integer multiply
$+_{si}$	Signed integer add
$-_{si}$	Signed integer subtract

Table 6-2. Notation and Conventions (Continued)

Notation/Convention	Meaning
$*_{si}$	Signed integer multiply
$*_{sui}$	Signed integer (first operand) multiplied by unsigned integer (second operand) producing signed result
/	Integer divide
$+_{fp}$	Single-precision floating-point add
$-_{fp}$	Single-precision floating-point subtract
$*_{fp}$	Single-precision floating-point multiply
$\div_{fp}$	Single-precision floating-point divide
$\sqrt{}_{fp}$	Single-precision floating-point square root
$<_{ui}, \leq_{ui}, >_{ui}, \geq_{ui}$	Unsigned integer comparison relations
$<_{si}, \leq_{si}, >_{si}, \geq_{si}$	Signed integer comparison relations
$<_{fp}, \leq_{fp}, >_{fp}, \geq_{fp}$	Single precision floating point comparison relations
$\neq$	Not equal
$=_{int}$	Integer equal to
$=_{ui}$	Unsigned integer equal to
$=_{si}$	Signed integer equal to
$=_{fp}$	Floating-point equal to
$X >>_{ui} Y$	Shift X right by Y bits extending Xs vacated bits with zeros
$X >>_{si} Y$	Shift X right by Y bits extending Xs vacated bits with the sign bit of X
$X <<_{ui} Y$	Shift X left by Y bits inserting Xs vacated bits with zeros
||	Used to describe the concatenation of two values (that is, 010 || 111 is the same as 010111)
&	AND logical operator
|	OR logical operator
$\oplus$, $\equiv$	Exclusive-OR, Equivalence logical operators (for example, $(a \equiv b) = (a \oplus \neg b)$)
0b*nnnn*	A number expressed in binary format.
0x*nnnn*	A number expressed in hexadecimal format.
?	Unordered comparison relation
${}^{X}0$	X zeros
${}^{X}1$	X ones
${}^{X}Y$	X copies of Y
X_Y	bit Y of X
$X_{Y:Z}$	bits Y through Z, inclusive, of X
LENGTH(x)	Length of x, in bits. If x is the word “element”, LENGTH(x) is the length, in bits, of the element implied by the instruction mnemonic.

Table 6-2. Notation and Conventions (Continued)

Notation/Convention	Meaning
ROTL(x,y)	Result of rotating x left by y bits
UItoUImod(X,Y)	Chop unsigned integer X- to Y-bit unsigned integer
UItoUIsat(X,Y)	Result of converting the unsigned-integer x to a y-bit unsigned-integer with unsigned-integer saturation
SItoUIsat(X,Y)	Result of converting the signed-integer x to a y-bit unsigned-integer with unsigned-integer saturation
SItoSImod(X,Y)	Chop integer X- to Y-bit integer
SItoSIsat(X,Y)	Result of converting the signed-integer x to a y-bit signed-integer with signed-integer saturation
RndToNearFP32	The single-precision floating-point number that is nearest in value to the infinitely-precise floating-point intermediate result x (in case of a tie, the even single-precision floating-point value is used).
RndToFPInt32Near	The value x if x is a single-precision floating-point integer; otherwise the single-precision floating-point integer that is nearest in value to x (in case of a tie, the even single-precision floating-point integer is used).
RndToFPInt32Trunc	The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x if x>0, or the smallest single-precision floating-point integer that is greater than x if x<0
RndToFPInt32Ceil	The value x if x is a single-precision floating-point integer; otherwise the smallest single-precision floating-point integer that is greater than x
RndToFPInt32Floor	The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x
CnvtFP32ToUI32Sat(x)	Result of converting the single-precision floating-point value x to a 32-bit unsigned-integer with unsigned-integer saturation
CnvtFP32ToSI32Sat(x)	Result of converting the single-precision floating-point value x to a 32-bit signed-integer with signed-integer saturation
CnvtUI32ToFP32(x)	Result of converting the 32-bit unsigned-integer x to floating-point single format
CnvtSI32ToFP32(x)	Result of converting the 32-bit signed-integer x to floating-point single format
MEM(X,Y)	Value at memory location X of size Y bytes
SwapDouble	Swap the doublewords in a quadword vector
ZeroExtend(X,Y)	Zero-extend X on the left with zeros to produce Y-bit value
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
RotateLeft(X,Y)	Rotate X left by Y bits
mod(X,Y)	Remainder of X/Y
UImaximum(X,Y)	Maximum of 2 unsigned integer values, X and Y
SImaximum(X,Y)	Maximum of 2 unsigned integer values, X and Y
FPmaximum(X,Y)	Maximum of 2 floating-point values, X and Y
UIminimum(X,Y)	Minimum of 2 unsigned integer values, X and Y
SIminimum(X,Y)	Minimum of 2 unsigned integer values, X and Y

Table 6-2. Notation and Conventions (Continued)

Notation/Convention	Meaning
FPminimum(X,Y)	Minimum of 2 floating-point values, X and Y
FPReciprocalEstimate12(X)	12-bit-accurate floating-point estimate of 1/X
FPReciprocalSQRTEstimate12(X)	12-bit-accurate floating-point estimate of 1/(sqrt(X))
FPLog$_2$Estimate3(X)	3-bit-accurate floating-point estimate of log2(X)
FPPower2Estimate3(X)	3-bit-accurate floating-point estimate of 2**X
CarryOut(X + Y)	Carry out of the sum of X and Y
ROTL[64](x, y)	Result of rotating the 64-bit value x left y positions
ROTL[32](x, y)	Result of rotating the 32-bit value x || x left y positions, where x is 32 bits long
0b*nnnn*	A number expressed in binary format.
0x*nnnn*	A number expressed in hexadecimal format.
(*n*)x	The replication of x, *n* times (that is, x concatenated to itself *n* – 1 times). (*n*)0 and (*n*)1 are special cases. A description of the special cases follows: • (*n*)0 means a field of *n* bits with each bit equal to 0. Thus (5)0 is equivalent to 0b00000. • (*n*)1 means a field of *n* bits with each bit equal to 1. Thus (5)1 is equivalent to 0b11111.
(**r**A|0)	The contents of **r**A if the **r**A field has the value 1–31, or the value 0 if the **r**A field is 0.
(**r**X)	The contents of **r**X
x[*n*]	*n* is a bit or field within x, where x is a register
x^n	x is raised to the *n*th power
ABS(x)	Absolute value of x
CEIL(x)	Least integer ≥ x
Characterization	Reference to the setting of status bits in a standard way that is explained in the text.
CIA	Current instruction address. The 64- or 32-bit address of the instruction being described by a sequence of pseudocode. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK = 1 to set the link register. Does not correspond to any architected register.
Clear	Clear the leftmost or rightmost *n* bits of a register to 0. This operation is used for rotate and shift instructions.
Clear left and shift left	Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a known non-negative array index by the width of an element. These operations are used for rotate and shift instructions.
Cleared	Bits = 0.
Do	Do loop. • Indenting shows range. • “To” and/or “by” clauses specify incrementing an iteration variable. • “While” clauses give termination conditions.

Table 6-2. Notation and Conventions (Continued)

Notation/Convention	Meaning
DOUBLE(x)	Result of converting x from floating-point single-precision format to floating-point double-precision format.
Extract	Select a field of *n* bits starting at bit position *b* in the source register, right or left justify this field in the target register, and clear all other bits of the target register to zero. This operation is used for rotate and shift instructions.
EXTS(x)	Result of extending x on the left with sign bits
GPR(x)	General-purpose register x
if...then...else...	Conditional execution, indenting shows range, else is optional.
Insert	Select a field of *n* bits in the source register, insert this field starting at bit position *b* of the target register, and leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a field when operating on double words; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)
Leave	Leave innermost do loop, or the do loop described in leave statement.
MASK(x, y)	Mask having ones in positions x through y (wrapping if x > y) and zeros elsewhere.
MEM(x, y)	Contents of y bytes of memory starting at address x.
NIA	Next instruction address, which is the 64- or 32-bit address of the next instruction to be executed (the branch destination) after a successful branch. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions which do not branch, the next instruction address is CIA + 4. Does not correspond to any architected register.
OEA	PowerPC operating environment architecture
Rotate	Rotate the contents of a register right or left *n* bits without masking. This operation is used for rotate and shift instructions.
ROTL[64](x, y)	Result of rotating the 64-bit value x left y positions
ROTL[32](x, y)	Result of rotating the 64-bit value x || x left y positions, where x is 32 bits long
Set	Bits are set to 1.
Shift	Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift). This operation is used for rotate and shift instructions.
SINGLE(x)	Result of converting x from floating-point double-precision format to floating-point single-precision format.
SPR(x)	Special-purpose register x
TRAP	Invoke the system trap handler.
Undefined	An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.
UISA	PowerPC user instruction set architecture
VEA	PowerPC virtual environment architecture

Table 6-3 describes instruction field notation conventions used throughout this chapter.

Table 6-3. Instruction Field Conventions

The PowerPC Architecture Specification	Equivalent in AltiVec Technology Specification as:
D	d
DS	ds
FLM	FM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S
RA, RB, RT, RS	A, B, D, S
SI	SIMM
U	IMM
UI	UIMM
VA, VB, VC, VT, VS	**v**A, **v**B, **v**C, **v**D, **v**S
/, //, ///	0...0 (shaded)

Precedence rules for pseudocode operators are summarized in Table 6-4.

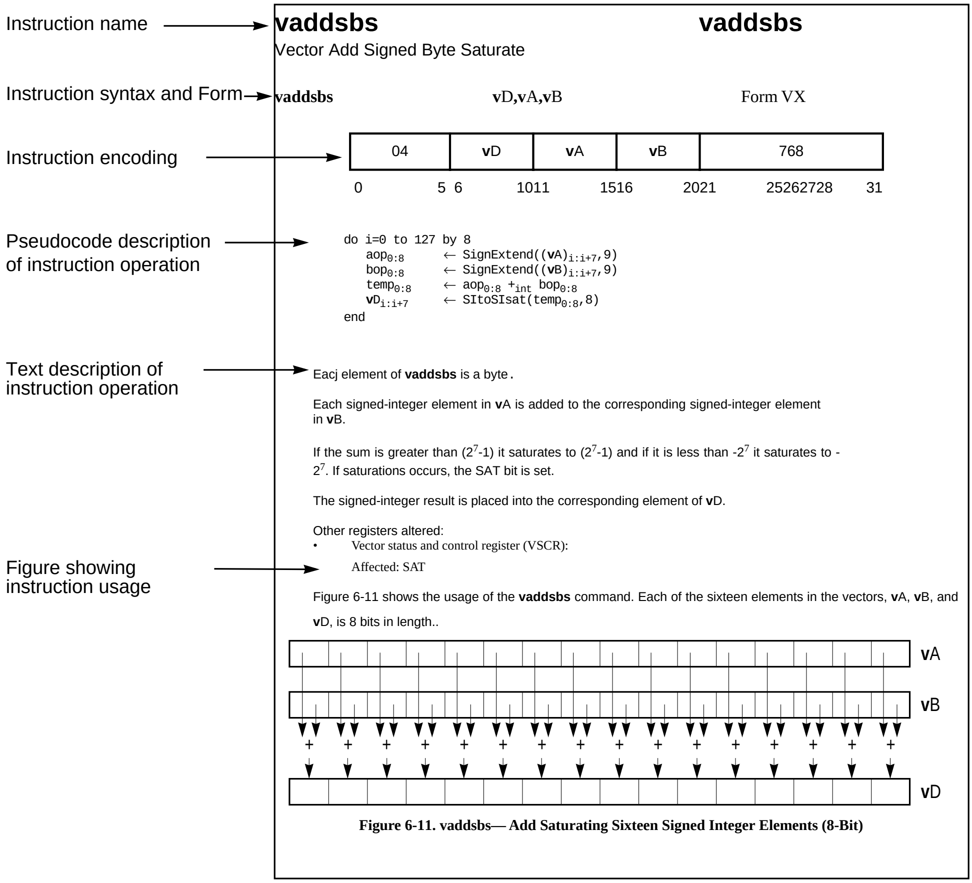
Table 6-4. Precedence Rules

Operators	Associativity
x[*n*], function evaluation	Left to right
(*n*)x or replication, x(*n*) or exponentiation	Right to left
unary –, ¬	Right to left
∗, ÷	Left to right
+, -	Left to right
||	Left to right
=, ≠, <, ≤, >, ≥, <U, >U, ?	Left to right
&, ⊕, ≡	Left to right
|	Left to right
– (range), : (range)	None
$\leftarrow$, $\leftarrow_{iea}$	None

Operators higher in Table 6-4 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 6-4, or to increase clarity; parenthesized expressions are evaluated before serving as operands.

6.2 AltiVec Instruction Set

The remainder of this chapter lists and describes the instruction set for the AltiVec architecture. The instructions are listed in alphabetical order by mnemonic. The diagram below shows the format for each instruction description page.



dss dss

Data Stream Stop

dss	STRM	(A=0)	Form X
dssall	STRM	(A=1)	

31	A	0 0	STRM	0 0 0 0 0	0 0 0 0 0	822	0
0 5	6	7 8	9 10	11 15	16 20	21 30	31

```
DataStreamPrefetchControl ← “stop” ‖ STRM
```

Note that A does not represent **r**A in this instruction.

If A=0 and a data stream associated with the stream ID specified by **STRM** exists, this instruction terminates prefetching of that data stream. It has no effect if the specified stream does not exist.

If A=1, this instruction terminates prefetching of all existing data streams (the STRM field is ignored.)

In addition, executing a **dss** instruction ensures that all accesses associated with data stream prefetching caused by preceding dst and dstst instructions that specified the same stream ID as that specified by the **dss** instruction (A=0), or by all preceding **dst** and **dstst** instructions (A=1), will be in group G1 with respect to the memory barrier created by a subsequent **sync** instruction, refer to Section 5.1.1, “PowerPC Memory Access Ordering,” for more information.

See Section 5.2.1, “Software-Directed Prefetch” for more information on using the **dss** instruction.

Other registers altered:

- None

Simplified mnemonics:

dss STRM	equivalent to	**dss**	**STRM**, **0**
dssall	equivalent to	**dss**	**0**, **1**

For more information on the **dss** instruction, refer to Chapter 5, “Cache, Exceptions, and Memory Management.”

dst — dst

Data Stream Touch

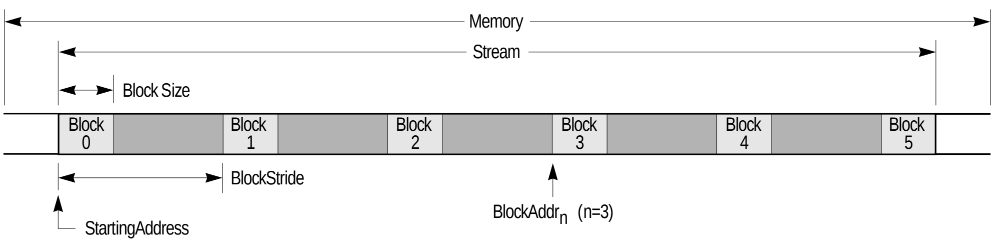
dst	**r**A,**r**B,STRM	(T=0)	Form X
dstt	**r**A,**r**B,STRM	(T=1)	

31	T	0 0	STRM	A	B	342	0
0–5	6	7–8	9–10	11–15	16–20	21–30	31

```
addr0:63 ← (rA)
DataStreamPrefetchControl ← "start" ‖ STRM ‖ T ‖ (rB) ‖ addr
```

This instruction initiates a software directed cache prefetch. The instruction is a hint to hardware that performance will probably be improved if the cache blocks containing the specified data stream are fetched into the data cache because the program will probably soon load from the stream.

The instruction associates the data stream specified by the contents of **r**A and **r**B with the stream ID specified by **STRM**. The instruction defines a data stream **STRM** as starting at an "Effective Address" (**r**A) and having "Count" units of "Size" bytes separated by "Stride" bytes (as specified in **r**B). The **T** bit of the instruction indicates whether the data stream is likely to be loaded from fairly frequently in the near future (**T** = 0) or to be transient and referenced very few times (**T** = 1).



The **dst** instruction does the following:

- Defines the characteristics of a data stream **STRM** by the contents of **r**A and **r**B
- Associates the stream with a specified stream ID, **STRM** (Range for STRM is 0-3)
- Indicates that the data in the specified stream **STRM** starting at the address in **r**A may soon be loaded
- Indicates whether memory locations within the stream are likely to be needed over a longer period of time (**T**=0) or be treated as transient data (**T**=1)
- Terminates prefetching from any stream that was previously associated with the specified stream ID, **STRM**.

The specified data stream is encoded for 32-bit as:

- Effective Address: **r**A, where **r**A ≠ 0
- Block Size: **r**B[3-7] if **r**B[3-7] ≠ 0; otherwise 32
- Block Count: **r**B[8-15] if **r**B[8-15] ≠ 0; otherwise 256
- Block Stride: **r**B[16-31] if **r**B[16-31] ≠ 0; otherwise 32768

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

Figure 6-1. Format of rB in dst instruction (32-bit)

The specified data stream is encoded for 64-bit as:

- Effective Address: **r**A, where **r**A ≠ 0
- Block Size: **r**B[35-39] if **r**B[35-39] ≠ 0; otherwise 32
- Block Count: **r**B[40-47] if **r**B[40-47] ≠ 0; otherwise 256
- Block Stride: **r**B[48-63] if **r**B[48-63] ≠ 0; otherwise 32768

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

Figure 6-2. Format of rB in dst instruction (64-bit)

Other registers altered:

- None

Simplified mnemonics:

dst	**r**A,**r**B,STRM	equivalent to	**dst**	**r**A,**r**B,STRM,**0**
dstt	**r**A,**r**B,STRM	equivalent to	**dst**	**r**A,**r**B,STRM,**1**

For more information on the **dst** instruction, refer to Chapter 5, “Cache, Exceptions, and Memory Management.”

dstst

dstst

Data Stream Touch for Store

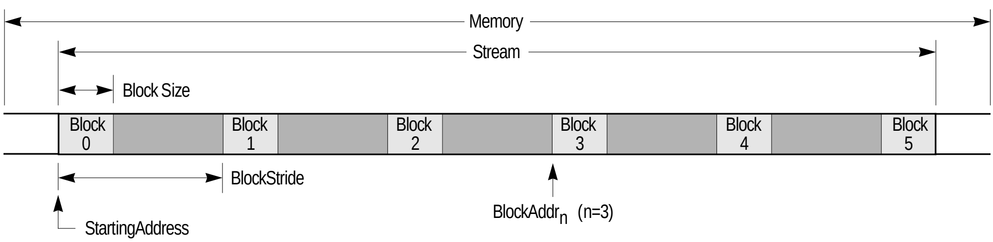
dstst	**r**A,**r**B,STRM	(T=0)	Form X
dststt	**r**A,**r**B,STRM	(T=1)	

31	T	0 0	STRM	A	B	374	0
0–5	6	7–8	9–10	11–15	16–20	21–30	31

addr$_{0:63}$ ← (**r**A)
DataStreamPrefetchControl ← “start” ‖ T ‖ static ‖ (**r**B) ‖ addr

This instruction initiates a software directed cache prefetch. The instruction is a hint to hardware that performance will probably be improved if the cache blocks containing the specified data stream are fetched into the data cache because the program will probably soon write to (store into) the stream.

The instruction associates the data stream specified by the contents of **r**A and **r**B with the stream ID specified by **STRM**. The instruction defines a data stream **STRM** as starting at an “Effective Address” (**r**A) and having “Count” units of “Size” bytes separated by “Stride” bytes (as specified in **r**B). The **T** bit of the instruction indicates whether the data stream is likely to be stored into fairly frequently in the near future (**T** = 0) or to be transient and referenced very few times (**T** = 1).



The **dstst** instruction does the following:

- Defines the characteristics of a data stream **STRM** by the contents of **r**A and **r**B
- Associates the stream with a specified stream ID, **STRM** (Range for STRM is 0-3)
- Indicates that the data in the specified stream **STRM** starting at the address in **r**A may soon be stored in to memory
- Indicates whether memory locations within the stream are likely to be stored into fairly frequently in the near future (**T**=0) or be treated as transient data (**T**=1)
- Terminates prefetching from any stream that was previously associated with the specified stream ID, **STRM**.

The specified data stream is encoded for 32-bit as:

- Effective Address: **r**A, where **r**A ≠ 0
- Block Size: **r**B[3-7] if **r**B[3-7] ≠ 0; otherwise 32
- Block Count: **r**B[8-15] if **r**B[8-15] ≠ 0; otherwise 256
- Block Stride: **r**B[16-31] if **r**B[16-31] ≠ 0; otherwise 32768

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

Figure 6-3. Format of rB in dst instruction (32-Bit)

The specified data stream is encoded for 64-bit as:

- Effective Address: **r**A, where **r**A ≠ 0
- Block Size: **r**B[35-39] if **r**B[35-39] ≠ 0; otherwise 32
- Block Count: **r**B[40-47] if **r**B[40-47] ≠ 0; otherwise 256
- Block Stride: **r**B[48-63] if **r**B[48-63] ≠ 0; otherwise 32768

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

Figure 6-4. Format of rB in dst instruction (64-Bit)

Other registers altered:

- None

Simplified mnemonics:

dstst	**r**A,**r**B,STRM	equivalent to	**dstst** **r**A,**r**B,STRM,**0**
dststt	**r**A,**r**B,STRM	equivalent to	**dstst** **r**A,**r**B,STRM,**1**

For more information on the **dstst** instruction, refer to Chapter 5, “Cache, Exceptions, and Memory Management.”

lvebx lvebx

Load Vector Element Byte Indexed

lvebx **v**D,**r**A,**r**B Form X

31	**v**D	A	B	7	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← EA28:31
vD ← undefined
if the processor is in big-endian mode
 then vDeb*8:(eb*8)+7← MEM(EA,1)
 else vD120-(eb*8):127-(eb*8)← MEM(EA,1)
```

— EA = (**r**A|0)+(**r**B); m = EA[28-31] (the offset of the byte in its aligned quadword).

- For 64-bit:

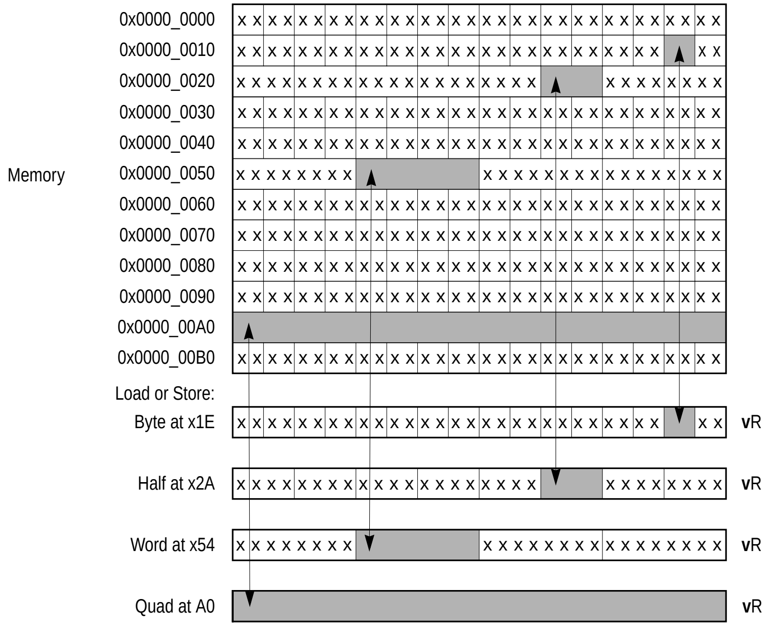
```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← EA60:63
vD ← undefined
if the processor is in big-endian mode
 then vDeb*8:(eb*8)+7← MEM(EA,1)
 else vD120-(eb*8):127-(eb*8)← MEM(EA,1)
```

— EA = (**r**A|0)+(**r**B); m = EA[60–63] (the offset of the byte in its aligned quadword).

For big-endian mode, the byte addressed by EA is loaded into byte m of **v**D. In little-endian mode, it is loaded into byte (15–m) of **v**D. Remaining bytes in **v**D are undefined.

Other registers altered:

- None



Note: In vector registers, x means boundedly undefined after a load and don’t care after a store. In memory, x means don’t care after a load, and leave at current value after a store.

Figure 6-5. Effects of Example Load/Store Instructions

lvehx lvehx

Load Vector Element Half Word Indexed

lvehx **v**D,**r**A,**r**B Form X

0–5	6–10	11–15	16–20	21–30	31
31	**v**D	A	B	39	0

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~1)
eb ← EA28:31
vD ← undefined
if the processor is in big-endian mode
 then vD(eb*8):(eb*8)+15← MEM(EA,2)
 else vD112-(eb*8):127-(eb*8)← MEM(EA,2)
```

  — Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~1. Let m = EA[28-30]; m is the half-word offset of the half-word in its aligned quadword in memory.

- For 64-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~1)
eb ← EA60:63
vD ← undefined
if the processor is in big-endian mode
 then vD(eb*8):(eb*8)+15← MEM(EA,2)
 else vD112-(eb*8):127-(eb*8)← MEM(EA,2)
```

  — Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~1. Let m = EA[60–62]; m is the half-word offset of the half-word in its aligned quadword in memory.

If the processor is in big-endian mode, the half-word addressed by EA is loaded into half-word m of **v**D. If the processor is in little-endian mode, the half-word addressed by EA is loaded into half-word (7-m) of **v**D. The remaining half-word s in **v**D are set to undefined values. Figure 6-5 shows this instruction.

Other registers altered:

- None

lvewx lvewx

Load Vector Element Word Indexed

lvewx **v**D,**r**A,**r**B Form X

31	**v**D	A	B	71	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~3)
eb ← EA28:31
vD ← undefined
if the processor is in big-endian mode
 then vDeb*8:(eb*8)+31← MEM(EA,4)
 else vD96-(eb*8):127-(eb*8)← MEM(EA,4)
```

  — Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~3. Let m = EA[28–29]; m is the word offset of the word in its aligned quadword in memory.

- For 64-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~3)
eb ← EA60:63
vD ← undefined
if the processor is in big-endian mode
 then vDeb*8:(eb*8)+31← MEM(EA,4)
 else vD96-(eb*8):127-(eb*8)← MEM(EA,4)
```

  — Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~3. Let m = EA[60–61]; m is the word offset of the word in its aligned quadword in memory.

If the processor is in big-endian mode, the word addressed by EA is loaded into word m of **v**D. If the processor is in little-endian mode, the word addressed by EA is loaded into word (3-m) of **v**D. The remaining words in **v**D are set to undefined values. Figure 6-5 shows this instruction.

Other registers altered:

- None

lvsl lvsl

Load Vector for Shift Left

lvsl **v**D,**r**A,**r**B Form X

31	**v**D	A	B	6	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA = 0 then b ← 0
    else b ← (rA)
addr_{0:31} ← b + (rB)
sh ← addr_{28-31}
if sh = 0x0 then (vD)_{0:127} ← 0x000102030405060708090A0B0C0D0E0F
if sh = 0x1 then (vD)_{0:127} ← 0x0102030405060708090A0B0C0D0E0F10
if sh = 0x2 then (vD)_{0:127} ← 0x02030405060708090A0B0C0D0E0F1011
if sh = 0x3 then (vD)_{0:127} ← 0x030405060708090A0B0C0D0E0F101112
if sh = 0x4 then (vD)_{0:127} ← 0x0405060708090A0B0C0D0E0F10111213
if sh = 0x5 then (vD)_{0:127} ← 0x05060708090A0B0C0D0E0F1011121314
if sh = 0x6 then (vD)_{0:127} ← 0x060708090A0B0C0D0E0F101112131415
if sh = 0x7 then (vD)_{0:127} ← 0x0708090A0B0C0D0E0F10111213141516
if sh = 0x8 then (vD)_{0:127} ← 0x08090A0B0C0D0E0F1011121314151617
if sh = 0x9 then (vD)_{0:127} ← 0x090A0B0C0D0E0F101112131415161718
if sh = 0xA then (vD)_{0:127} ← 0x0A0B0C0D0E0F10111213141516171819
if sh = 0xB then (vD)_{0:127} ← 0x0B0C0D0E0F101112131415161718191A
if sh = 0xC then (vD)_{0:127} ← 0x0C0D0E0F101112131415161718191A1B
if sh = 0xD then (vD)_{0:127} ← 0x0D0E0F101112131415161718191A1B1C
if sh = 0xE then (vD)_{0:127} ← 0x0E0F101112131415161718191A1B1C1D
if sh = 0xF then (vD)_{0:127} ← 0x0F101112131415161718191A1B1C1D1E
```

— Let the EA be the sum (**r**A|0)+(**r**B). Let sh = EA[28–31].

- For 64-bit:

```
if rA = 0 then b ← 0
    else b ← (rA)
addr_{0:63} ← b + (rB)
sh ← addr_{60:63}
if sh = 0x0 then (vD)_{0:127} ← 0x000102030405060708090A0B0C0D0E0F
if sh = 0x1 then (vD)_{0:127} ← 0x0102030405060708090A0B0C0D0E0F10
if sh = 0x2 then (vD)_{0:127} ← 0x02030405060708090A0B0C0D0E0F1011
if sh = 0x3 then (vD)_{0:127} ← 0x030405060708090A0B0C0D0E0F101112
if sh = 0x4 then (vD)_{0:127} ← 0x0405060708090A0B0C0D0E0F10111213
if sh = 0x5 then (vD)_{0:127} ← 0x05060708090A0B0C0D0E0F1011121314
if sh = 0x6 then (vD)_{0:127} ← 0x060708090A0B0C0D0E0F101112131415
if sh = 0x7 then (vD)_{0:127} ← 0x0708090A0B0C0D0E0F10111213141516
if sh = 0x8 then (vD)_{0:127} ← 0x08090A0B0C0D0E0F1011121314151617
if sh = 0x9 then (vD)_{0:127} ← 0x090A0B0C0D0E0F101112131415161718
if sh = 0xA then (vD)_{0:127} ← 0x0A0B0C0D0E0F10111213141516171819
if sh = 0xB then (vD)_{0:127} ← 0x0B0C0D0E0F101112131415161718191A
if sh = 0xC then (vD)_{0:127} ← 0x0C0D0E0F101112131415161718191A1B
if sh = 0xD then (vD)_{0:127} ← 0x0D0E0F101112131415161718191A1B1C
if sh = 0xE then (vD)_{0:127} ← 0x0E0F101112131415161718191A1B1C1D
if sh = 0xF then (vD)_{0:127} ← 0x0F101112131415161718191A1B1C1D1E
```

— Let the EA be the sum (**r**A|0)+(**r**B). Let sh = EA[60–63].

Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1E || 0x1F. Bytes sh:sh+15 of X are placed into **v**D. Figure 6-6 shows how this instruction works.

Other registers altered:

- None

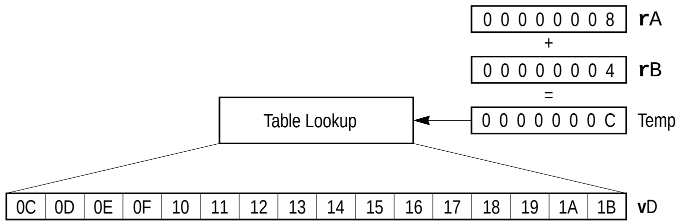


Figure 6-6. Load Vector for Shift Left

The above **lvsl** instruction followed by a Vector Permute (**vperm**) would do a simulated alignment of a four-element floating-point vector misaligned on quad-word boundary at address 0x0....C.

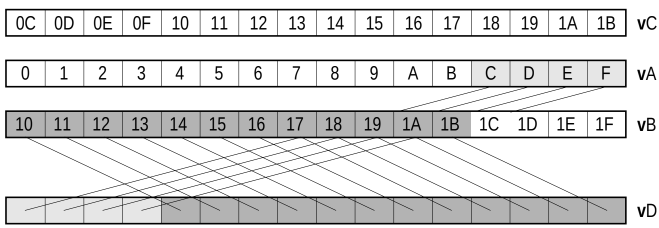


Figure 6-7. Instruction vperm Used in Aligning Data

Refer, also, to the description of the **lvsr** instruction for suggested uses of the **lvsl** instruction.

lvsr lvsr

Load Vector for Shift Right

lvsr **v**D,**r**A,**r**B Form X

31	**v**D	A	B	38	0
0–5	6–10	11–15	16–20	21–30	31

- For 32-bit:

```
if rA = 0 then b ← 0
else           b ← (rA)
EA ← b + (rB)
sh ← EA28:31
if sh=0x0 then vD ← 0x101112131415161718191A1B1C1D1E1F
if sh=0x1 then vD ← 0x0F101112131415161718191A1B1C1D1E
if sh=0x2 then vD ← 0x0E0F101112131415161718191A1B1C1D
if sh=0x3 then vD ← 0x0D0E0F101112131415161718191A1B1C
if sh=0x4 then vD ← 0x0C0D0E0F101112131415161718191A1B
if sh=0x5 then vD ← 0x0B0C0D0E0F101112131415161718191A
if sh=0x6 then vD ← 0x0A0B0C0D0E0F10111213141516171819
if sh=0x7 then vD ← 0x090A0B0C0D0E0F101112131415161718
if sh=0x8 then vD ← 0x08090A0B0C0D0E0F1011121314151617
if sh=0x9 then vD ← 0x0708090A0B0C0D0E0F10111213141516
if sh=0xA then vD ← 0x060708090A0B0C0D0E0F101112131415
if sh=0xB then vD ← 0x05060708090A0B0C0D0E0F1011121314
if sh=0xC then vD ← 0x0405060708090A0B0C0D0E0F10111213
if sh=0xD then vD ← 0x030405060708090A0B0C0D0E0F101112
if sh=0xE then vD ← 0x02030405060708090A0B0C0D0E0F1011
if sh=0xF then vD ← 0x0102030405060708090A0B0C0D0E0F10
```

  — Let the EA be the sum (**r**A|0)+(**r**B). Let sh = EA[28–31].

- For 64-bit:

```
if rA = 0 then b ← 0
else           b ← (rA)
EA ← b + (rB)
sh ← EA60:63
if sh=0x0 then vD ← 0x101112131415161718191A1B1C1D1E1F
if sh=0x1 then vD ← 0x0F101112131415161718191A1B1C1D1E
if sh=0x2 then vD ← 0x0E0F101112131415161718191A1B1C1D
if sh=0x3 then vD ← 0x0D0E0F101112131415161718191A1B1C
if sh=0x4 then vD ← 0x0C0D0E0F101112131415161718191A1B
if sh=0x5 then vD ← 0x0B0C0D0E0F101112131415161718191A
if sh=0x6 then vD ← 0x0A0B0C0D0E0F10111213141516171819
if sh=0x7 then vD ← 0x090A0B0C0D0E0F101112131415161718
if sh=0x8 then vD ← 0x08090A0B0C0D0E0F1011121314151617
if sh=0x9 then vD ← 0x0708090A0B0C0D0E0F10111213141516
if sh=0xA then vD ← 0x060708090A0B0C0D0E0F101112131415
if sh=0xB then vD ← 0x05060708090A0B0C0D0E0F1011121314
if sh=0xC then vD ← 0x0405060708090A0B0C0D0E0F10111213
if sh=0xD then vD ← 0x030405060708090A0B0C0D0E0F101112
if sh=0xE then vD ← 0x02030405060708090A0B0C0D0E0F1011
if sh=0xF then vD ← 0x0102030405060708090A0B0C0D0E0F10
```

  — Let the EA be the sum (**r**A|0)+(**r**B). Let sh = EA[60-63].

Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1E || 0x1F. Bytes (16-sh):(31-sh) of X are placed into **v**D.

Note that **lvsl** and **lvsr** can be used to create the permute control vector to be used by a subsequent **vperm** instruction. Let X and Y be the contents of **v**A and **v**B specified by the **vperm**. The control vector created by **lvsl** causes the **vperm** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes. The control vector created by **vsr** causes the **vperm** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

These instructions can also be used to rotate or shift the contents of a vector register by sh bytes. For rotating, the vector register to be rotated should be specified as both **v**A and **v**B for **vperm**. For shifting left, the **v**B register for **vperm** should contain all zeros and **v**A should contain the value to be shifted, and vice versa for shifting right. Figure 6-6 shows a similar instruction only in that figure the shift is to the left

Other registers altered:

- None

lvx lvx

Load Vector Indexed

lvx **v**D,**r**A,**r**B (LRU = 0) Form X

31	**v**D	A	B	103	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit and 64-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0xF)
if the processor is in big-endian mode
 then vD ← MEM(EA,16)
 else vD ← MEM(EA+8,8) || MEM(EA,8)
```

Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~0xF.

If the processor is in big-endian mode, the quadword in memory addressed by EA is loaded into **v**D.

If the processor is in little-endian mode, the doubleword addressed by EA is loaded into **v**D[64–127] and the doubleword addressed by EA+8 is loaded into **v**D[0–63]. Note that normal little-endian PowerPC address swizzling is also performed. See Section 3.1, “Data Organization in Memory,” for more information.

Figure 6-5 shows this instruction.

Other registers altered:

- None

lvxl lvxl

Load Vector Indexed LRU

lvxl **v**D,**r**A,**r**B (LRU = 1) Form X

31	vD	A	B	359	0
0–5	6–10	11–15	16–20	21–30	31

- For 32-bit and 64-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0xF)
if the processor is in big-endian mode
 then vD ← MEM(EA,16)
 else vD ← MEM(EA+8,8) || MEM(EA,8)
```

Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~0xF.

If the processor is in big-endian mode, the quadword addressed by EA is loaded into **v**D.

If the processor is in little-endian mode, the doubleword addressed by EA is loaded into **v**D[64–127] and the doubleword addressed by EA+8 is loaded into **v**D[0–63]. Note that normal little-endian PowerPC address swizzling is also performed. See Section 3.1, “Data Organization in Memory,” for more information.

lvxl provides a hint that the quadword addressed by EA will probably not be needed again by the program in the near future.

Note that on some implementations, the hint provided by the **lvxl** instruction and the corresponding hint provided by the Store Vector Indexed LRU (**stvxl**) instruction (see Section 5.2.1.2, “Transient Streams (dstt)”) are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for reuse when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference. Figure 6-5 shows this instruction.

Other registers altered:

- None

mfvscr mfvscr

Move from Vector Status and Control Register

mfvscr **v**D Form VX

04	**v**D	0 0 0 0 0	0 0 0 0 0	1540	0
0 5	6 10	11 15	16 20	21 30	31

vD ← ^{96}0 ‖ (VSCR)

The contents of the VSCR are placed into **v**D.

Note that the programmer should assume that **mtvscr** and **mfvscr** take substantially longer to execute than other VX instructions

Other registers altered:

- None

mtvscr mtvscr

Move to Vector Status and Control Register

mtvscr **v**B Form VX

04	0 0 0 0 0	0 0 0 0 0	**v**B	1604	0
0 5	6 10	11 15	16 20	21 30	31

$VSCR \leftarrow (\mathbf{v}B)_{96:127}$

The contents of **v**B are placed into the VSCR.

Other registers altered:

- None

stvebx stvebx

Store Vector Element Byte Indexed

stvebx **v**S,**r**A,**r**B Form X

31	**v**S	A	B	135	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← EA28:31
if the processor is in big-endian mode
 then MEM(EA,1) ← (vS)eb*8:(eb*8)+7
 else MEM(EA,1) ← (vS)120-(eb*8):127-eb*8
```

  — Let the EA be the sum (**r**A|0)+(**r**B). Let m = EA[28–31]; m is the byte offset of the byte in its aligned quadword in memory.

- For 64-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← EA60:63
if the processor is in big-endian mode
 then MEM(EA,1) ← (vS)eb*8:(eb*8)+7
 else MEM(EA,1) ← (vS)120-(eb*8):127-eb*8
```

  — Let the EA be the sum (**r**A|0)+(**r**B). Let m = EA[60–63]; m is the byte offset of the byte in its aligned quadword in memory.

If the processor is in big-endian mode, byte m of **v**S is stored into the byte in memory addressed by EA. If the processor is in little-endian mode, byte (15-m) of **v**S is stored into the byte addressed by EA. Figure 6-5 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvehx stvehx

Store Vector Element Half Word Indexed

stvehx **v**S,**r**A,**r**B Form X

31	vS	A	B	167	0
0–5	6–10	11–15	16–20	21–30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0x1)
eb ← EA28:31
if the processor is in big-endian mode
 then MEM(EA,2) ← (vS)eb*8:(eb*8)+15
 else MEM(EA,2) ← (vS)112-eb*8:127-(eb*8)
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~0x1. Let m = EA[28–30]; m is the half-word offset of the half-word in its aligned quadword in memory.

- For 64-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0x1)
eb ← EA60:63
if the processor is in big-endian mode
 then MEM(EA,2) ← (vS)eb*8:(eb*8)+15
 else MEM(EA,2) ← (vS)112-(eb*8):127-eb*8
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with ~0x1. Let m = EA[60–62]; m is the half-word offset of the half-word in its aligned quadword in memory.

If the processor is in big-endian mode, half-word m of **v**S is stored into the half-word addressed by EA. If the processor is in little-endian mode, half-word (7-m) of **v**S is stored into the half-word addressed by EA. Figure 6-5 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvewx stvewx

Store Vector Element Word Indexed

stvewx **v**S,**r**A,**r**B Form X

31	**v**S	A	B	199	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFC
eb ← EA28:31
if the processor is in big-endian mode
 then MEM(EA,4) ← (vS)eb*8:(eb*8)+31
 else MEM(EA,4) ← (vS)96-eb*8:127-(eb*8)
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFFC. Let m = EA[28-29]; m is the word offset of the word in its aligned quadword in memory.

- For 64-bit:

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFFC
eb ← EA60:63
if the processor is in big-endian mode
 then MEM(EA,4) ← (vS)eb*8:(eb*8)+31
 else MEM(EA,4) ← (vS)96-eb*8:127-(eb*8)
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFFF_FFFF_FFFC. Let m = EA[60-61]; m is the word offset of the word in its aligned quadword in memory.

If the processor is in big-endian mode, word m of **v**S is stored into the word addressed by EA. If the processor is in little-endian mode, word (3-m) of **v**S is stored into the word addressed by EA. Figure 6-5 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvx stvx

Store Vector Indexed

stvx **v**S,**r**A,**r**B (LRU = 0) Form X

31	**v**S	A	B	231	0
0 5	6 10	11 15	16 20	21 30	31

- For 32-bit:

```
if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFF0
if the processor is in big-endian mode
 then MEM(EA,16) ← (vS)
 else MEM(EA,16) ← (vS)64:127 ‖ (vS)0:63
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFF0.

- For 64-bit:

```
if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFF0
if the processor is in big-endian mode
 then MEM(EA,16) ← (vS)
 else MEM(EA,16) ← (vS)64:127 ‖ (vS)0:63
```

— Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFFF_FFFF_FFF0.

If the processor is in big-endian mode, the contents of **v**S are stored into the quadword addressed by EA. If the processor is in little-endian mode, the contents of **v**S[64–127] are stored into the doubleword addressed by EA, and the contents of **v**S[0–63] are stored into the doubleword addressed by EA+8.

stvxl and **stvxlt** provide a hint that the quadword addressed by EA will probably not be needed again by the program in the near future.

Figure 6-5 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvxl

Store Vector Indexed LRU

stvxl **v**S,**r**A,**r**B (LRU = 1) Form X

31	vS	A	B	487	0
0–5	6–10	11–15	16–20	21–30	31

- For 32-bit:

```
if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFF0
if the processor is in big-endian mode
 then MEM(EA,16) ← (vS)
 else MEM(EA,16) ← (vS)64:127 || (vS)0:63
```

  — Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFF0.

- For 64-bit:

```
if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFF0
if the processor is in big-endian mode
 then MEM(EA,16) ← (vS)
 else MEM(EA,16) ← (vS)64:127 || (vS)0:63
```

  — Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFFF_FFFF_FFF0.

Let the EA be the result of ANDing the sum (**r**A|0)+(**r**B) with 0xFFFF_FFFF_FFFF_FFF0. If the processor is in big-endian mode, the contents of **v**S are stored into the quadword addressed by EA. If the processor is in little-endian mode, the contents of **v**S[64–127] are stored into the doubleword addressed by EA, and the contents of **v**S[0–63] are stored into the doubleword addressed by EA+8. The **stvxl** and **stvxlt** instructions provide a hint that the quad word addressed by EA will probably not be needed again by the program in the near future.

Note that on some implementations, the hint provided by the **stvxl** instruction (see Section 5.2.2, "Prioritizing Cache Block Replacement") is applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for reuse when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference. Figure 6-5 shows how a store instruction is performed on the vector registers.

Other registers altered:

- None

vaddcuw

vaddcuw

Vector Add Carryout Unsigned Word

vaddcuw **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	384
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  aop0:32 ← ZeroExtend((vA)i:i+31,33)
  bop0:32 ← ZeroExtend((vB)i:i+31,33)
  temp0:32← aop0:32 +int bop0:32
  vDi:i+31← ZeroExtend(temp0,32)
end
```

Each unsigned-integer word element in **v**A is added to the corresponding unsigned-integer word element in **v**B. The carry out of bit 0 of the 32-bit sum is zero-extended to 32 bits and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-8 shows the usage of the **vaddcuw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

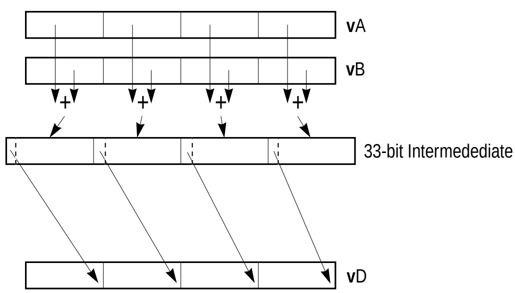


Figure 6-8. vaddcuw—Determine Carries of Four Unsigned Integer Adds (32-Bit)

vaddfp vaddfp

Vector Add Floating Point

vaddfp **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	10
0–5	6–10	11–15	16–20	21–31

```
do i = 0,127,32
  (vD)i:i+31 ← RndToNearFP32((vA)i:i+31 +fp (vB)i:i+31)
end
```

The four 32-bit floating-point values in **v**A are added to the four 32-bit floating-point values in **v**B. The four intermediate results are rounded and placed in VD.

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.

Other registers altered:

- None

Figure 6-9 shows the usage of the **vaddfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

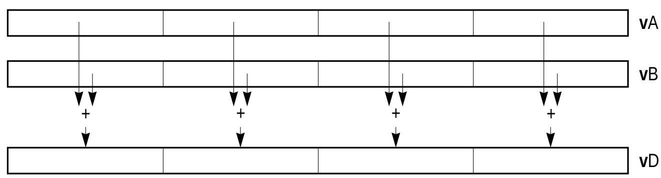


Figure 6-9. vaddfp—Add Four Floating-Point Elements (32-Bit)

vaddsbs vaddsbs

Vector Add Signed Byte Saturate

vaddsbs **v**D,**v**A,**v**B Form VX

04	vD	vA	vB	768
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 8
  aop0:8 ← SignExtend((vA)i:i+7,9)
  bop0:8 ← SignExtend((vB)i:i+7,9)
  temp0:8← aop0:8 +int bop0:8
  vDi:i+7 ← SItoSIsat(temp0:8,8)
end
```

Each element of **vaddsbs** is a byte.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B.

If the sum is greater than (2^7-1) it saturates to (2^7-1) and if it is less than -2^7 it saturates to -2^7. If saturation occurs, the SAT bit is set.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-10 shows the usage of the **vaddsbs** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

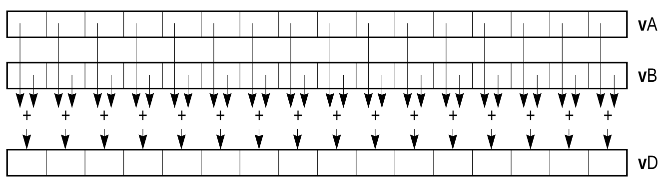


Figure 6-10. vaddsbs— Add Saturating Sixteen Signed Integer Elements (8-Bit)

vaddshs

vaddshs

Vector Add Signed Half Word Saturate

vaddshs **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	832
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  aop0:16 ← SignExtend((vA)i:i+15,16)
  bop0:16 ← SignExtend((vB)i:i+15,16)
  temp0:16← aop0:16 +int bop0:16
  vDi:i+15← SItoSIsat(temp0:16,16)
end
```

Each element of **vaddshs** is a half word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B.

If the sum is greater than $(2^{15}-1)$ it saturates to $(2^{15}-1)$ and if it is less than -2^{15} it saturates to -2^{15}. If saturation occurs, the SAT bit is set.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-16 shows the usage of the **vaddshs** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

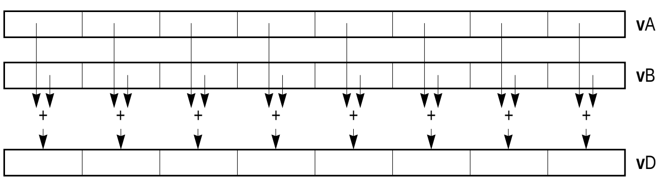


Figure 6-11. vaddshs— Add Saturating Eight Signed Integer Elements (16-Bit)

vaddsws

vaddsws

Vector Add Signed Word Saturate

vaddsws **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	896
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
  aop0:32 ← SignExtend((vA)i:i+31,33)
  bop0:32 ← SignExtend((vB)i:i+31,33)
  temp0:32← aop0:32 +int bop0:32
  vDi:i+31← SItoSIsat(temp0:32,32)
end
```

Each element of **vaddsws** is a word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B.

If the sum is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than (-2^{31}) it saturates to (-2^{31}). If saturation occurs, the SAT bit is set.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-12 shows the usage of the **vaddsws** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

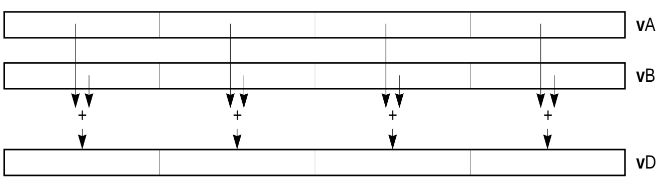


Figure 6-12. vaddsws—Add Saturating Four Signed Integer Elements (32-Bit)

vaddubm vaddubm

Vector Add Unsigned Byte Modulo

vaddubm **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	0
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  vD[i:i+7] ← (vA)[i:i+7] +int (vB)[i:i+7]
end
```

Each element of **vaddubm** is a byte.

Each integer element in **v**A is modulo added to the corresponding integer element in **v**B. The integer result is placed into the corresponding element of **v**D.

Note that the **vaddubm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-13 shows the **vaddubm** command usage. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

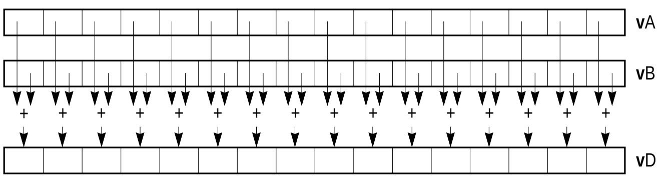


Figure 6-13. vaddubm—Add Sixteen Integer Elements (8-Bit)

vaddubs

vaddubs

Vector Add Unsigned Byte Saturate

vaddubs **v**D,**v**A,**v**B Form VX

04	vD	vA	vB	512
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 8
  aop0:8 ← ZeroExtend((vA)i:i+7,9)
  bop0:8 ← ZeroExtend((vB)i:i+7,9)
  temp0:8← aop0:8 +int bop0:8
  vDi:i+7 ← UItoUIsat(temp0:8,8)
end
```

Each element of **vaddubs** is a byte.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B.

If the sum is greater than (2^8-1) it saturates to (2^8-1) and the SAT bit is set.

The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-14 shows the usage of the **vaddubs** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

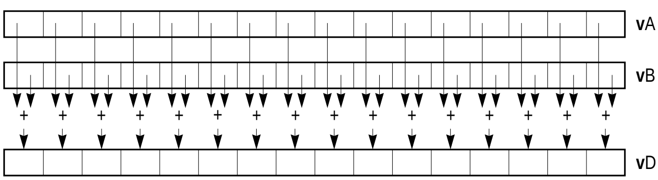


Figure 6-14. vaddubs—Add Saturating Sixteen Unsigned Integer Elements (8-Bit)

vadduhm vadduhm

Vector Add Unsigned Half Word Modulo

vadduhm **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	64
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  vD_{i:i+15} ← (vA)_{i:i+15} +_{int} (vB)_{i:i+15}
end
```

Each element of **vadduhm** is a half word.

Each integer element in **v**A is added to the corresponding integer element in **v**B. The integer result is placed into the corresponding element of **v**D.

Note that the **vadduhm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-15 shows the usage of the **vadduhm** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

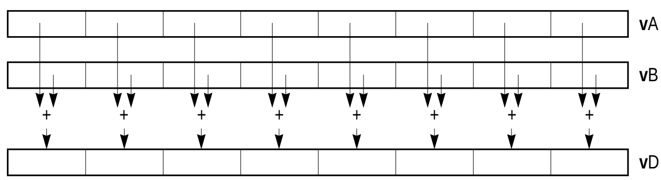


Figure 6-15. vadduhm—Add Eight Integer Elements (16-Bit)

vadduhs vadduhs

Vector Add Unsigned Half Word Saturate

vadduhs **v**D,**v**A,**v**B Form VX

04	**v**D	**v**A	**v**B	576
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  aop0:16 ← ZeroExtend((vA)i:i+15,17)
  bop0:16← ZeroExtend((vB)i:i+15,17)
  temp0:16← aop0:16 +int bop0:16
  vDi:i+15← UItoUIsat(temp0:16,16)
end
```

Each element of **vadduhs** is a half word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B.

If the sum is greater than $(2^{16}-1)$ it saturates to $(2^{16}-1)$ and the SAT bit is set.

The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-16 shows the usage of the **vadduhs** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

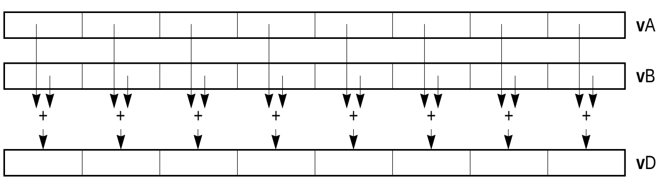


Figure 6-16. vadduhs—Add Saturating Eight Unsigned Integer Elements (16-Bit)

vadduwm vadduwm

Vector Add Unsigned Word Modulo

vadduwm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	128
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vDi:i+31← (vA)i:i+31 +int (vB)i:i+31
end
```

Each element of **vadduwm** is a word.

Each integer element in **v**A is modulo added to the corresponding integer element in **v**B. The integer result is placed into the corresponding element of **v**D.

Note that the **vadduwm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Form:

- VX

Figure 6-17 shows the usage of the **vadduwm** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

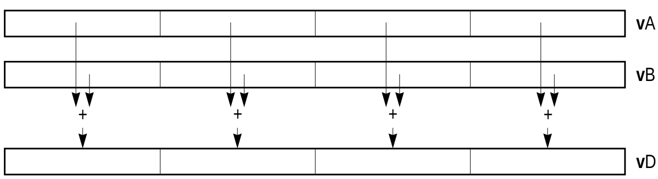


Figure 6-17. vadduwm—Add Four Integer Elements (32-Bit)

vadduws

Vector Add Unsigned Word Saturate

vadduws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	640
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 3
  aop0:32 ← ZeroExtend((vA)i:i+31,33)
  bop0:32 ← ZeroExtend((vB)i:i+31,33)
  temp0:32← aop0:32 +int bop0:32
  vDi:i+31← UItoUIsat(temp0:32,32)
end
```

Each element of **vadduws** is a word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B.

If the sum is greater than $(2^{32}-1)$ it saturates to $(2^{32}-1)$ and the SAT bit is set.

The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-18 shows the usage of the **vadduws** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

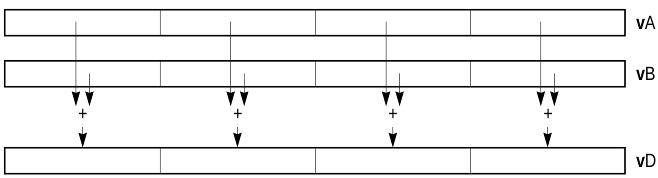


Figure 6-18. vadduws—Add Saturating Four Unsigned Integer Elements (32-Bit)

vand

Vector Logical AND

vand **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1028
0–5	6–10	11–15	16–20	21–31

vD ← (**v**A) & (**v**B)

The contents of **v**A are bitwise ANDed with the contents of **v**B and the result is placed into **v**D.

Other registers altered:

- None

Figure 6-19 shows usage of the **vand** command.

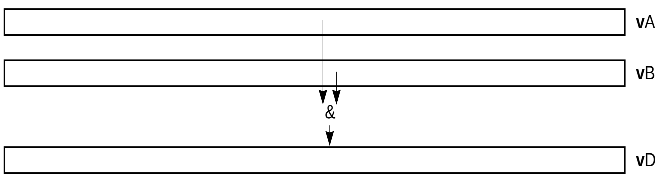


Figure 6-19. vand—Logical Bitwise AND

vandc vandc

Vector Logical AND with Complement

vandc **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1092
0 5	6 10	11 15	16 20	21 31

vD ← (**v**A) & ¬(**v**B)

The contents of **v**A are ANDed with the one’s complement of the contents of **v**B and the result is placed into **v**D.

Other registers altered:

- None

Figure 6-19 shows usage of the **vandc** command.

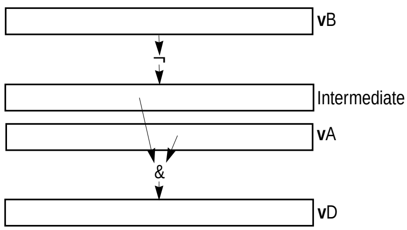


Figure 6-20. vand—Logical Bitwise AND with Complement

vavgsb

vavgsb

Vector Average Signed Byte

vavgsb **v**D,**v**A,**v**B   Form: VX

04	**v**D	**v**A	**v**B	1282
0 – 5	6 – 10	11 – 15	16 – 20	21 – 31

do i=0 to 127 by 8
  $aop_{0:8} \leftarrow$ SignExtend((**v**A)$_{i:i+7}$,9)
  $bop_{0:8} \leftarrow$ SignExtend((**v**B)$_{i:i+7}$,9)
  $temp_{0:8} \leftarrow aop_{0:8} +_{int} bop_{0:8} +_{int} 1$
  vD$_{i:i+7} \leftarrow temp_{0:7}$
end

Each element of **vavgsb** is a byte.

Each signed-integer byte element in **v**A is added to the corresponding signed-integer byte element in **v**B, producing an 9-Bit signed-integer sum. The sum is incremented by 1. The high-order 8 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-21 shows the usage of the **vavgsb** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

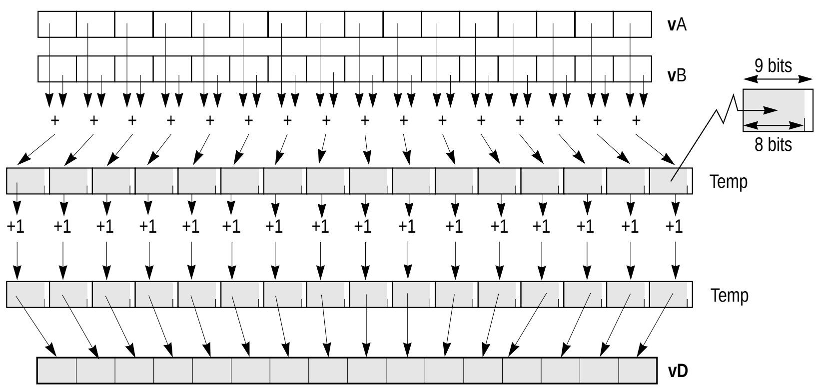


Figure 6-21. vavgsb— Average Sixteen Signed Integer Elements (8-Bit)

vavgsh

vavgsh

Vector Average Signed Half Word

vavgsh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1346
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  aop0:16 ← SignExtend((vA)i:i+15,17)
  bop0:16 ← SignExtend((vB)i:i+15,17)
  temp0:16 ← aop0:15 +int bop0:15 +int 1
  vDi:i+15 ← temp0:15
end
```

Each element of **vavgsh** is a half word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B, producing an 17-bit signed-integer sum. The sum is incremented by 1. The high-order 16 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-22 shows the usage of the **vavgsh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

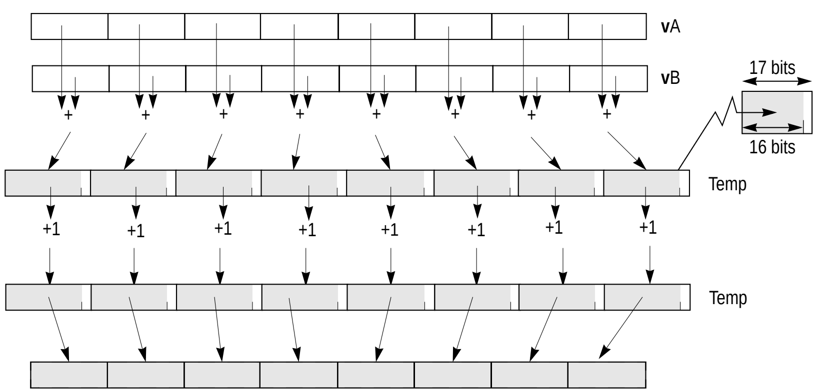


Figure 6-22. vavgsh—Average Eight Signed Integer Elements (16-bits)

vavgsw

Vector Average Signed Word

vavgsw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1410
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  aop0:32 ← SignExtend((vA)i:i+31,33)
  bop0:32 ← SignExtend((vB)i:i+31,33)
  temp0:32 ← aop0:32 +int bop0:32 +int 1
  vDi:i+31 ← temp0:31
end
```

Each element of **vavgsw** is a word.

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B, producing an 33-bit signed-integer sum. The sum is incremented by 1. The high-order 32 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-23 shows the usage of the **vavgsw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

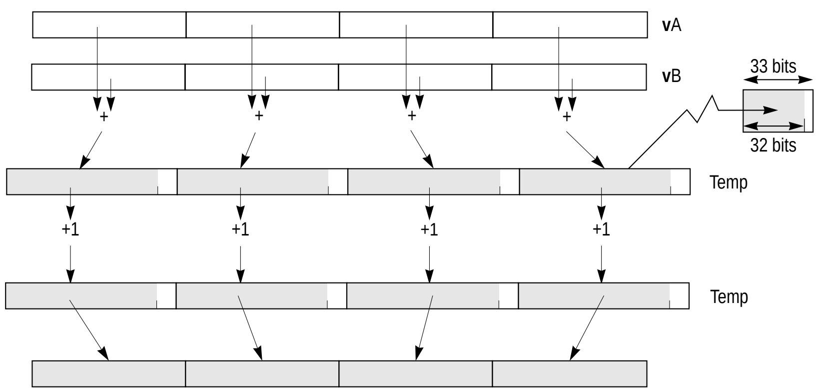


Figure 6-23. vavgsw— Average Four Signed Integer Elements (32-Bit)

vavgub vavgub

Vector Average Unsigned Byte

vavgub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1026
0 5	6 10	11 15	16 20	21 31

do i=0 to 127 by 8
  $aop_{0:8} \leftarrow$ ZeroExtend(($\mathbf{v}$A)$_{i:i+7}$,9)
  $bop_{0:n} \leftarrow$ ZeroExtend(($\mathbf{v}$B)$_{i:i+71}$,9)
  $temp_{0:n} \leftarrow aop_{0:8} +_{int} bop_{0:8} +_{int} 1$
  $\mathbf{v}D_{i:i+7} \leftarrow temp_{0:7}$
end

Each element of **vavgub** is a byte.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing an 9-bit unsigned-integer sum. The sum is incremented by 1. The high-order 8 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-24 shows the usage of the **vavgub** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

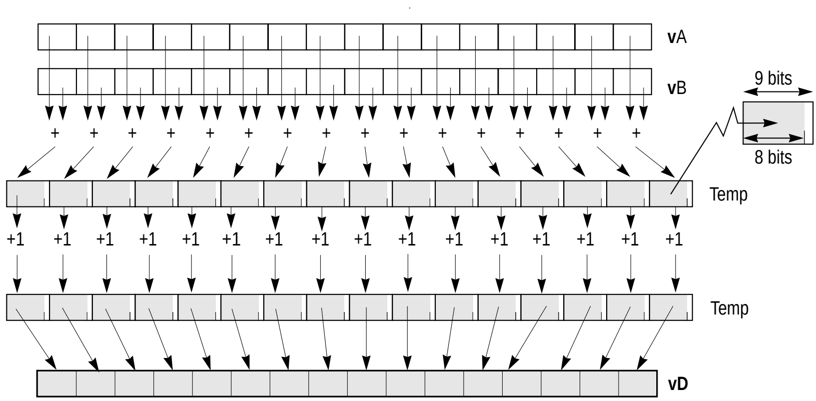


Figure 6-24. vavgub—Average Sixteen Unsigned Integer Elements (8-bits)

vavguh   vavguh

Vector Average Unsigned Half Word

vavguh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1090
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  aop0:16 ← ZeroExtend((vA)i:i+15,17)
  bop0:16 ← ZeroExtend((vB)i:i+15,17)
  temp0:16← aop0:16 +int bop0:16 +int 1
  vDi:i+15← temp0:15
end
```

Each element of **vavguh** is a half word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing a 17-bit unsigned-integer. The sum is incremented by 1. The high-order 16 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-22 shows the usage of the **vavgsh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

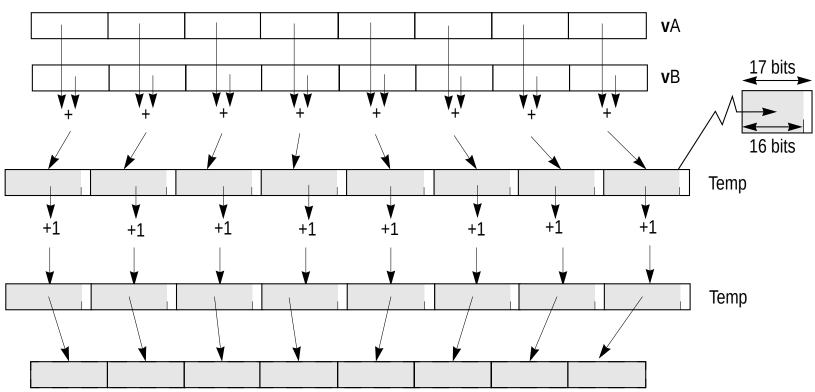


Figure 6-25. vavgsh— Average Eight Signed Integer Elements (16-Bit)

vavguw vavguw

Vector Average Unsigned Word

vavguw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1154
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  aop0:32← ZeroExtend((vA)i:i+31,33)
  bop0:32 ← ZeroExtend((vB)i:i+31,33)
  temp0:32← aop0:32 +int bop0:32 +int 1
  vDi:i+31← temp0:31
end
```

Each element of **vavguw** is a word.

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing an 33-bit unsigned-integer sum. The sum is incremented by 1. The high-order 32 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-26 shows the usage of the **vavguw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

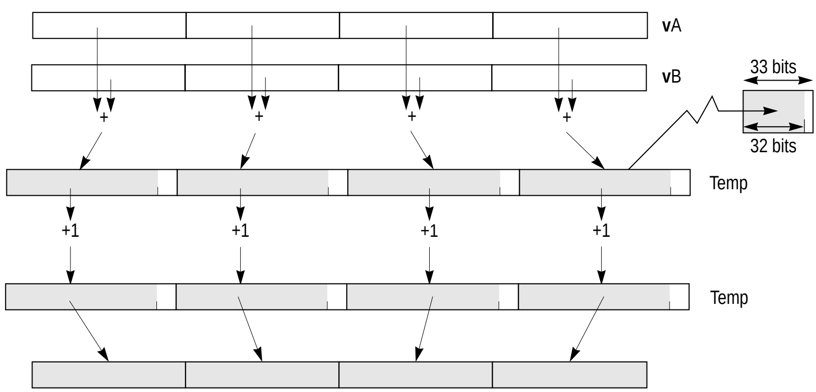


Figure 6-26. vavguw—Average Four Unsigned Integer Elements (32-Bit)

vcfsx vcfsx

Vector Convert from Signed Fixed-Point Word

vcfsx **v**D,**v**B,UIMM Form: VX

04	vD	UIMM	vB	842
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vDi:i+31 ← CnvtSI32ToFP32((vB)i:i+31) ÷fp 2^UIMM
end
```

Each signed fixed-point integer word element in **v**B is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIMM} (UIMM = Unsigned immediate value) and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-26 shows the usage of the **vcfsx** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

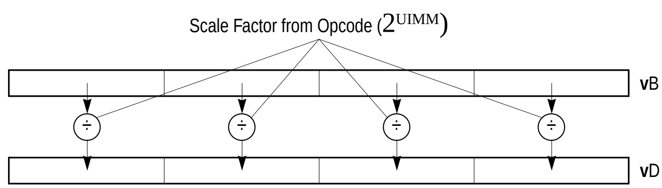


Figure 6-27. vcfsx—Convert Four Signed Integer Elements to Four Floating-Point Elements (32-Bit)

vcfux vcfux

Vector Convert from Unsigned Fixed-Point Word

vcfux **v**D,**v**B,UIMM Form: VX

04	vD	UIMM	vB	778
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vDi:i+31 ← CnvtUI32ToFP32((vB)i:i+31) ÷fp 2^UIMM
end
```

Each unsigned fixed-point integer word element in **v**B is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIMM} and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-28 shows the usage of the **vcfux** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

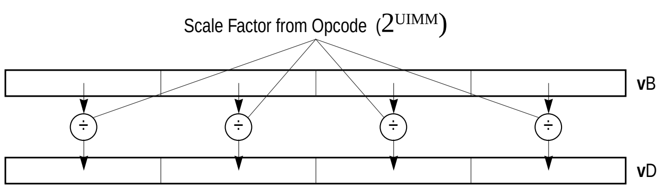


Figure 6-28. vcfux—Convert Four Unsigned Integer Elements to Four Floating-Point Elements (32-Bit)

vcmpbfp*x* vcmpbfp*x*

Vector Compare Bounds Floating Point

vcmpbfp	**v**D,**v**A,**v**B	(Rc = 0)	Form: VXR
vcmpbfp.	**v**D,**v**A,**v**B	(Rc = 1)	

04	**v**D	**v**A	**v**B	Rc	966
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
  le ← ((vA)i:i+31 ≤fp (vB)i:i+31)
  ge ← ((vA)i:i+31 ≥fp -(vB)i:i+31)
  vDi:i+31 ← ¬le || ¬ge || 30 0
end
if Rc=1 then do
  ib ← (vD = 128 0)
  CR24:27 ← 0b00 || ib || 0b0
end
```

Each single-precision word element in **v**A is compared to the corresponding element in **v**B. A 2-bit value is formed that indicates whether the element in **v**A is within the bounds specified by the element in **v**B, as follows.

Bit 0 of the 2-bit value is zero if the element in **v**A is less than or equal to the element in **v**B, and is one otherwise. Bit 1 of the 2-bit value is zero if the element in **v**A is greater than or equal to the negative of the element in **v**B, and is one otherwise.

The 2-bit value is placed into the high-order two bits of the corresponding word element (bits 0–1 for word element 0, bits 32–33 for word element 1, bits 64–65 for word element 2, bits 96–97 for word element 3) of **v**D and the remaining bits of the element are cleared.

If Rc=1, CR Field 6 is set to indicate whether all four elements in **v**A are within the bounds specified by the corresponding element in **v**B, as follows.

- CR6 = 0b00 || all_within_bounds || 0

Note that if any single-precision floating-point word element in **v**B is negative; the corresponding element in **v**A is out of bounds. Note that if a **v**A or a **v**B element is a NaN, the two high order bits of the corresponding result will both have the value 1.

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

Other registers altered:

- Condition register (CR6):

  Affected: Bit 2 (if Rc = 1)

Figure 6-29 shows the usage of the **vcmpbfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

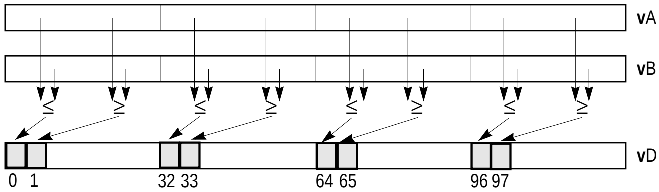


Figure 6-29. vcmpbfp—Compare Bounds of Four Floating-Point Elements (32-Bit)

vcmpeqfp*x* vcmpeqfp*x*

Vector Compare Equal-to-Floating Point

vcmpeqfp **v**D,**v**A,**v**B Form: VXR
vcmpeqfp. **v**D,**v**A,**v**B

04	**v**D	**v**A	**v**B	Rc	198
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
  if (vA)i:i+31 =fp (vB)i:i+31
    then vDi:i+31 ← 0xFFFF_FFFF
    else vDi:i+31 ← 0x0000_0000
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR24:27 ← t || 0b0 || f || 0b0
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1. CR6 filed is set according to all, some, or none of the elements pairs compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0

Note that if a **v**A or **v**B element is a NaN, the corresponding result will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-29 shows the usage of the **vcmpeqfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

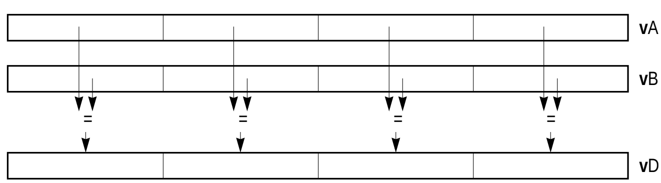


Figure 6-30. vcmpeqfp—Compare Equal of Four Floating-Point Elements (32-Bit)

vcmpequb*x* vcmpequb*x*

Vector Compare Equal-to Unsigned Byte

vcmpequb	**v**D,**v**A,**v**B	Form: VXR
vcmpequb.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	6
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 8
  if (vA)i:i+7 =int (vB)i:i+7
  then vDi:i+7 ← 81
  else vDi:i+7 ← 80
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR[24:27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpequb** is a byte.

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0

Note that **vcmpequb**[.] can be used for unsigned or signed integers.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc = 1)

Figure 6-31 shows the usage of the **vcmpequb** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

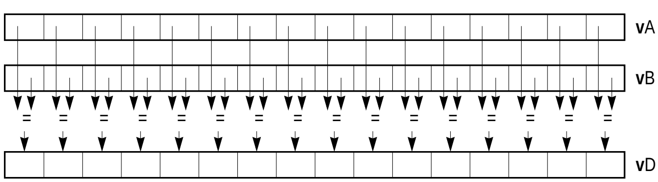


Figure 6-31. vcmpequb—Compare Equal of Sixteen Integer Elements (8-bits)

vcmpequhx vcmpequhx

Vector Compare Equal-to Unsigned Half Word

vcmpequh **v**D,**v**A,**v**B Form: VXR
vcmpequh. **v**D,**v**A,**v**B

04	vD	vA	vB	Rc	70
0–5	6–10	11–15	16–20	21	22–31

```
do i=0 to 127 by 16
  if (vA)i:i+15 =int (vB)i:i+15
  then vDi:i+15 ← 161
  else vDi:i+15 ← 160
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR[24:27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpequh** is a half word.

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0.

Note that **vcmpequh**[**.**] can be used for unsigned or signed integers.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc = 1)

Figure 6-32 shows the usage of the **vcmpequh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

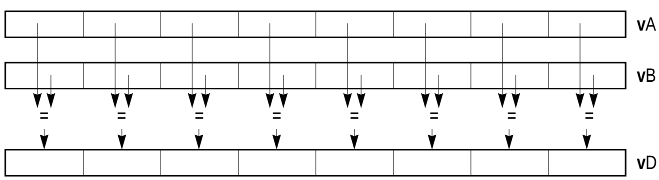


Figure 6-32. vcmpequh—Compare Equal of Eight Integer Elements (16-Bit)

vcmpequw*x* vcmpequw*x*

Vector Compare Equal-to Unsigned Word

vcmpequw **v**D,**v**A,**v**B Form: VXR

vcmpequw. **v**D,**v**A,**v**B

04	**v**D	**v**A	**v**B	Rc	134
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
  if (vA)i:i+311 =int (vB)i:i+31
      then vDi:i+31 ← n1
      else vDi:i+31 ← n0
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR[24:27] ← t ‖ 0b0 ‖ f ‖ 0b0
end
```

Each element of **vcmpequw** is a word.

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal.

- CR6 = all_equal || 0b0 || none_equal || 0b0

Note that **vcmpequw**[.] can be used for unsigned or signed integers.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-33 shows the usage of the **vcmpequw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

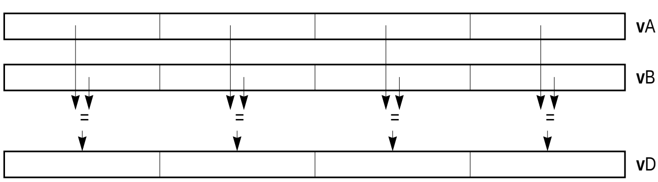


Figure 6-33. vcmpequw—Compare Equal of Four Integer Elements (32-Bit)

vcmpgefp*x* vcmpgefp*x*

Vector Compare Greater-Than-or-Equal-to Floating Point

vcmpgefp	**v**D,**v**A,**v**B	(Rc = 0)	Form: VXR
vcmpgefp.	**v**D,**v**A,**v**B	(Rc = 1)	

04	**v**D	**v**A	**v**B	Rc	454
0 – 5	6 – 10	11 – 15	16 – 20	21	22 – 31

```
do i=0 to 127 by 32
  if (vA)i:i+31 ≥fp (vB)i:i+31
  then vDi:i+31 ← 0xFFFF_FFFF
  else vDi:i+31 ← 0x0000_0000
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR24:27 ← t || 0b0 || f || 0b0
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is greater than or equal to the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_or_equal || some_greater_or_equal || none_great_or_equal.

CR6 = all_greater_or_equal || 0b0 || none greater_or_equal || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-17 shows the usage of the **vcmpgefp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

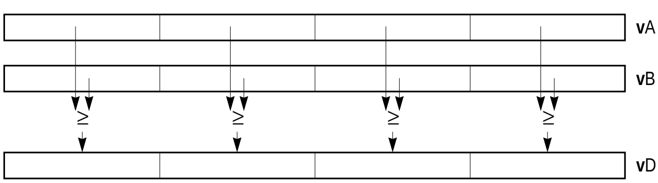


Figure 6-34. vcmpgefp—Compare Greater-Than-or-Equal of Four Floating-Point Elements (32-Bit)

vcmpgtfpx vcmpgtfpx

Vector Compare Greater-Than Floating-Point

vcmpgtfp	**v**D,**v**A,**v**B	Form: VXR
vcmpgtfp.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	710
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
  if (vA)i:i+31 >fp (vB)i:i+31
      then vDi:i+31 ← 0xFFFF_FFFF
      else vDi:i+31 ← 0x0000_0000
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR[24:27] ← t || 0b0 || f || 0b0
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_greater_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

]Figure 6-17 shows the usage of the **vcmpgtfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

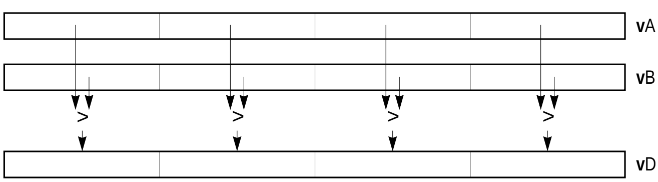


Figure 6-35. vcmpgtfp—Compare Greater-Than of Four Floating-Point Elements (32-Bit)

vcmpgtsbx vcmpgtsbx

Vector Compare Greater-Than Signed Byte

vcmpgtsb **v**D,**v**A,**v**B Form: VXR
vcmpgtsb. **v**D,**v**A,**v**B

04	vD	vA	vB	Rc	774
0–5	6–10	11–15	16–20	21	22–31

```
do i=0 to 127 by 8
  if (vA)i:i+7 >si (vB)i:i+7
      then vDi:i+7 ← 81
      else vDi:i+7 ← 80
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR24:27 ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtsb** is a byte.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-36 shows the usage of the **vcmpgtsb** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

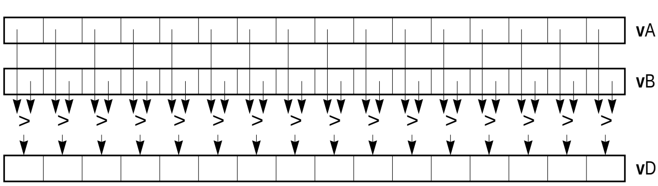


Figure 6-36. vcmpgtsb—Compare Greater-Than of Sixteen Signed Integer Elements (8-Bit)

vcmpgtsh*x* vcmpgtsh*x*

Vector Compare Greater-Than Condition Register Signed Half Word

vcmpgtsh	**v**D,**v**A,**v**B	Form: VXR
vcmpgtsh.	**v**D,**v**A,**v**B	

04	vD	vA	vB	Rc	838
0–5	6–10	11–15	16–20	21	22–31

```
do i=0 to 127 by 16
  if (vA)i:i+15 >si (vB)i:i+15
      then vDi:i+15 ← 161
      else vDi:i+15 ← 160
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR24:27 ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtsh** is a half word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-16 shows the usage of the **vcmpgtsh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

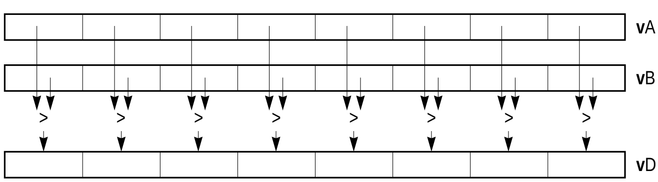


Figure 6-37. vcmpgtsh—Compare Greater-Than of Eight Signed Integer Elements (16-Bit)

vcmpgtsw*x* vcmpgtsw*x*

Vector Compare Greater-Than Signed Word

vcmpgtsw **v**D,**v**A,**v**B Form: VXR
vcmpgtsw. **v**D,**v**A,**v**B

04	**v**D	**v**A	**v**B	Rc	902
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
if (vA)i:i+31 >si (vB)i:i+31
    then vDi:i+31 ← 321
    else vDi:i+31 ← 320
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR24:27 ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtsw** is a word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-38 shows the usage of the **vcmpgtsw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

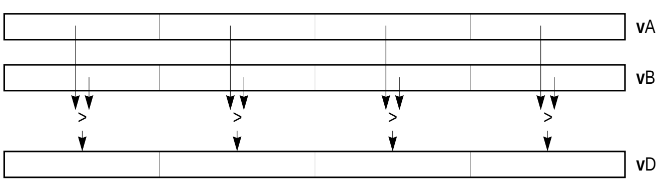


Figure 6-38. vcmpgtsw—Compare Greater-Than of Four Signed Integer Elements (32-Bit)

vcmpgtub*x* vcmpgtub*x*

Vector Compare Greater-Than Unsigned Byte

vcmpgtub	**v**D,**v**A,**v**B	Form: VXR
vcmpgtub.	**v**D,**v**A,**v**B	

04	**v**D	**v**A	**v**B	Rc	518
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 8
  if (vA)_{i:i+7} >_{ui} (vB)_{i:i+7}
      then vD_{i:i+7} ← ^8 1
      else vD_{i:i+7} ← ^8 0
end
if Rc=1 then do
  t ← (vD = ^128 1)
  f ← (vD = ^128 0)
  CR[24–27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtub** is a byte.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-14 shows the usage of the **vcmpgtub** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

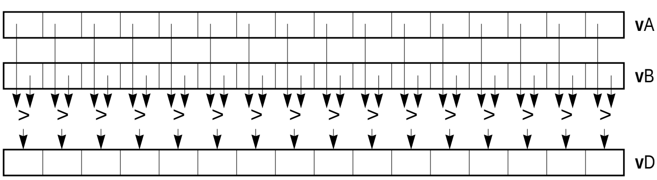


Figure 6-39. vcmpgtub—Compare Greater-Than of Sixteen Unsigned Integer Elements (8-Bit)

vcmpgtuh*x* vcmpgtuh*x*

Vector Compare Greater-Than Unsigned Half Word

vcmpgtuh **v**D,**v**A,**v**B Form: VXR

vcmpgtuh. **v**D,**v**A,**v**B

04	**v**D	**v**A	**v**B	Rc	582
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 16
  if (vA)i:i+151 >ui (vB)i:i+15
      then vDi:i+15 ← 161
      else vDi:i+15 ← 160
end
if Rc=1 then do
  t ← (vD = 1281)
  f ← (vD = 1280)
  CR[24–27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtuh** is a half word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-16 shows the usage of the **vcmpgtuh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

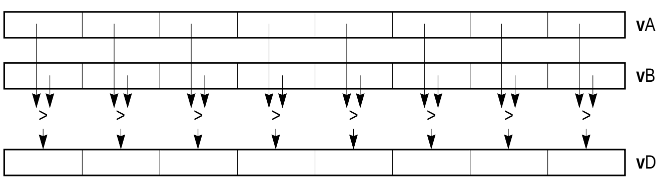


Figure 6-40. vcmpgtuh—Compare Greater-Than of Eight Unsigned Integer Elements (16-Bit)

vcmpgtuw*x* vcmpgtuw*x*

Vector Compare Greater-Than Unsigned Word

vcmpgtuw **v**D,**v**A,**v**B Form: VXR

vcmpgtuw. **v**D,**v**A,**v**B

04	**v**D	**v**A	**v**B	Rc	646
0 5	6 10	11 15	16 20	21	22 31

```
do i=0 to 127 by 32
  if (vA)i:i+31 >ui (vB)i:i+31
      then vDi:i+31 ← 32 1
      else vDi:i+31 ← 32 0
end
if Rc=1 then do
  t ← (vD = 128 1)
  f ← (vD = 128 0)
  CR[24–27] ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtuw** is a word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc = 1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6 = all_greater_than || 0b0 || none_greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0-3 (if Rc = 1)

Figure 6-41 shows the usage of the **vcmpgtuw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

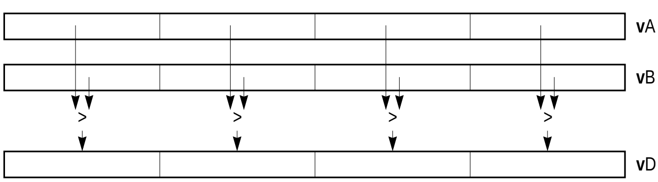


Figure 6-41. vcmpgtuw—Compare Greater-Than of Four Unsigned Integer Elements (32-Bit)

vctsxs

Vector Convert to Signed Fixed-Point Word Saturate

vctsxs **v**D,**v**B,UIMM Form: VX

04	vD	UIMM	vB	970
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
  if (vB)i+1:i+8=255 | (vB)i+1:i+8 + UIMM ≤ 254 then
      vDi:i+31 ← CnvtFP32ToSI32Sat((vB)i:i+31 *fp 2^UIMM)
   else
      do
         if (vB)i=0 then vDi:i+31 ← 0x7FFF_FFFF
         else vDi:i+31 ← 0x8000_0000
         VSCRSAT ← 1
      end
end
```

Each single-precision word element in **v**B is multiplied by 2^{UIMM}. The product is converted to a signed integer using the rounding mode, Round toward Zero.

If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1); if it is less than -2^{31} it saturates to -2^{31}. A signed-integer result is placed into the corresponding word element of **v**D.

Fixed-point integers used by the vector convert instructions can be interpreted as consisting of 32-UIMM integer bits followed by UIMM fraction bits. The vector convert to fixed-point word instructions support only the rounding mode, Round toward Zero. A single-precision number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate vector round to floating-point integer instruction before the vector convert to fixed-point word instruction.

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-42 shows the usage of the **vctsxs** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

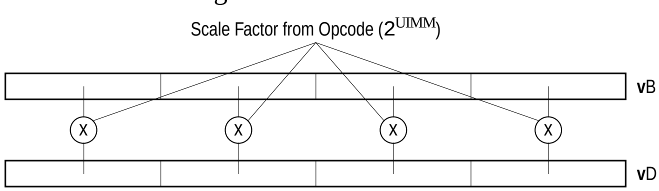


Figure 6-42. vctsxs—Convert Four Floating-Point Elements to Four Signed Integer Elements (32-Bit)

vctuxs vctuxs

Vector Convert to Unsigned Fixed-Point Word Saturate

vctuxs vD,vB,UIMM Form: VX

04	vD	UIMM	vB	906
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  if (vB)i+1:i+8=255 | (vB)i+1:i+8 + UIMM ≤ 254 then
      vDi:i+31 ← CnvtFP32ToUI32Sat((vB)i:i+31 *fp 2UIM)
   else
      do
        if (vB)i=0 thenvDi:i+31 ← 0xFFFF_FFFF
        elsevDi:i+31 ← 0x0000_0000
        VSCRSAT ← 1
     end
end
```

Each single-precision floating-point word element in **v**B is multiplied by 2^{UIM}. The product is converted to an unsigned fixed-point integer using the rounding mode Round toward Zero.

If the intermediate result is greater than $(2^{32}-1)$ it saturates to $(2^{32}-1)$ and if it is less than 0 it saturates to 0.

The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-43 shows the usage of the **vctuxs** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

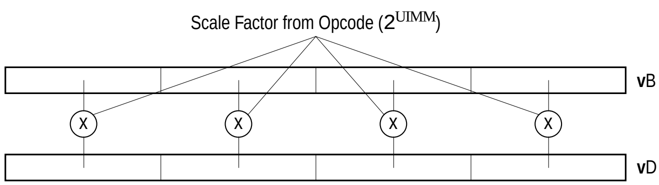


Figure 6-43. vctuxs—Convert Four Floating-Point Elements to Four Unsigned Integer Elements (32-Bit)

vexptefp vexptefp

Vector 2 Raised to the Exponent Estimate Floating Point

vexptefp **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	394
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
  x ← (vB)i:i+31
  vDi:i+31 ← 2^x
end
```

The single-precision floating-point estimate of 2 raised to the power of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

The estimate has a relative error in precision no greater than one part in 16, that is,

$$\left|\frac{\text{estimate} - 2^{x}}{2^{x}}\right| \leq \frac{1}{16}$$

where *x* is the value of the element in **v**B. The most significant 12 bits of the estimate's significant are monotonic. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

If an operation has an integral value and the resulting value is not 0 or +∞, the result is exact.

Operation with various special values of the element in **v**B is summarized below.

Value of Element in vB	Result
-∞	+0
-0	+1
+0	+1
+∞	+∞
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.

Other registers altered:

- None

Figure 6-44 shows the usage of the **vexptefp** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

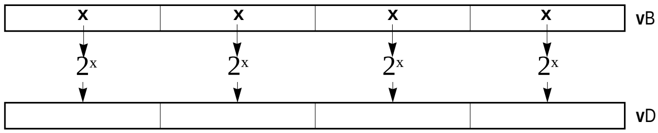


Figure 6-44. vexptefp—2 Raised to the Exponent Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vlogefp

vlogefp

Vector Log_2 Estimate Floating Point

vlogefp **v**D,**v**B Form: VX

04	vD	0 0 0 0 0	vB	458
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
  x ← (vB)i:i+31
  vDi:i+31 ← log2(x)
end
```

The single-precision floating-point estimate of the base 2 logarithm of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

The estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than 2^{-5}. The estimate has a relative error in precision no greater than one part in 8, as described below:

$$\left(|\text{estimate - } \log_2(x)| \le \frac{1}{32}\right) \qquad \text{unless} \qquad |x-1| \le \frac{1}{8}$$

where x is the value of the element in **v**B, except when $|x\text{-}1| \le 1 \div 8$. The most significant 12 bits of the estimate's significant are monotonic. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the element in **v**B is summarized below.

Value	Result
-∞	QNaN
less than 0	QNaN
±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.

Other registers altered:

- None

Figure 6-44 shows the usage of the **vexptefp** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

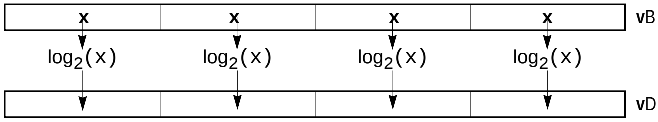


Figure 6-45. vexptefp—Log$_2$ Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vmaddfp vmaddfp

Vector Multiply Add Floating Point

vmaddfp **v**D,**v**A,**v**C,**v**B Form: VA

04	**v**D	**v**A	**v**B	**v**C	46
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
  vDi:i+31 ← RndToNearFP32(((vA)i:i+31 *fp (vC)i:i+31) +fp (vB)i:i+31)
end
```

Each single-precision floating-point word element in **v**A is multiplied by the corresponding single-precision floating-point word element in **v**C. The corresponding single-precision floating-point word element in **v**B is added to the product. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

Note that a vector multiply floating-point instruction is not provided. The effect of such an instruction can be obtained by using **vmaddfp** with **v**B containing the value -0.0 (0x8000_0000) in each of its four single-precision floating-point word elements. (The value must be -0.0, not +0.0, in order to obtain the IEEE-conforming result of -0.0 when the result of the multiplication is -0.)

Other registers altered:

- None

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.

Figure 6-46 shows the usage of the **vmaddfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

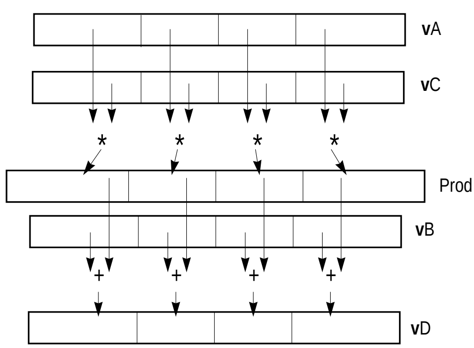


Figure 6-46. vmaddfp—Multiply-Add Four Floating-Point Elements (32-Bit)

vmaxfp vmaxfp

Vector Maximum Floating Point

vmaxfp **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1034
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  if (vA)i:i+31 ≥fp (vB)i:i+31
      then vDi:i+31 ← (vA)i:i+31
      else vDi:i+31 ← (vB)i:i+31
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The larger of the two single-precision floating-point values is placed into the corresponding word element of **v**D.

The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

Other registers altered:

- None

Figure 6-47 shows the usage of the **vmaxfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

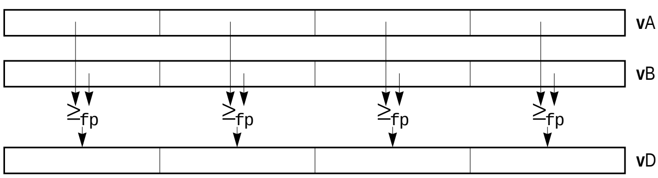


Figure 6-47. vmaxfp—Maximum of Four Floating-Point Elements (32-Bit)

vmaxsb vmaxsb

Vector Maximum Signed Byte

vmaxsb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	258
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  if (vA)i:i+7 ≥si (vB)i:i+7
      then vDi:i+7 ← (vA)i:i+7
      else vDi:i+7 ← (vB)i:i+7
end
```

Each element of **vmaxsb** is a byte.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-48 shows the usage of the **vmaxsb** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

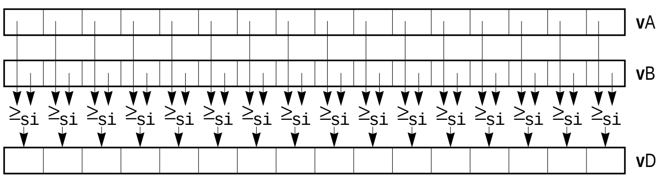


Figure 6-48. vmaxsb—Maximum of Sixteen Signed Integer Elements (8-Bit)

vmaxsh vmaxsh

Vector Maximum Signed Half Word

vmaxsh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	322
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  if (vA)i:i+7 ≥si (vB)i:i+15
      then vDi:i+15← (vA)i:i+15
      else vDi:i+15 ← (vB)i:i+15
end
```

Each element of **vmaxsh** is a half word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-49 shows the usage of the **vmaxsh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

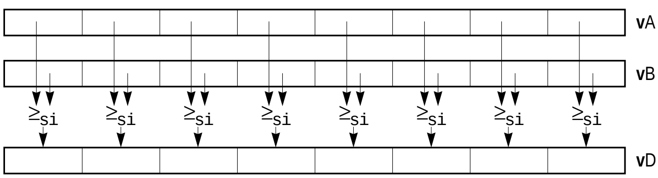


Figure 6-49. vmaxsh—Maximum of Eight Signed Integer Elements (16-Bit)

vmaxsw vmaxsw

Vector Maximum Signed Word

vmaxsw **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	386
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
  if (vA)i:i+31 ≥si (vB)i:i+31
      then vDi:i+31 ← (vA)i:i+31
      else vDi:i+31 ← (vB)i:i+31
end
```

Each element of **vmaxsw** is a word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-50 shows the usage of the **vmaxsw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

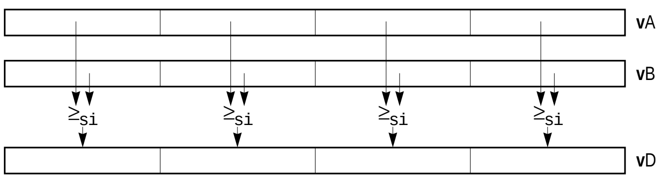


Figure 6-50. vmaxsw—Maximum of Four Signed Integer Elements (32-Bit)

vmaxub vmaxub

Vector Maximum Signed Byte

vmaxub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	2
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  if (vA)i:i+7 ≥ui (vB)i:i+7
      then vDi:i+7 ← (vA)i:i+7
      else vDi:i+7 ← (vB)i:i+7
end
```

Each element of **vmaxub** is a byte.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-48 shows the usage of the **vmaxub** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

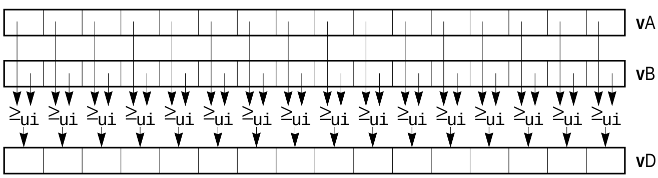


Figure 6-51. vmaxub—Maximum of Sixteen Unsigned Integer Elements (8-Bit)

vmaxuh vmaxuh

Vector Maximum Unsigned Half Word

vmaxuh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	66
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  if (vA)i:i+15 ≥ui (vB)i:i+15
      then vDi:i+15 ← (vA)i:i+15
      else vDi:i+15 ← (vB)i:i+15
end
```

Each element of **vmaxuh** is a half word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-52 shows the usage of the **vmaxuh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

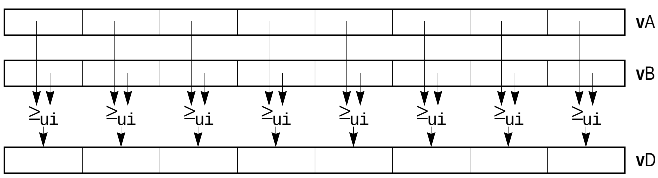


Figure 6-52. vmaxuh—Maximum of Eight Unsigned Integer Elements (16-Bit)

vmaxuw

vmaxuw

Vector Maximum Unsigned Word

vmaxuw **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	130
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
  if (vA)i:i+31 ≥ui (vB)i:i+31
      then vDi:i+31 ← (vA)i:i+31
      else vDi:i+31 ← (vB)i:i+31
end
```

Each element of **vmaxuw** is a word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-50 shows the usage of the **vmaxuw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

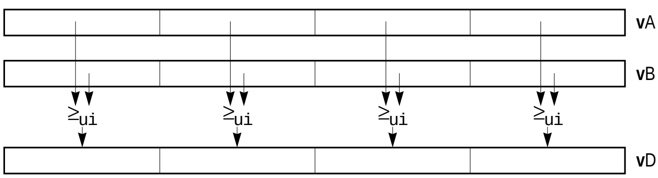


Figure 6-53. vmaxuw—Maximum of Four Unsigned Integer Elements (32-Bit)

vmhaddshs vmhaddshs

Vector Multiply High and Add Signed Half Word Saturate

vmhaddshs **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	32
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 16
  prod_{0:31}← (vA)_{i:i+15} *_{si} (vB)_{i:i+15}
   temp_{0:16}← prod_{0:16} +_{int} SignExtend((vC)_{i:i+15},17)
   vD_{i:i+15}← SItoSIsat(temp_{0:16},16)
end
```

Each signed-integer half word element in **v**A is multiplied by the corresponding signed-integer half word element in **v**B, producing a 32-bit signed-integer product. Bits 0-16 of the intermediate product are added to the corresponding signed-integer half-word element in **v**C after they have been sign extended to 17-bits. The 16-bit saturated result from each of the eight 17-bit sums is placed in register **v**D.

If the intermediate result is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than (-2^{15}) it saturates to (-2^{15}).

The signed-integer result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-16 shows the usage of the **vmhaddshs** command. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, are 16 bits in length.

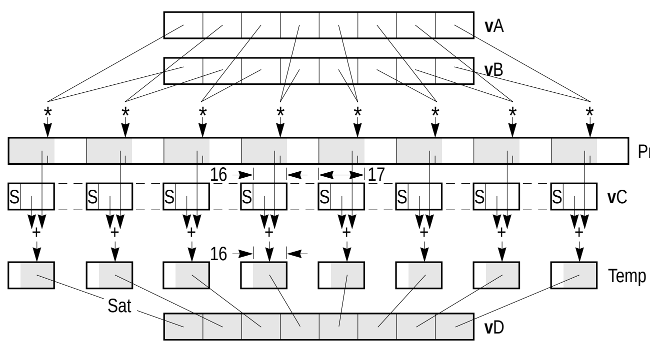


Figure 6-54. vmhaddshs—Multiply-High and Add Eight Signed Integer Elements (16-Bit)

vmhraddshs vmhraddshs

Vector Multiply High Round and Add Signed Half Word Saturate

vmhraddshs **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	33
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 16
  prod0:31 ← (vA)i:i+15 *si (vB)i:i+15
  prod0:31 ← prod0:31 +int 0x0000_4000
  temp0:16 ← prod0:16 +int SignExtend((vC)i:i+15,17)
  (vD)i:i+15 ← SItoSIsat(temp0:16,16)
end
```

Each signed integer halfword element in register **v**A is multiplied by the corresponding signed integer halfword element in register **v**B, producing a 32-bit signed integer product. The value 0x0000_4000 is added to the product, producing a 32-bit signed integer sum. Bits 0—16 of the sum are added to the corresponding signed integer halfword element in register **v**D.

If the intermediate result is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than (-2^{15}) it saturates to (-2^{15}).

The signed integer result is and placed into the corresponding halfword element of register **v**D.

Figure 6-16 shows the usage of the **vmhraddshs** command. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, are 16 bits in length.

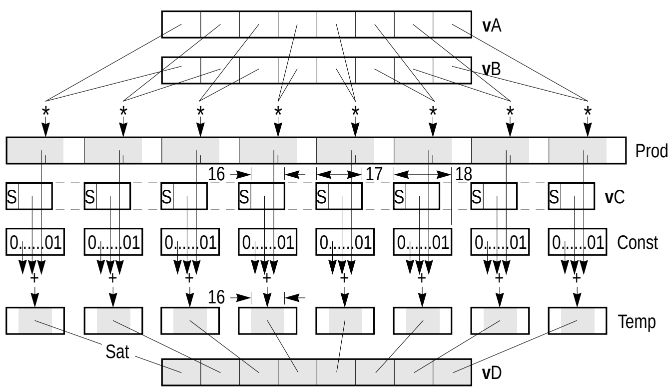


Figure 6-55. vmhraddshs—Multiply-High Round and Add Eight Signed Integer Elements (16-Bit)

vminfp

vminfp

Vector Minimum Floating Point

vminfp vD,vA,vB Form: VX

04	vD	vA	vB	1098
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  if (vA)i:i+31 <fp (vB)i:i+31
      then vDi:i+31 ← (vA)i:i+31
      else vDi:i+31 ← (vB)i:i+31
end
```

Each single-precision floating-point word element in register **v**A is compared to the corresponding single-precision floating-point word element in register **v**B. The smaller of the two single-precision floating-point values is placed into the corresponding word element of register **v**D.

The minimum of + 0.0 and - 0.0 is - 0.0. The minimum of any value and a NaN is a QNaN.

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

Figure 6-56 shows the usage of the **vminfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

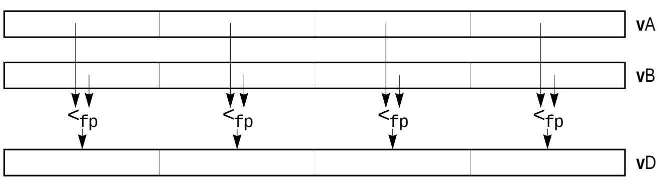


Figure 6-56. vminfp—Minimum of Four Floating-Point Elements (32-Bit)

vminsb

vminsb

Vector Minimum Signed Byte

vminsb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	770
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  if (vA)i:i+7 <si (vB)i:i+7
       then vDi:i+7 ← (vA)i:i+7
       else vDi:i+7 ← (vB)i:i+7
end
```

Each element of **vminsb** is a byte.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-48 shows the usage of the **vminsb** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

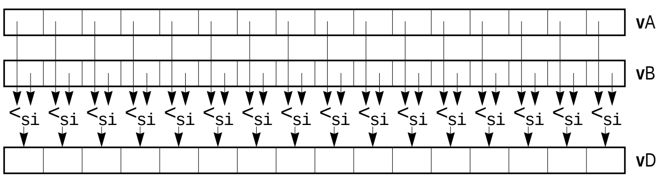


Figure 6-57. vminsb—Minimum of Sixteen Signed Integer Elements (8-Bit)

vminsh

vminsh

Vector Minimum Signed Half Word

vminsh **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	834
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  if (vA)i:i+15 <si (vB)i:i+15
      then vDi:i+15 ← (vA)i:i+15
      else vDi:i+15 ← (vB)i:i+15
end
```

Each element of **vminsh** is a half word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-49 shows the usage of the **vminsh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

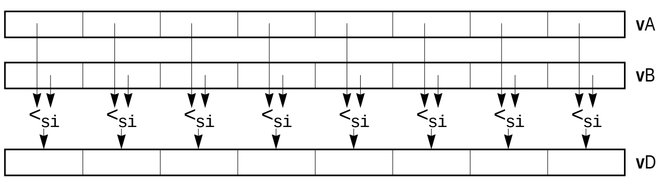


Figure 6-58. vminsh—Minimum of Eight Signed Integer Elements (16-Bit)

vminsw vminsw

Vector Minimum Signed Word

vminsw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	898
0 5	6 10	11 15	16 20	21 31

$$
\begin{array}{l}
\texttt{do i=0 to 127 by 32} \\
\quad \texttt{if } (\mathbf{v}\mathrm{A})_{i:i+31} <_{si} (\mathbf{v}\mathrm{B})_{i:i+31} \\
\qquad \texttt{then } \mathbf{v}\mathrm{D}_{i:i+31} \leftarrow (\mathbf{v}\mathrm{A})_{i:i+31} \\
\qquad \texttt{else } \mathbf{v}\mathrm{D}_{i:i+31} \leftarrow (\mathbf{v}\mathrm{B})_{\mathbf{i:i+31}} \\
\texttt{end}
\end{array}
$$

Each element of **vminsw** is a word.

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-50 shows the usage of the **vminsw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

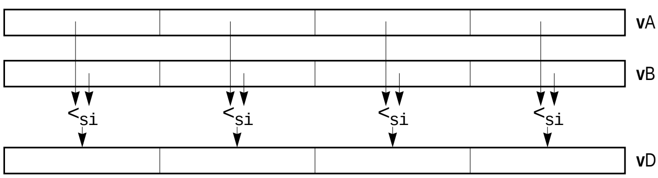


Figure 6-59. vminsw—Minimum of Four Signed Integer Elements (32-Bit)

vminub vminub

Vector Minimum Unsigned Byte

vminub **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	514
0 – 5	6 – 10	11 – 15	16 – 20	21 – 31

```
do i=0 to 127 by 8
  if (vA)i:i+7 <ui (vB)i:i+7
      then vDi:i+7 ← (vA)i:i+7
      else vDi:i+7 ← (vB)i:i+7
end
```

Each element of **vminub** is a byte.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-60 shows the usage of the **vminub** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

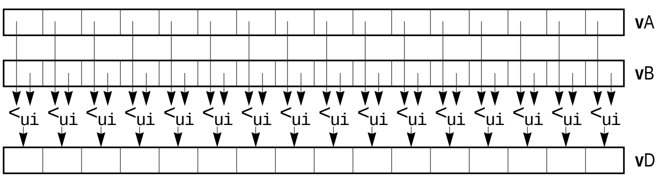


Figure 6-60. vminub—Minimum of Sixteen Unsigned Integer Elements (8-Bit)

vminuh

vminuh

Vector Minimum Unsigned Half Word

vminuh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	578
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  if (vA)i:i+15 <ui (vB)i:i+15
      then vDi:i+15 ← (vA)i:i+15
      else vDi:i+15 ← (vB)i:i+15
end
```

Each element of **vminuh** is a half word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-49 shows the usage of the **vminuh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

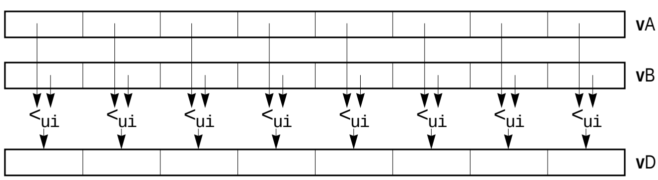


Figure 6-61. vminuh—Minimum of Eight Unsigned Integer Elements (16-Bit)

vminuw vminuw

Vector Minimum Unsigned Word

vminuw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	642
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  if (vA)i:i+31 <ui (vB)i:i+31
      then vDi:i+31 ← (vA)i:i+31
      else vDi:i+31 ← (vB)i:i+31
end
```

Each element of **vminuw** is a word.

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-50 shows the usage of the **vminuw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

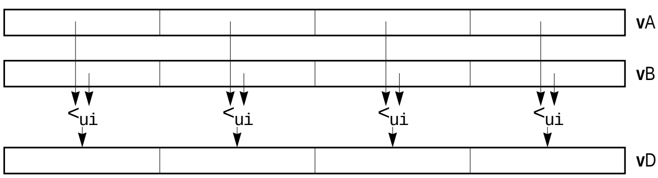


Figure 6-62. vminuw—Minimum of Four Unsigned Integer Elements (32-Bit)

vmladduhm vmladduhm

Vector Multiply Low and Add Unsigned Half Word Modulo

vmladduhm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	34
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 16
  prod0:31← (vA)i:i+15 *ui (vB)i:i+15
  vDi:i+15← prod0:31 +int (vC)i:i+15
end
```

Each integer half-word element in **v**A is multiplied by the corresponding integer half-word element in **v**B, producing a 32-bit integer product. The product is added to the corresponding integer half-word element in **v**C. The integer result is placed into the corresponding half-word element of **v**D.

Note that **vmladduhm** can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-16 shows the usage of the **vmladduhm** command. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, are 16 bits in length.

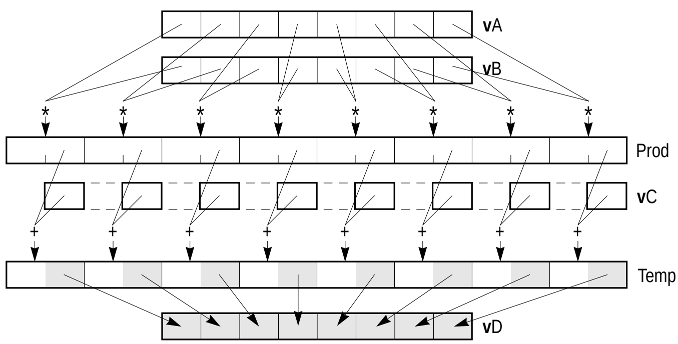


Figure 6-63. vmladduhm—Multiply-Add of Eight Integer Elements (16-Bit)

vmrghb

vmrghb

Vector Merge High Byte

vmrghb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	12
0 5	6 10	11 15	16 20	21 31

do i=0 to 63 by 8

$\mathbf{v}D_{i*2:(i*2)+15} \leftarrow (\mathbf{v}A)_{i:i+7} \parallel (\mathbf{v}B)_{i:i+7}$

end

Each element of **vmrghb** is a byte.

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-64 shows the usage of the **vmrghb** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

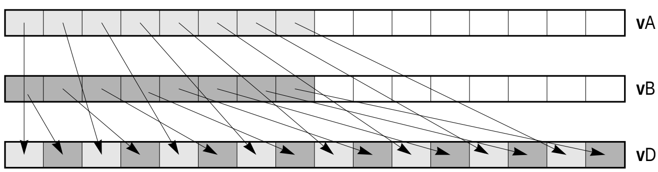


Figure 6-64. vmrghb—Merge Eight High-Order Elements (8-Bit)

vmrghh

vmrghh

Vector Merge High Half word

vmrghh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	76
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
  vD_{i*2:(i*2)+31} ← (vA)_{i:i+15} ‖ (vB)_{i:i+15}
end
```

Each element of **vmrghh** is a half word.

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-65 shows the usage of the **vmrghh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

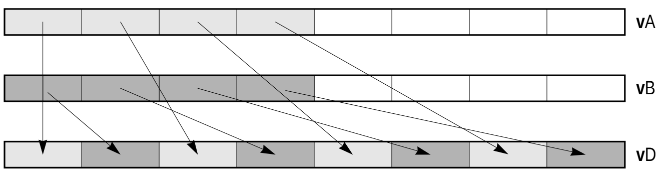


Figure 6-65. vmrghh—Merge Four High-Order Elements (16-Bit)

vmrghw

vmrghw

Vector Merge High Word

vmrghw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	140
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 32
  vD_{i*2:(i*2)+63} ← (vA)_{i:i+31} ‖ (vB)_{i:i+31}
end
```

Each element of **vmrghw** is a word.

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-66 shows the usage of the **vmrghw** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

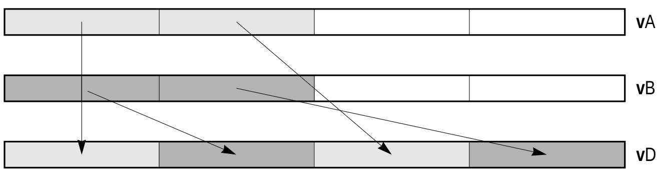


Figure 6-66. vmrghw—Merge Four High-Order Elements (32-Bit)

vmrglb vmrglb

Vector Merge Low Byte

vmrglb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	268
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 63 by 8
  vD_{i*2:(i*2)+15} ← (vA)_{i+64:i+71} ‖ (vB)_{i+64:i+71}
end
```

Each element offer **vmrglb** is a byte.

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-67 shows the usage of the **vmrglb** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

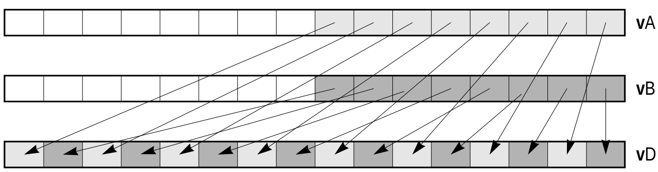


Figure 6-67. vmrglb—Merge Eight Low-Order Elements (8-Bit)

vmrglh

vmrglh

Vector Merge Low Half Word

vmrglh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	332
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
  vDi*2:(i*2)+31 ← (vA)i+64:i+79 ‖ (vB)i+64:i+79
end
```

Each element of **vmrglh** is a half word.

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-65 shows the usage of the **vmrglh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

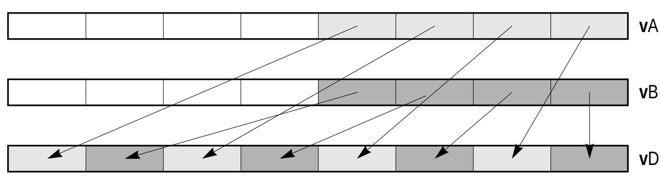


Figure 6-68. vmrglh—Merge Four Low-Order Elements (16-Bit)

vmrglw

Vector Merge Low Word

vmrglw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	396
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 32
  vDi*2:(i*2)+63 ← (vA)i+64:i+95 || (vB)i+64:i+95
end
```

Each element of **vmrglw** is a word.

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-69 shows the usage of the **vmrglw** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, are 16 bits in length.

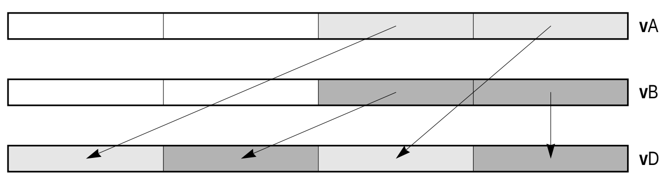


Figure 6-69. vmrglw—Merge Four Low-Order Elements (32-Bit)

vmsummbm vmsummbm

Vector Multiply Sum Mixed-Sign Byte Modulo

vmsummbm **v**D,**v**A,**v**B,**v**C Form: VA

04	vD	vA	vB	vC	37
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
  temp0:31 ← (vC)i:i+31
   do j=0 to 31 by 8
    prod0:15 ← (vA)i+j:i+j+7 *sui (vB)i+j:i+j+7
    temp0:31 ← temp0:31 +int SignExtend(prod0:15,32)
    end
  vDi:i+31 ← temp0:31
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the four signed-integer byte elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B, producing a signed-integer 16-bit product.
- The signed-integer modulo sum of these four products is added to the signed-integer word element in **v**C.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-70 shows the usage of the **vmsummbm** command. Each of the sixteen elements in the vectors, **v**A, and **v**B, are 8 bits in length. Each of the four elements in the vectors, **v**C and **v**D are 32 bits in length.

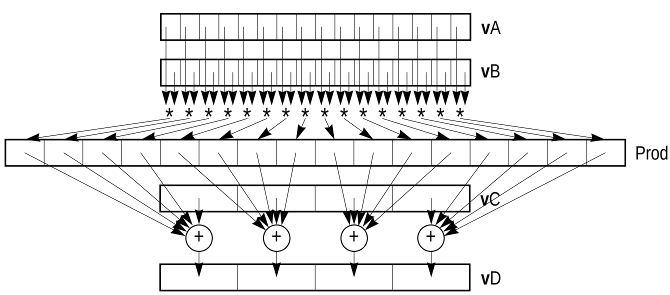


Figure 6-70. vmsummbm—Multiply-Sum of Integer Elements (8-Bit to 32-Bit)

vmsumshm vmsumshm

Vector Multiply Sum Signed Half Word Modulo

vmsumshm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	40
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
  temp0:31 ← (vC)i:i+31
   do j=0 to 31 by 16
     prod0:31 ← (vA)i+j:i+j+15 *si (vB)i+j:i+j+15
     temp0:31 ← temp0:31 +int prod0:31
     vDi:i+31 ← temp0:31
  end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two signed-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding signed-integer half-word element in **v**B, producing a signed-integer 32-bit product.
- The signed-integer modulo sum of these two products is added to the signed-integer word element in **v**C.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-71 shows the usage of the **vmsumshm** command. Each of the eight elements in the vectors, **v**A, and **v**B, are16 bits in length. Each of the four elements in the vectors, **v**C and **v**D are 32 bits in length.

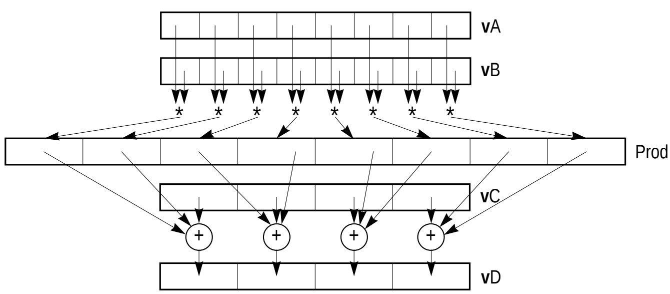


Figure 6-71. vmsumshm—Multiply-Sum of Signed Integer Elements (16-Bit to 32-Bit)

vmsumshs vmsumshs

Vector Multiply Sum Signed Half Word Saturate

vmsumshs **v**D,**v**A,**v**B,**v**C Form: VA

04	vD	vA	vB	vC	41
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
  temp0:33 ← SignExtend((vC)i:i+31,34)
   do j=0 to 31 by 16
    prod0:31 ← (vA)i+j:i+j+15 *si (vB)i+j:i+j+15
    temp0:33 ← temp0:33 +int SignExtend(prod0:31,34)
    vDi:i+31 ← SItoSIsat(temp0:33,32)
  end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two signed-integer half-word elements in the corresponding word element of **v**A is multiplied by the corresponding signed-integer half-word element in **v**B, producing a signed-integer 32-bit product.
- The signed-integer sum of these two products is added to the signed-integer word element in **v**C.
- If this intermediate result is greater than $(2^{31}-1)$ it saturates to $(2^{31}-1)$ and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-72 shows the usage of the **vmsumshs** command. Each of the eight elements in the vectors, **v**A, and **v**B, are16 bits in length. Each of the four elements in the vectors, **v**C and **v**D are 32 bits in length.

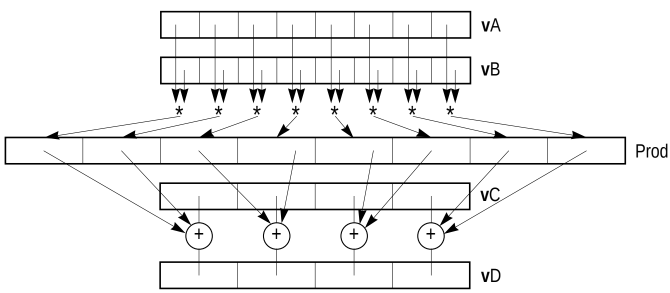


Figure 6-72. vmsumshs—Multiply-Sum of Signed Integer Elements (16-Bit to 32-Bit)

vmsumubm

vmsumubm

Vector Multiply Sum Unsigned Byte Modulo

vmsumubm **v**D,**v**A,**v**B,**v**C Form: VA

04	vD	vA	vB	vC	36
0–5	6–10	11–15	16–20	21–25	26–31

```
do i=0 to 127 by 32
  temp0:31 ← (vC)i:i+31
   do j=0 to 31 by 8
     prod0:15 ← (vA)i+j:i+j+7 *ui (vB)i+j:i+j+7
     temp0:32 ← temp0:32 +int ZeroExtend(prod0:15,32)
     vDi:i+31 ← temp0:31
  end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the four unsigned-integer byte elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B, producing an unsigned-integer 16-bit product.
- The unsigned-integer modulo sum of these four products is added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-73 shows the usage of the **vmsumubm** command. Each of the sixteen elements in the vectors, **v**A, and **v**B, are 8 bits in length. Each of the four elements in the vectors, **v**C and **v**D are 32 bits in length.

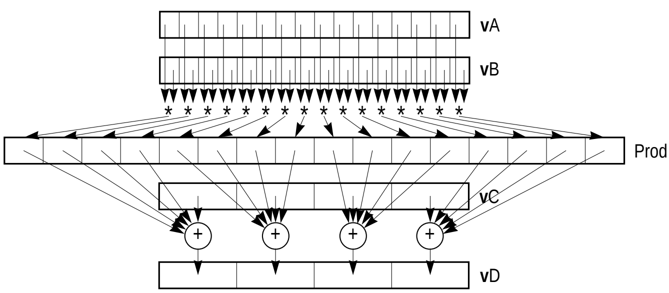


Figure 6-73. vmsumubm—Multiply-Sum of Unsigned Integer Elements (8-Bit to 32-Bit)

vmsumuhm vmsumuhm

Vector Multiply Sum Unsigned Half Word Modulo

vmsumuhm **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	A	**v**B	**v**C	38
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
  temp0:31 ← (vC)i:i+31
   do j=0 to 31 by 16
     prod0:31 ← (vA)i+j:i+j+15 *ui (vB)i+j:i+j+15
     temp0:31 ← temp0:31 +int prod0:31
     vDi:i+31 ← temp2:33
  end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two unsigned-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B, producing a unsigned-integer 32-bit product.
- The unsigned-integer sum of these two products is added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-74 shows the usage of the **vmsumuhm** command. Each of the eight elements in the vectors, **v**A, and **v**B, are16 bits in length. Each of the four elements in the vectors, **v**C and **v**D are 32 bits in length.

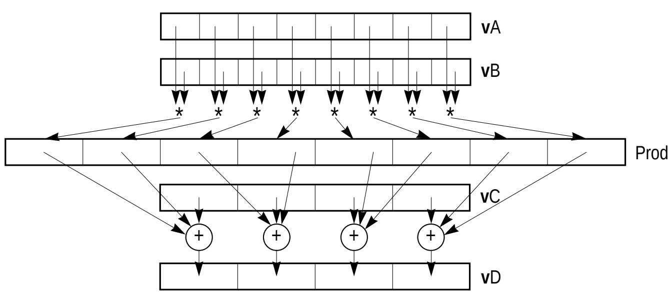


Figure 6-74. vmsumuhm—Multiply-Sum of Unsigned Integer Elements (16-Bit to 32-Bit)

vmsumuhs vmsumuhs

Vector Multiply Sum Unsigned Half Word Saturate

vmsumuhs **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	39
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
  temp_{0:33} ← ZeroExtend((vC)_{i:i+31},34)
   do j=0 to 31 by 16
     prod_{0:31} ← (vA)_{i+j:i+j+15} *_{ui} (vB)_{i+j:i+j+15}
     temp_{0:33} ← temp_{0:33} +_{int} ZeroExtend(prod_{0:31},34)
     vD_{i:i+31} ← UItoUIsat(temp_{0:33},32)
  end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two unsigned-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B, producing an unsigned-integer 32-bit product.
- The unsigned-integer sum of these two products is saturate-added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-75 shows the usage of the **vmsumuhs** command. Each of the eight elements in the vectors, **v**A, and **v**B, are16 bits in length. Each of the four elements in the vectors, **v**C and **v**D are 32 bits in length.

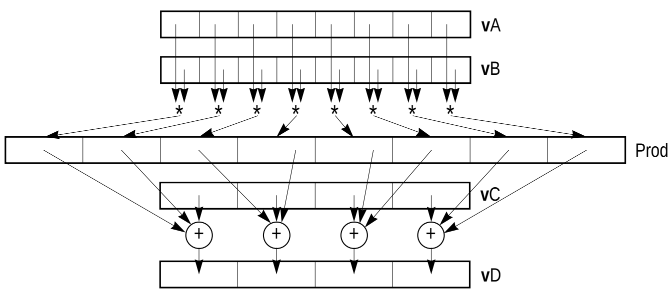


Figure 6-75. vmsumuhs—Multiply-Sum of Unsigned Integer Elements (16-Bit to 32-Bit)

vmulesb vmulesb

Vector Multiply Even Signed Byte

vmulesb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	776
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  prod0:15← (vA)i:i+7 *si (vB)i:i+7
   vDi:i+15← prod0:15
end
```

Each even-numbered signed-integer byte element in **v**A is multiplied by the corresponding signed-integer byte element in **v**B. The eight 16-bit signed-integer products are placed, in the same order, into the eight half-words of **v**D.

Other registers altered:

- None

Figure 6-76 shows the usage of the **vmulesb** command. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

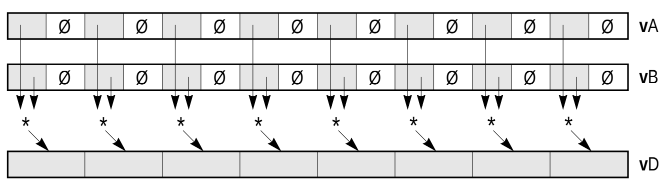


Figure 6-76. vmulesb—Even Multiply of Eight Signed Integer Elements (8-Bit)

vmulesh vmulesh

Vector Multiply Even Signed Half Word

vmulesh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	840
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  prod0:31← (vA)i:i+15 *si (vB)i:i+15
  vDi:i+31← prod0:31
end
```

Each even-numbered signed-integer half-word element in **v**A is multiplied by the corresponding signed-integer half-word element in **v**B. The four 32-bit signed-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-77 shows the usage of the **vmulesh** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the four elements in the vector **v**D, is 32 bits in length.

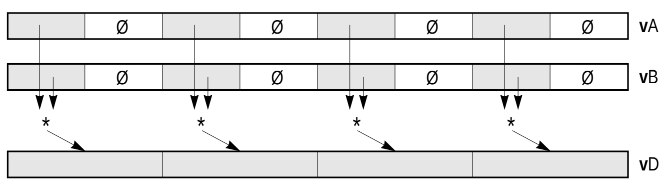


Figure 6-77. vmulesb—Even Multiply of Four Signed Integer Elements (16-Bit)

vmuleub

Vector Multiply Even Unsigned Byte

vmuleub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	520
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  prod0:15 ← (vA)i:i+7 *ui (vB)i:i+7
  (vD)i:i+15 ← prod0:15
end
```

Each even-numbered unsigned-integer byte element in register **v**A is multiplied by the corresponding unsigned-integer byte element in register **v**B. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight halfwords of register **v**D.

Other registers altered:

- None

Figure 6-78 shows the usage of the **vmuleub** command. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

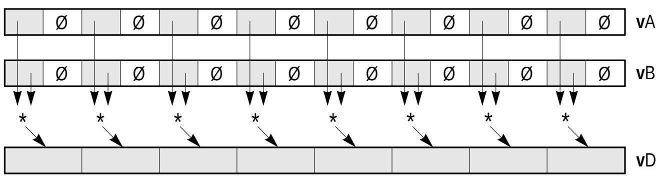


Figure 6-78. vmuleub—Even Multiply of Eight Unsigned Integer Elements (8-Bit)

vmuleuh

vmuleuh

Vector Multiply Even Unsigned Half Word

vmuleuh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	584
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  prod0:31 ← (vA)i:i+15 *ui (vB)i:i+15
  (vD)i:i+31 ← prod0:31
end
```

Each even-numbered unsigned-integer halfword element in register **v**A is multiplied by the corresponding unsigned-integer halfword element in register **v**B. The four 32-bit unsigned-integer products are placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-79 shows the usage of the **vmuleuh** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the four elements in the vector **v**D, is 32 bits in length.

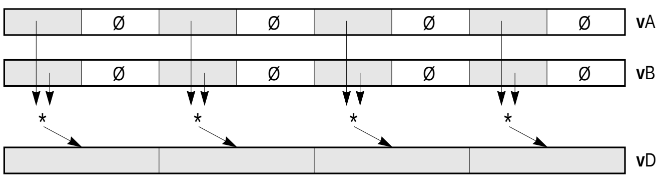


Figure 6-79. vmuleuh—Even Multiply of Four Unsigned Integer Elements (16-Bit)

vmulosb

vmulosb

Vector Multiply Odd Signed Byte

vmulosb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	264
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 16
  prod0:15← (vA)i+8:i+15 *si (vB)i+8:i+15
  vDi:i+15← prod0:15
end
```

Each odd-numbered signed-integer byte element in **v**A is multiplied by the corresponding signed-integer byte element in **v**B. The eight 16-bit signed-integer products are placed, in the same order, into the eight half-words of **v**D.

Other registers altered:

- None

Figure 6-80 shows the usage of the **vmulosb** command. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

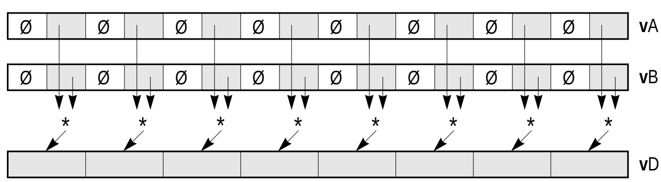


Figure 6-80. vmulosb—Odd Multiply of Eight Signed Integer Elements (8-Bit)

vmulosh

vmulosh

Vector Multiply Odd Signed Half Word

vmulosh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	328
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  prod_{0:31} ← (vA)_{i+16:i+31} *_{si} (vB)_{i+16:i+31}
  vD_{i:i+31} ← prod_{0:31}
end
```

Each odd-numbered signed-integer half-word element in **v**A is multiplied by the corresponding signed-integer half-word element in **v**B. The four 32-bit signed-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-81 shows the usage of the **vmuleuh** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the four elements in the vector **v**D, is 32 bits in length.

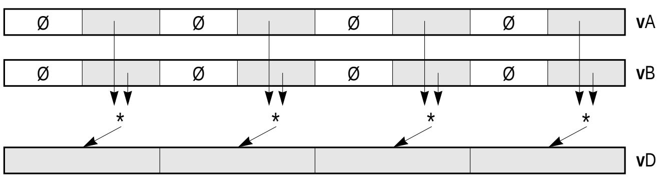


Figure 6-81. vmuleuh—Odd Multiply of Four Unsigned Integer Elements (16-Bit)

vmuloub vmuloub

Vector Multiply Odd Unsigned Byte

vmuloub **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	8
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  prod_{0:15} ← (vA)_{i+8:i+15} *ui (vB)_{i+n:i+15}
  vD_{i:i+15} ← prod_{0:15}
end
```

Each odd-numbered unsigned-integer byte element in **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight half-word s of **v**D.

Other registers altered:

- None

Figure 6-76 shows the usage of the **vmuloub** command. Each of the sixteen elements in the vectors, **v**A, and **v**B, is 8 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

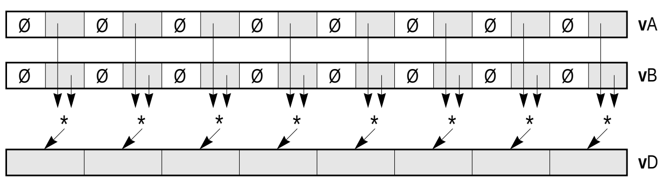


Figure 6-82. vmuloub—Odd Multiply of Eight Unsigned Integer Elements (8-Bit)

vmulouh vmulouh

Vector Multiply Odd Unsigned Half Word

vmulouh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	72
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  prod0:31← (vA)i+16:i+31 *ui (vB)i+n:i+311
  vDi:i+31← prod0:31
end
```

Each odd-numbered unsigned-integer half-word element in **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B. The four 32-bit unsigned-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-79 shows the usage of the **vmulouh** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the four elements in the vector **v**D, is 32 bits in length.

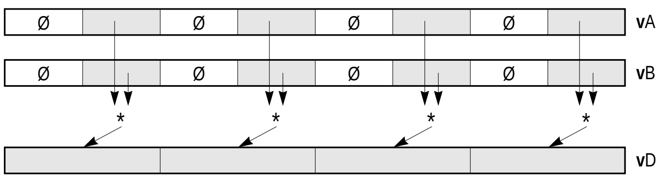


Figure 6-83. vmulouh—Odd Multiply of Four Unsigned Integer Elements (16-Bit)

vnmsubfp vnmsubfp

Vector Negative Multiply-Subtract Floating Point

vnmsubfp **v**D,**v**A,**v**C,**v**B Form: VA

04	**v**D	**v**A	**v**B	**v**C	47
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
  vD[i:i+31] ← -RndToNearFP32(((vA)[i:i+31] *fp (vC)[i:i+31]) -fp (vB)[i:i+31])
end
```

Each single-precision floating-point word element in **v**A is multiplied by the corresponding single-precision floating-point word element in **v**C. The corresponding single-precision floating-point word element in **v**B is subtracted from the product. The sign of the difference is inverted. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

Note that only one rounding occurs in this operation. Also note that a QNaN result is not negated.

Other registers altered:

- None

Figure 6-84 shows the usage of the **vnmsubfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

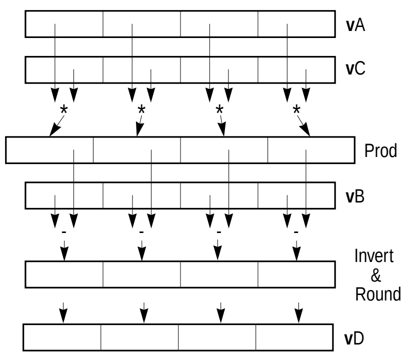


Figure 6-84. vnmsubfp—Negative Multiply-Subtract of Four Floating-Point Elements (32-Bit)

vnor vnor

Vector Logical NOR

vnor **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1284
0 5	6 10	11 15	16 20	21 31

vD ← ¬((**v**A) | (**v**B))

The contents of **v**A are bitwise ORed with the contents of **v**B and the complemented result is placed into **v**D.

Other registers altered:

- None

Simplified mnemonics:

vnot **v**D, **v**S equivalent to **vnor** **v**D, **v**S, **v**S

Figure 6-85 shows the usage of the **vnor** command.

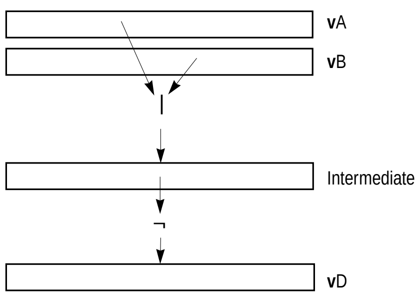


Figure 6-85. vnor—Bitwise NOR of 128-bit Vector

vor vor

Vector Logical OR

vor **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1156
0 5	6 10	11 15	16 20	21 31

vD ← (**v**A) | (**v**B)

The contents of **v**A are ORed with the contents of **v**B and the result is placed into **v**D.

Other registers altered:

- None

Simplified mnemonics:

vmr vD, **v**S equivalent to **vor v**D, **v**S, **v**S

Figure 6-85 shows the usage of the **vor** command.

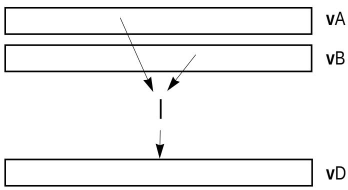


Figure 6-86. vor—Bitwise OR of 128-bit Vector

vperm vperm

Vector Permute

vperm vD,vA,vB,vC Form: VA

04	vD	vA	vB	vC	43
0–5	6–10	11–15	16–20	21–25	26–31

```
temp0:255 ← (vA) ‖ (vB)
do i=0 to 127 by 8
  b ← (vC)i+3:i+7 ‖ 0b000
  vDi:i+7 ← tempb:b+7
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B. For each integer i in the range 0–15, the contents of the byte element in the source vector specified in bits 3–7 of byte element i in **v**C are placed into byte element i of **v**D.

Other registers altered:

- None

Programming note: See the programming notes with the Load Vector for Shift Left and Load Vector for Shift Right instructions for examples of usage on the **vperm** instruction.

Figure 6-87 shows the usage of the **vperm** command. Each of the sixteen elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 8 bits in length.

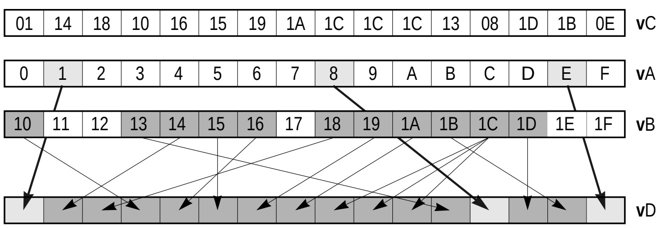


Figure 6-87. vperm—Concatenate Sixteen Integer Elements (8-Bit)

vpkpx   vpkpx

Vector Pack Pixel32

vpkpx **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	782
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
  vD_{i}         ← (vA)_{i*2+7}
  vD_{i+1:i+5}   ← (vA)_{(i*2)+8:(i*2)+12}
  vD_{i+6:i+10}  ← (vA)_{(i*2)+16:(i*2)+20}
  vD_{i+11:i+15} ← (vA)_{(i*2)+24:(i*2)+28}
  vD_{i+64}      ← (vB)_{(i*2)+7}
  vD_{i+65:i+69} ← (vB)_{(i*2)+8:(i*2)+12}
  vD_{i+70:i+74} ← (vB)_{(i*2)+16:(i*2)+20}
  vD_{i+75:i+79} ← (vB)_{(i*2)+24:(i*2)+28}
end
```

The source vector is the concatenation of the contents of **v**A followed by the contents of **v**B. Each word element in the source vector is packed to produce a 16-bit value as described below and placed into the corresponding half-word element of **v**D. A word is packed to 16 bits by concatenating, in order, the following bits.

- bit 7 of the first byte (bit 7 of the word)
- bits 0–4 of the second byte (bits 8–12 of the word)
- bits 0–4 of the third byte (bits 16–20 of the word)
- bits 0–4 of the fourth byte (bits 24–28 of the word)

Other registers altered:

- None

Programming note: Each source word can be considered to be a 32-bit pixel consisting of four 8-bit channels. Each target 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.

Figure 6-88 shows the usage of the **vpkpx** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

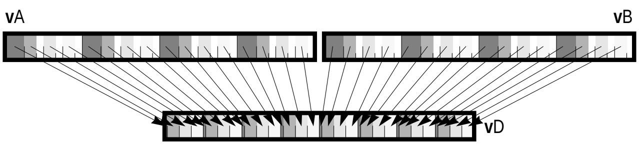


Figure 6-88. vpkpx—Pack Eight Elements (32-Bit) to Eight Elements (16-Bit)

vpkshss vpkshss

Vector Pack Signed Half Word Signed Saturate

vpkshss **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	398
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 63 by 8
  vD[i:i+7] ← SItoSIsat((vA)[i*2:(i*2)+15],8)
  vD[i+64:i+71] ← SItoSIsat((vB)[i*2:(i*2)+15],8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer half-word element in the source vector is converted to an 8-bit signed integer. If the value of the element is greater than $(2^7 - 1)$ the result saturates to $(2^7 - 1)$ and if the value is less than -2^7 the result saturates to -2^7. The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-89 shows the usage of the **vpkshss** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the sixteen elements in the vector **v**D, is 8 bits in length.

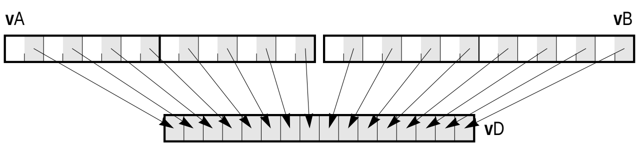


Figure 6-89. vpkshss—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Signed Integer Elements (8-Bit)

vpkshus vpkshus

Vector Pack Signed Half Word Unsigned Saturate

vpkshus **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	270
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
  vD_{i:i+7} ← SItoUIsat((vA)_{i*2:(i*2)+7},8)
  vD_{i+64:i+71} ← SItoUIsat((vB)_{i*2:(i*2)+7},8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer half-word element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than $(2^8 - 1)$ the result saturates to $(2^8 - 1)$ and if the value is less than 0 the result saturates to 0. The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-90 shows the usage of the **vpkshus** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the sixteen elements in the vector **v**D, is 8 bits in length.

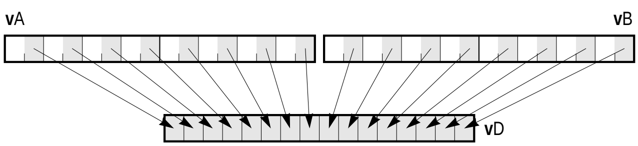


Figure 6-90. vpkshus—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkswss vpkswss

Vector Pack Signed Word Signed Saturate

vpkswss **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	462
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 63 by 16
  vDi:i+15← SItoSIsat((vA)i*2:(i*2)+31,16)
  vDi+64:i+79← SItoSIsat((vB)i*2:(i*2)+31,16)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer word element in the source vector is converted to a 16-bit signed integer half word. If the value of the element is greater than (2^{15} - 1) the result saturates to (2^{15} - 1) and if the value is less than -2^{15} the result saturates to -2^{15}. The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-91 shows the usage of the **vpkswss** command. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

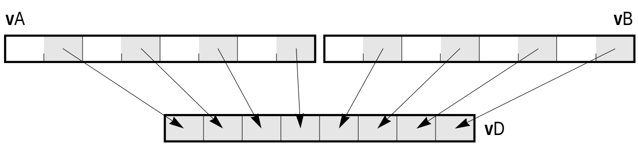


Figure 6-91. vpkswss—Pack Eight Signed Integer Elements (32-Bit) to Eight Signed Integer Elements (16-Bit)

vpkswus vpkswus

Vector Pack Signed Word Unsigned Saturate

vpkswus **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	334
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
  vDi:i+15← SItoUIsat((vA)i*2:(i*2)+31,16)
  vDi+64:i+79← SItoUIsat((vB)i*2:(i*2)+31,16)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than (2^{16} - 1) the result saturates to (2^{16} - 1) and if the value is less than 0 the result saturates to 0. The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-92 shows the usage of the **vpkswus** command. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

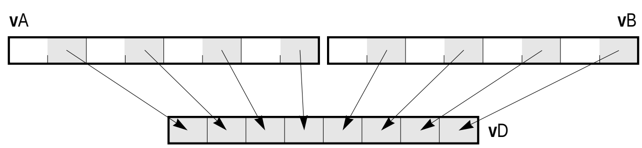


Figure 6-92. vpkswus—Pack Eight Signed Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vpkuhum

vpkuhum

Vector Pack Unsigned Half Word Unsigned Modulo

vpkuhum **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	14
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
  vDi:i+7 ← (vA)(i*2)+8:(i*2)+15
  vDi+64:i+71 ← (vB)(i*2)+8:(i*2)+15
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

The low-order byte of each half-word element in the source vector is placed into the corresponding byte element of **v**D.

Other registers altered:

- None

Figure 6-93 shows the usage of the **vpkuhum** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the sixteen elements in the vector **v**D, is 8 bits in length.

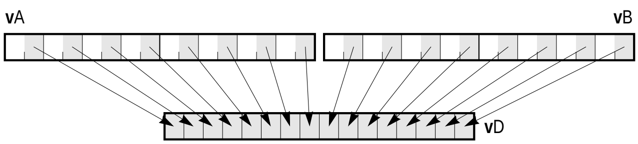


Figure 6-93. vpkuhum—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkuhus vpkuhus

Vector Pack Unsigned Half Word Unsigned Saturate

vpkuhus **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	142
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
  vD[i:i+7] ← UItoUIsat((vA)[i*2:(i*2)+15],8)
  vD[i+64:i+71] ← UItoUIsat((vB)[i*2:(i*2)+15],8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each unsigned integer half-word element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than (2^8 - 1) the result saturates to (2^8 - 1). The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-94 shows the usage of the **vpkuhus** command. Each of the eight elements in the vectors, **v**A, and **v**B, is 16 bits in length. Each of the sixteen elements in the vector **v**D, is 8 bits in length.

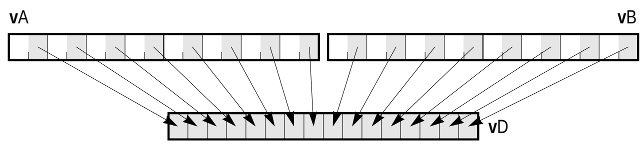


Figure 6-94. vpkuhus—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkuwum vpkuwum

Vector Pack Unsigned Word Unsigned Modulo

vpkuwum **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	78
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
  vD[i:i+15] ← (vA)[(i*2)+16:(i*2)+31]
  vD[i+64:i+79] ← (vB)[(i*2)+16:(i*2)+31]
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

The low-order half-word of each word element in the source vector is placed into the corresponding half-word element of **v**D.

Other registers altered:

- None

Figure 6-95 shows the usage of the **vpkuwum** command. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

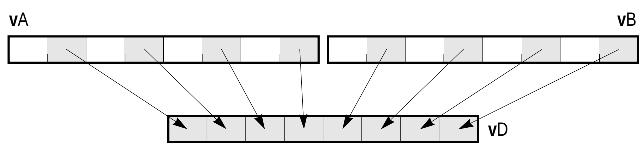


Figure 6-95. vpkuwum—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vpkuwus vpkuwus

Vector Pack Unsigned Word Unsigned Saturate

vpkuwus **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	206
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 63 by 16
  vD[i:i+15] ← UItoUIsat((vA)[i*2:(i*2)+31],16)
  vD[i+64:i+79] ← UItoUIsat((vB)[i*2:(i*2)+31],16)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each unsigned integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than $(2^{16} - 1)$ the result saturates to $(2^{16} - 1)$. The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-96 shows the usage of the **vpkuwus** command. Each of the four elements in the vectors, **v**A, and **v**B, is 32 bits in length. Each of the eight elements in the vector **v**D, is 16 bits in length.

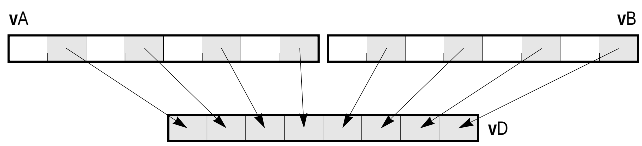


Figure 6-96. vpkuwum—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vrefp vrefp

Vector Reciprocal Estimate Floating Point

vrefp **v**D,**v**B Form: VX

04	vD	0 0 0 0 0	vB	266
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  x ← (vB)i:i+31
  vDi:i+31 ← 1/x
end
```

The single-precision floating-point estimate of the reciprocal of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

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 **v**B. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the element in **v**B is summarized below.

Value	Result
-∞	-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.

Other registers altered:

- None

Figure 6-97 shows the usage of the **vrefp** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

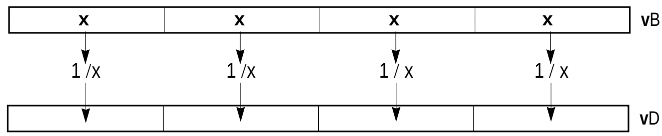


Figure 6-97. vrefp—Reciprocal Estimate of Four Floating-Point Elements (32-Bit)

vrfim vrfim

Vector Round to Floating-Point Integer toward Minus Infinity

vrfim **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	714
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vD[i:i+31] ← RndToFPInt32Floor((vB)[i:i+31])
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round toward -Infinity, and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-98 shows the usage of the **vrfim** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

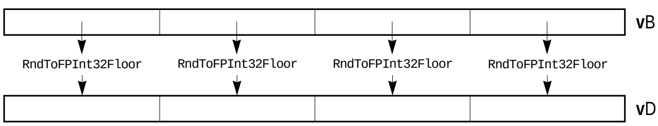


Figure 6-98. vrfim— Round to Minus Infinity of Four Floating-Point Integer Elements (32-Bit)

vrfin vrfin

Vector Round to Floating-Point Integer Nearest

vrfin **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	522
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vDi:i+31 ← RndToFPInt32Near((vB)i:i+31)
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round to Nearest, and placed into the corresponding word element of **v**D.

Note the result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-99 shows the usage of the **vrfin** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

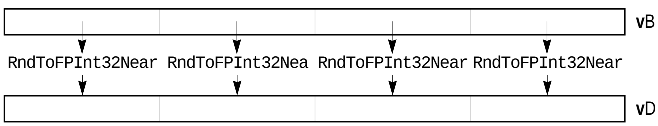


Figure 6-99. vrfin—Nearest Round to Nearest of Four Floating-Point Integer Elements (32-Bit)

vrfip vrfip

Vector Round to Floating-Point Integer toward Plus Infinity

vrfip **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	650
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vDi:i+31 ← RndToFPInt32Ceil((vB)i:i+31)
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round toward +Infinity, and placed into the corresponding word element of **v**D.

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

Other registers altered:

- None

Figure 6-100 shows the usage of the **vrfip** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

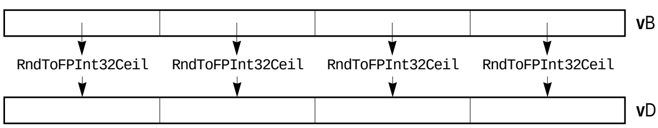


Figure 6-100. vrfip—Round to Plus Infinity of Four Floating-Point Integer Elements (32-Bit)

vrfiz vrfiz

Vector Round to Floating-Point Integer toward Zero

vrfiz **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	586
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vD_{i:i+31} ← RndToFPInt32Trunc((vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode Round toward Zero, and placed into the corresponding word element of **v**D.

Note, the result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-101 shows the usage of the **vrfiz** command. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

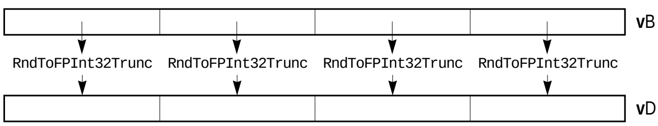


Figure 6-101. vrfiz—Round-to-Zero of Four Floating-Point Integer Elements (32-Bit)

vrlb vrlb

Vector Rotate Left Integer Byte

vrlb **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	4
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  sh ← (vB)i+5:i+7
  vDi:i+7 ← ROTL((vA)i:i+7,sh)
end
```

Each element is a byte. Each element in **v**A is rotated left by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-102 shows the usage of the **vrlb** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

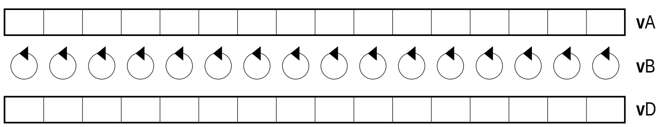


Figure 6-102. vrlb—Left Rotate of Sixteen Integer Elements (8-Bit)

vrlh vrlh

Vector Rotate Left Integer Half Word

vrlh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	68
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  vDi:i+15 ← ROTL((vA)i:i+15,sh)
end
```

Each element is a half word

Each element in **v**A is rotated left by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-103 shows the usage of the **vrlh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits in length.

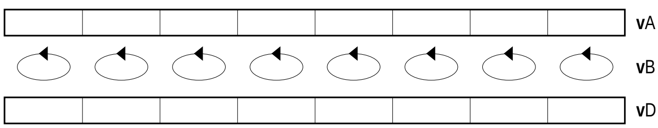


Figure 6-103. vrlh—Left Rotate of Eight Integer Elements (16-Bit)

vrlw vrlw

Vector Rotate Left Integer Word

vrlw **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	132
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  sh ← (vB)i+27:i+31
  vDi:i+31 ← ROTL((vA)i:i+31,sh)
end
```

Each element is a word. Each element in **v**A is rotated left by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-104 shows the usage of the **vrlw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

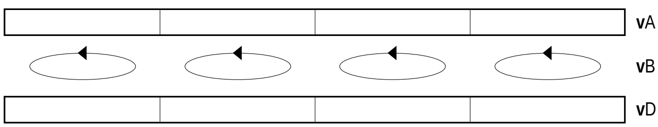


Figure 6-104. vrlw—Left Rotate of Four Integer Elements (32-Bit)

vrsqrtefp vrsqrtefp

Vector Reciprocal Square Root Estimate Floating Point

vrsqrtefp **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	330
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  x ← (vB)i:i+31
  vDi:i+31 ← 1 ÷fp (√fp(x))
end
```

The single-precision estimate of the reciprocal of the square root of each single-precision element in **v**B is placed into the corresponding word element of **v**D. The estimate has a relative error in precision no greater than one part in 4096, as explained below:

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

where x is the value of the element in **v**B. Note that the value placed into the element of **v**D may vary between implementations and between different executions on the same implementation. Operation with various special values of the element in **v**B is summarized below.

Value	Result
-∞	QNaN
less than 0	QNaN
-0	-∞
+0	+∞
+∞	+0
NaN	QNaN

Other registers altered:

- None

Figure 6-105 shows the usage of the **vrsqrtefp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

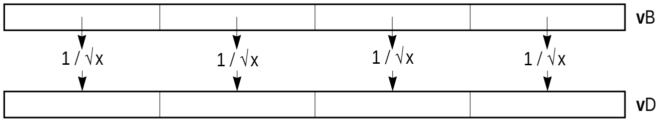


Figure 6-105. vrsqrtefp—Reciprocal Square Root Estimate of Four Floating-Point Elements (32-Bit)

vsel vsel

Vector Conditional Select

vsel **v**D,**v**A,**v**B,**v**C Form: VA

04	**v**D	**v**A	**v**B	**v**C	42
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127
  if (vC)i=0 then vDi ← (vA)i
   else  vDi ← (vB)i
end
```

For each bit in **v**C that contains the value 0, the corresponding bit in **v**A is placed into the corresponding bit of **v**D. For each bit in **v**C that contains the value 1, the corresponding bit in **v**B is placed into the corresponding bit of **v**D.

Other registers altered:

- None

Figure 6-106 shows the usage of the **vsel** command. Each of the vectors, **v**A, **v**B, **v**C, and **v**D, is 128 bits in length.

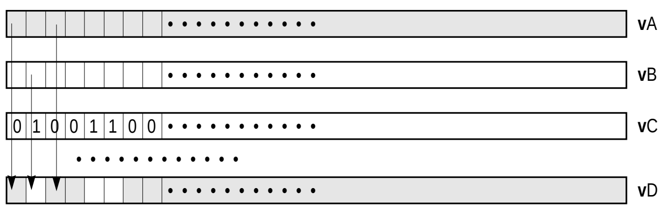


Figure 6-106. vsel—Bitwise Conditional Select of Vector Contents(128-bit)

vsl vsl

Vector Shift Left

vsl **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	452
0 5	6 10	11 15	16 20	21 31

```
sh ← (vB)125:127
t ← 1
do i = 0 to 127 by 8
  t ← t & ((vB)i+5:i+7 = sh)
  if t = 1 then vD ← (vA) <<ui sh
  else   vD ← undefined
end
```

The contents of **v**A are shifted left by the number of bits specified in **v**B[125–127]. Bits shifted out of bit 0 are lost. Zeros are supplied to the vacated bits on the right. The result is placed into **v**D.

The contents of the low-order three bits of all byte elements in **v**B must be identical to **v**B[125–127]; otherwise the value placed into **v**D is undefined.

Other registers altered:

- None

Figure 6-107 shows the usage of the **vsl** command.

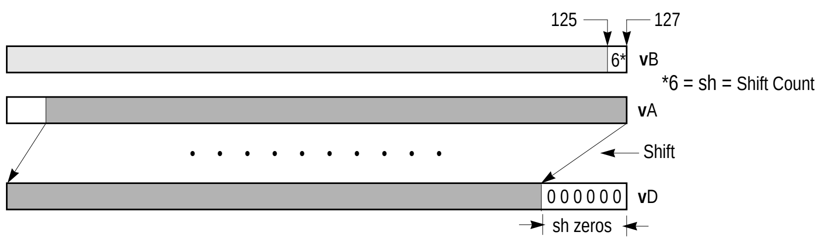


Figure 6-107. vsl—Shift Bits Left in Vector (128-Bit)

vslb vslb

Vector Shift Left Integer Byte

vslb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	260
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  sh ← (vB)i+5):i+7
  vDi:i+7 ← (vA)i:i+7 <<ui sh
end
```

Each element is a byte. Each element in **v**A is shifted left by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-102 shows the usage of the **vslb** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

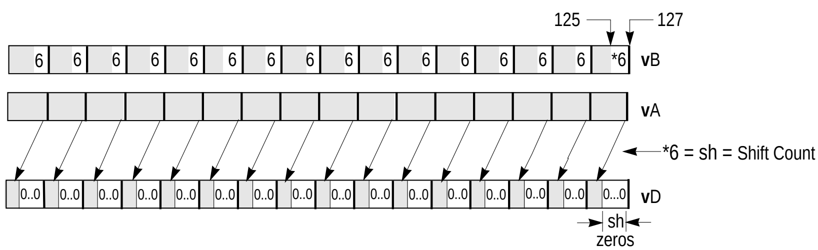


Figure 6-108. vslb—Shift Bits Left in Sixteen Integer Elements (8-Bit)

vsldoi

vsldoi

Vector Shift Left Double by Octet Immediate

vsldoi **v**D, **v**A, **v**B, SHB Form: VA

04	**v**D	**v**A	**v**B	0	SH	44
0 5	6 10	11 15	16 20	21	22 25	26 31

vD ← ((**v**A) ‖ (**v**B)) $<<_{ui}$ (SHB ‖ 0b000)

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B. Bytes SHB:SHB+15 of the source vector are placed into **v**D.

Other registers altered:

- None

Figure 6-14 shows the usage of the **vsldoi** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

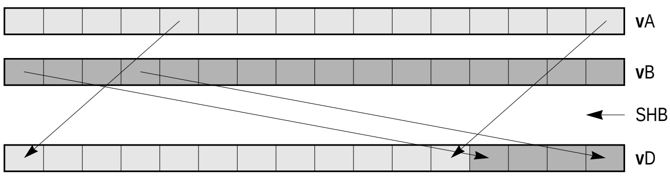


Figure 6-109. vsldoi—Shift Left by Bytes Specified

vslh

vslh

Vector Shift Left Integer Half Word

vslh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	324
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  vDi:i+15 ← (vA)i:i+15 <<ui sh
end
```

Each element is a half word. Each element in **v**A is shifted left by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-16 shows the usage of the **vslh** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits in length.

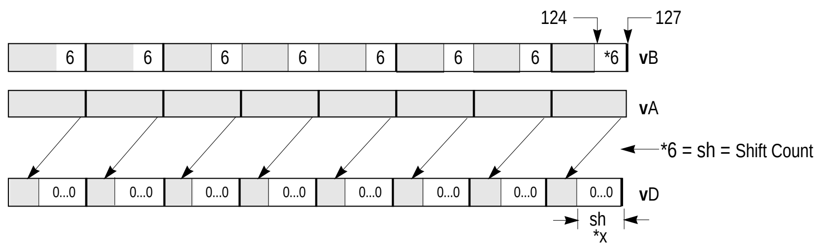


Figure 6-110. vslh—Shift Bits Left in Eight Integer Elements (16-Bit)

vslo **vslo**

Vector Shift Left by Octet

vslo **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	1036
0 5	6 10	11 15	16 20	21 31

```
shb ← (vB)121:124
vD  ← (vA) <<ui (shb || 0b000)
```

The contents of **v**A are shifted left by the number of bytes specified in **v**B[121–124]. Bytes shifted out of byte 0 are lost. Zeros are supplied to the vacated bytes on the right. The result is placed into **v**D.

Other registers altered:

- None

Figure 6-111 shows the usage of the **vslo** command.

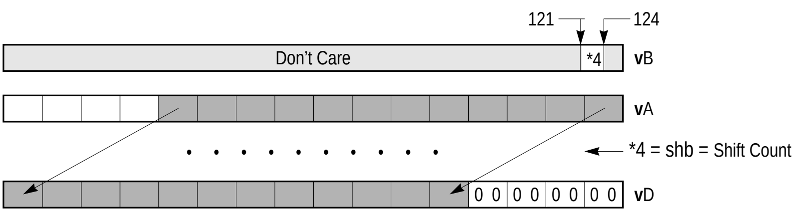


Figure 6-111. vslo—Left Byte Shift of Vector (128-Bit)

vslw vslw

Vector Shift Left Integer Word

vslw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	388
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  sh ← (vB)i+27:i+31
  vDi:i+31 ← (vA)i:i+31 <<ui sh
end
```

Each element is a word. Each element in **v**A is shifted left by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-112 shows the usage of the **vslw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

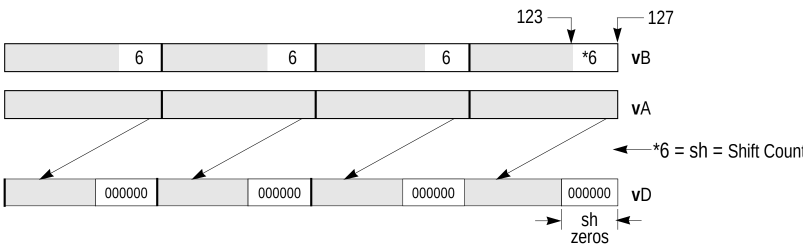


Figure 6-112. vslw—Shift Bits Left in Four Integer Elements (32-Bit)

vspltb vspltb

Vector Splat Byte

vspltb **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	524
0 5	6 10	11 15	16 20	21 31

```
b ← UIMM*8
do i=0 to 127 by 8
  vD[i:i+7] ← (vB)[b:b+7]
end
```

Each element of **vspltb** is a byte.

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming note: The vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a vector register by a constant).

Figure 6-113 shows the usage of the **vspltb** command. Each of the sixteen elements in the vectors **v**B and **v**D is 8 bits in length.

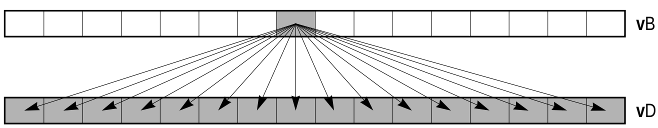


Figure 6-113. vspltb—Copy Contents to Sixteen Elements (8-Bit)

vspltth vspltth

Vector Splat Half Word

vspltth **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	588
0–5	6–10	11–15	16–20	21–31

```
b ← UIMM*16
do i=0 to 127 by 16
  vD[i:i+15] ← (vB)[b:b+15]
end
```

Each element of **vspltth** is a half word.

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming note: The vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a vector register by a constant).

Figure 6-16 shows the usage of the **vspltth** command. Each of the eight elements in the vectors **v**B and **v**D is 16 bits in length.

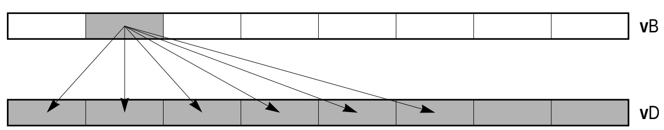


Figure 6-114. vspltth—Copy Contents to Eight Elements (16-Bit)

vspltisb vspltisb

Vector Splat Immediate Signed Byte

vspltisb **v**D,SIMM Form: VX

04	**v**D	SIMM	0 0 0 0 0	780
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  vDi:i+7 ← SignExtend(SIMM,8)
end
```

Each element of **vspltisb** is a byte.

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-115 shows the usage of the **vspltisb** command. Each of the sixteen elements in the vector, **v**D, is 8 bits in length.

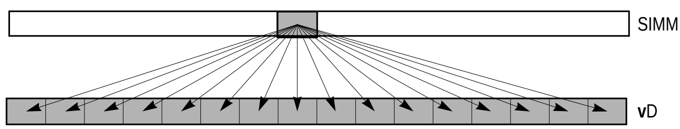


Figure 6-115. vspltisb—Copy Value into Sixteen Signed Integer Elements (8-Bit)

vspltish vspltish

Vector Splat Immediate Signed Half Word

vspltish **v**D,SIMM Form: VX

04	**v**D	SIMM	0 0 0 0 0	844
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  vD[i:i+15] ← SignExtend(SIMM,16)
end
```

Each element of **vspltish** is a half word.

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-16 shows the usage of the **vspltish** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits in length.

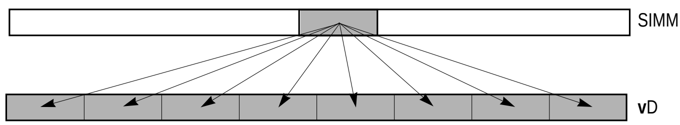


Figure 6-116. vspltish—Copy Value to Eight Signed Integer Elements (16-Bit)

vspltisw vspltisw

Vector Splat Immediate Signed Word

vspltisw **v**D,SIMM Form: VX

04	**v**D	SIMM	0 0 0 0 0	908
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vDi:i+31 ← SignExtend(SIMM,32)
end
```

Each element of **vspltisw** is a word.

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-117 shows the usage of the **vspltisw** command. Each of the four elements in the vector, and **v**D, is 32 bits in length.

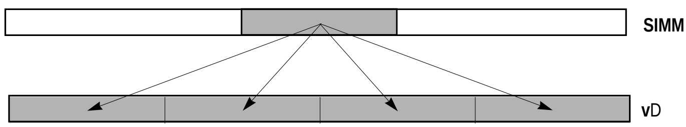


Figure 6-117. vspltisw—Copy Value to Four Signed Elements (32-Bit)

vspltw

vspltw

Vector Splat Word

vspltw **v**D,**v**B,UIMM Form: VX

04	**v**D	UIMM	**v**B	652
0 5	6 10	11 15	16 20	21 31

```
b ← UIMM*32
do i=0 to 127 by 32
  vD[i:i+31] ← (vB)[b:b+31]
end
```

Each element of **vspltw** is a word.

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming note: The Vector Splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a Vector Register by a constant).

Figure 6-118 shows the usage of the **vspltw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

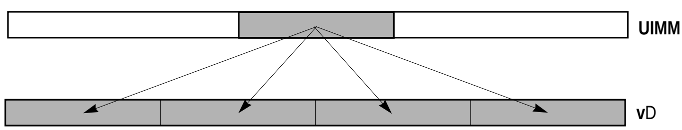


Figure 6-118. vspltw—Copy contents to Four Elements (32-Bit)

vsr vsr

Vector Shift Right

vsr **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	708
0–5	6–10	11–15	16–20	21–31

```
sh ← (vB)125:127
t ← 1
do i = 0 to 127 by 8
  t ← t & ((vB)i+5:i+7 = sh)
  if t = 1 then vD ← (vA) >>ui sh
  else   vD ← undefined
end
```

Let sh = **v**B[125–127]; sh is the shift count in bits (0≤sh≤7). The contents of **v**A are shifted right by sh bits. Bits shifted out of bit 127 are lost. Zeros are supplied to the vacated bits on the left. The result is placed into **v**D.

The contents of the low-order three bits of all byte elements in register **v**B must be identical to **v**B[125-127]; otherwise the value placed into register **v**D is undefined.

Other registers altered:

- None

Programming notes:

A pair of **vslo** and **vsl** or **vsro** and **vsr** instructions, specifying the same shift count register, can be used to shift the contents of a vector register left or right by the number of bits (0–127) specified in the shift count register. The following example shifts the contents of **v**X left by the number of bits specified in **v**Y and places the result into **v**Z.

```
vslo     VZ,VX,VY
vsl      VZ,VZ,VY
```

A double-register shift by a dynamically specified number of bits (0–127) can be performed in six instructions. The following example shifts (**v**W) || (**v**X) left by the number of bits specified in **v**Y and places the high-order 128 bits of the result into **v**Z.

```
vslo     t1,VW,VY  #shift high-order reg left
vsl      t1,t1,VY
vsububm  t3,V0,VY  #adjust shift count ((V0)=0)
vsro     t2,VX,t3  #shift low-order reg right
vsr      t2,t2,t3
vor      VZ,t1,t2  #merge to get final result
```

Figure 6-119 shows the usage of the **vsr** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

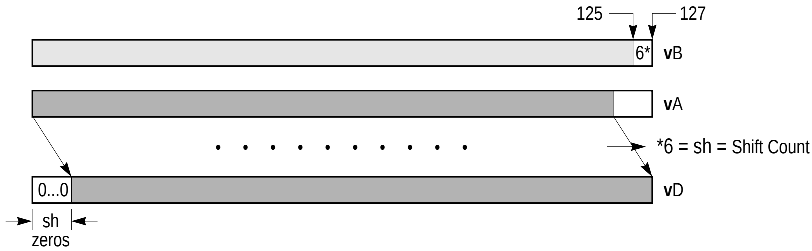


Figure 6-119. vsr—Shift Bits Right for Vectors (128-Bit)

vsrab

vsrab

Vector Shift Right Algebraic Byte

vsrab **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	772
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  sh ← (vB)i+2:i+7
  vDi:i+7 ← (vA)i:i+7 >>si sh
end
```

Each element is a byte. Each element in **v**A is shifted right by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit n-1 of the element are lost. Bit 0 of the element is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-120 shows the usage of the **vsrab** command. Each of the sixteen elements in the vectors, **v**A, and **v**D, is 8 bits in length.

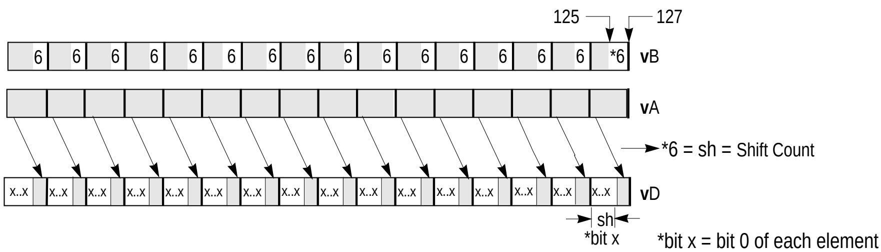


Figure 6-120. vsrab—Shift Bits Right in Sixteen Integer Elements (8-Bit)

vsrah vsrah

Vector Shift Right Algebraic Half Word

vsrah **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	836
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  vDi:i+15 ← (vA)i:i+15 >>si sh
end
```

Each element is a half word. Each element in **v**A is shifted right by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 15 of the element are lost. Bit 0 of the element is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-121 shows the usage of the **vsrah** command. Each of the eight elements in the vectors, **v**A, and **v**D, is 16 bits in length.

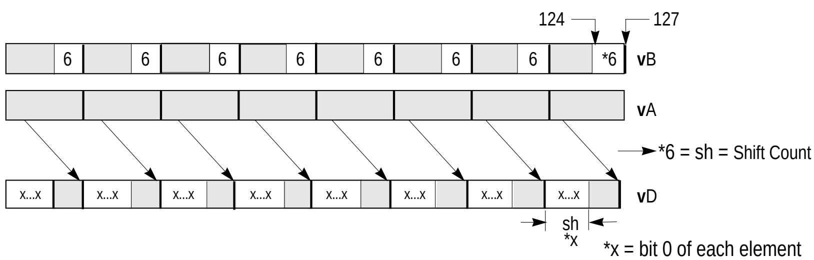


Figure 6-121. vsrah—Shift Bits Right for Eight Integer Elements (16-Bit)

vsraw

vsraw

Vector Shift Right Algebraic Word

vsraw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	900
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  sh ← (vB)i+27:i+31
  vDi:i+31 ← (vA)i:i+31 >>si sh
end
```

Each element is a word. Each element in **v**A is shifted right by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 31 of the element are lost. Bit 0 of the element is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-122 shows the usage of the **vsraw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

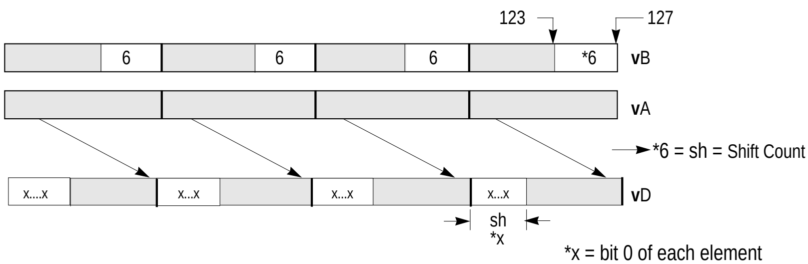


Figure 6-122. vsraw—Shift Bits Right in Four Integer Elements (32-Bit)

vsrb

vsrb

Vector Shift Right Byte

vsrb **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	516
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  sh ← (vB)i+5:i+7
  vDi:i+7 ← (vA)i:i+7 >>ui sh
end
```

Each element is a byte. Each element in **v**A is shifted right by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit 7 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-120 shows the usage of the **vsrb** command. Each of the sixteen elements in the vectors, **v**A, and **v**D, is 8 bits in length.

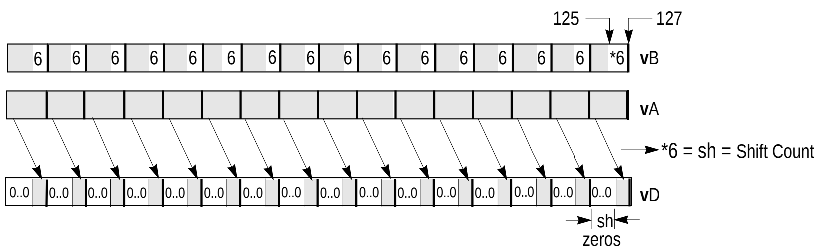


Figure 6-123. vsrb—Shift Bits Right in Sixteen Integer Elements (8-Bit)

vsrh vsrh

Vector Shift Right Half Word

vsrh **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	580
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  vDi:i+15 ← (vA)i:i+15 >>ui sh
end
```

Each element is a half word. Each element in **v**A is shifted right by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 15 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-124 shows the usage of the **vsrh** command. Each of the eight elements in the vectors, **v**A, and **v**D, is 16 bits in length.

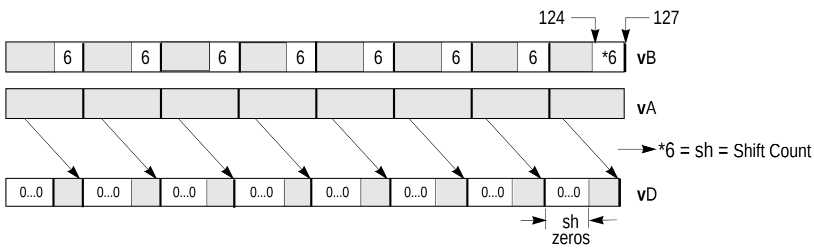


Figure 6-124. vsrh—Shift Bits Right for Eight Integer Elements (16-Bit)

vsro vsro

Vector Shift Right by Octet

vsro **v**D,**v**A,**v**B Form: VX

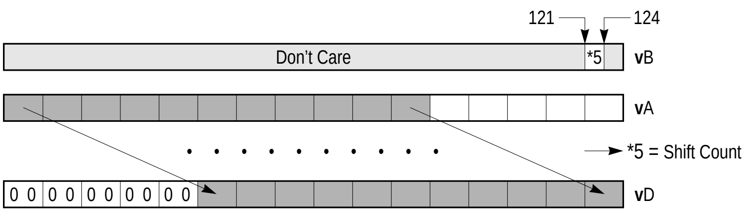
04	**v**D	**v**A	**v**B	1100
0 5	6 10	11 15	16 20	21 31

```
shb ← (vB)121:124
vD  ← (vA) >>ui (shb || 0b000)
```

The contents of **v**A are shifted right by the number of bytes specified in **v**B[121–124]. Bytes shifted out of **v**A are lost. Zeros are supplied to the vacated bytes on the left. The result is placed into **v**D.

Other registers altered:

- None



vsrw vsrw

Vector Shift Right Word

vsrw **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	644
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  sh ← (vB)i+(27):i+31
  vDi:i+31 ← (vA)i:i+31 >>ui sh
end
```

Each element is a word. Each element in **v**A is shifted right by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 31 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-122 shows the usage of the **vsrw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

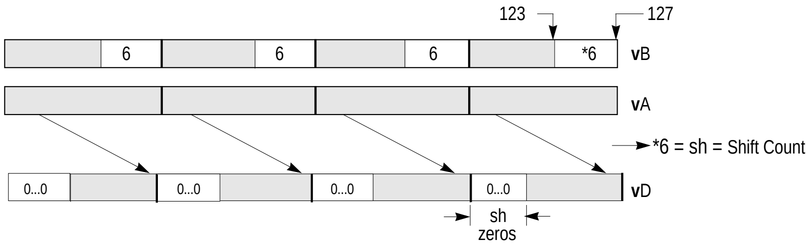


Figure 6-125. vsrw—Shift Bits Right in Four Integer Elements (32-Bit)

vsubcuw

vsubcuw

Vector Subtract Carryout Unsigned Word

vsubcuw **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	1408
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  aop0:32← ZeroExtend((vA)i:i+31,33)
  bop0:32← ZeroExtend((vB)i:i+31,33)
  temp0:32← aop0:32 +int ¬bop0:32 +int 1
  vDi:i+31← ZeroExtend(temp0,32)
end
```

Each unsigned-integer word element in **v**B is subtracted from the corresponding unsigned-integer word element in **v**A. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-126 shows the usage of the **vsubcuw** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

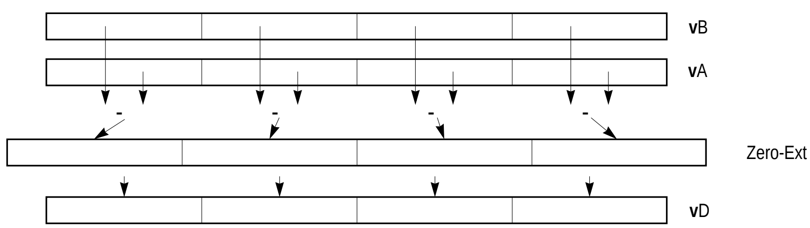


Figure 6-126. vsubcuw—Subtract Carryout of Four Unsigned Integer Elements (32-Bit)

vsubfp

vsubfp

Vector Subtract Floating Point

vsubfp **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	74
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
  vD[i:i+31] ← RndToNearFP32((vA)[i:i+31] -fp (vB)[i:i+31])
end
```

Each single-precision floating-point word element in **v**B is subtracted from the corresponding single-precision floating-point word element in **v**A. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

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.

Other registers altered:

- None

Figure 6-17 shows the usage of the **vsubfp** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

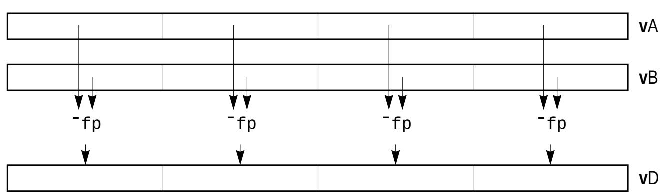


Figure 6-127. vsubfp—Subtract Four Floating Point Elements (32-Bit)

vsubsbs vsubsbs

Vector Subtract Signed Byte Saturate

vsubsbs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1792
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  aop0:8 ← SignExtend((vA)i:i+7,9)
  bop0:8 ← SignExtend((vB)i:i+7,9)
  temp0:8← aop0:8 +int ¬bop0:8 +int 1
  vDi:i+7← SItoSIsat(temp0:8,8)
end
```

Each element is a byte. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than (2^7-1) it saturates to (2^7-1) and if it is less than -2^7 it saturates to -2^7, where 8 is the length of the element.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-128 shows the usage of the **vsubsbs** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

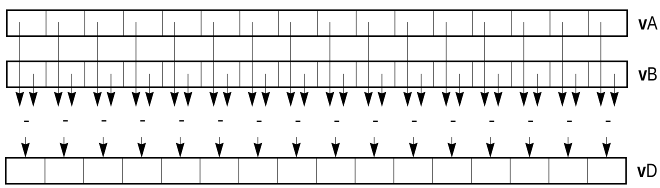


Figure 6-128. vsubsbs—Subtract Sixteen Signed Integer Elements (8-Bit)

vsubshs vsubshs

Vector Subtract Signed Half Word Saturate

vsubshs **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	1856
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  aop0:16← SignExtend((vA)i:i+15,17)
  bop0:16← SignExtend((vB)i:i+15,17)
  temp0:16← aop0:16 +int -bop0:16 +int 1
  vDi:i+15← SItoSIsat(temp0:16,16)
end
```

Each element is a half word. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than $(2^{15}-1)$ it saturates to $(2^{15}-1)$ and if it is less than -2^{15} it saturates to -2^{15}, where 16 is the length of the element.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-129 shows the usage of the **vsubshs** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits in length.

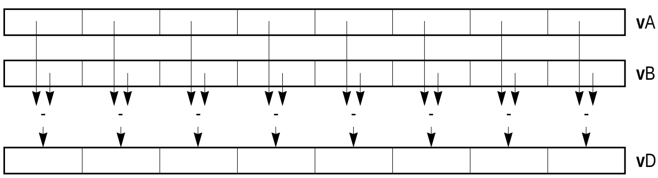


Figure 6-129. vsubshs—Subtract Eight Signed Integer Elements (16-Bit)

vsubsws vsubsws

Vector Subtract Signed Word Saturate

vsubsws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1920
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  aop0:32← SignExtend((vA)i:i+31,33)
  bop0:32← SignExtend((vB)i:i+31,33)
  temp0:32← aop0:32 +int ¬bop0:32 +int 1
  vDi:i+31← SItoSIsat(temp0:32,32)
end
```

Each element is a word. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than $(2^{31}-1)$ it saturates to $(2^{31}-1)$ and if it is less than -2^{31} it saturates to -2^{31}, where 32 is the length of the element.

The signed-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-130 shows the usage of the **vsubsws** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

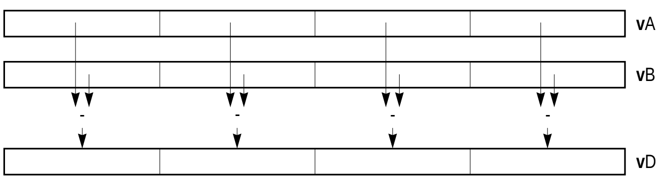


Figure 6-130. vsubsws—Subtract Four Signed Integer Elements (32-Bit)

vsububm vsububm

Vector Subtract Unsigned Byte Modulo

vsububm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1024
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
  vDi:i+7← (vA)i:i+7 +int –(vB)i:i+7
end
```

Each element of **vsububm** is a byte.

Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The integer result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Note the **vsububm** instruction can be used for unsigned or signed integers.

Figure 6-128 shows the usage of the **vsububm** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

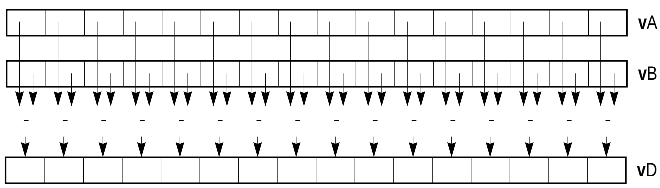


Figure 6-131. vsububm—Subtract Sixteen Integer Elements (8-Bit)

vsububs

vsububs

Vector Subtract Unsigned Byte Saturate

vsububs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1536
0 5	6 10	11 15	16 20	21 31

$$
\begin{array}{l}
\texttt{do i=0 to 127 by 8} \\
\quad \texttt{aop}_{\texttt{0:8}} \leftarrow \texttt{ZeroExtend((}\mathbf{v}\texttt{A)}_{\texttt{i:i+7}}\texttt{,9)} \\
\quad \texttt{bop}_{\texttt{0:8}} \leftarrow \texttt{ZeroExtend((}\mathbf{v}\texttt{B)}_{\texttt{i:i+7}}\texttt{,9)} \\
\quad \texttt{temp}_{\texttt{0:8}} \leftarrow \texttt{aop}_{\texttt{0:8}} +_{\texttt{int}} \neg\texttt{bop}_{\texttt{0:8}} +_{\texttt{int}} \texttt{1} \\
\quad \mathbf{v}\texttt{D}_{\texttt{i:i+7}} \leftarrow \texttt{SItoUIsat(temp}_{\texttt{0:8}}\texttt{,8)} \\
\texttt{end}
\end{array}
$$

Each element is a byte. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 8 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-128 shows the usage of the **vsububs** command. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits in length.

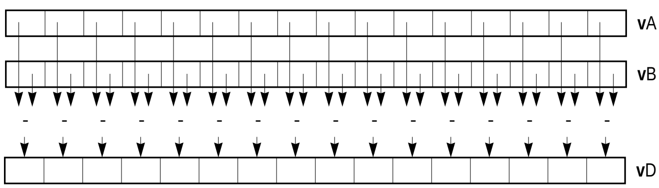


Figure 6-132. vsububs—Subtract Sixteen Unsigned Integer Elements (8-Bit)

vsubuhm

vsubuhm

Vector Subtract Signed Half Word Modulo

vsubuhm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1088
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  vD[i:i+15] ← (vA)[i:i+15] +int −(vB)[i:i+15]
end
```

Each element is a half word. Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The integer result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Note the **vsubuhm** instruction can be used for unsigned or signed integers.

Figure 6-133 shows the usage of the **vsubuhm** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits in length.

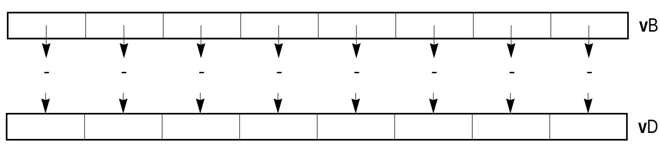


Figure 6-133. vsubuhm—Subtract Eight Integer Elements (16-Bit)

vsubuhs

vsubuhs

Vector Subtract Signed Half Word Saturate

vsubuhs **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	1600
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
  aop0:16← ZeroExtend((vA)i:i+15,17)
  bop0:16← ZeroExtend((vB)i:i+n:1,17)
  temp0:16← aop0:n +int ¬bop0:16 +int 1
  vDi:i+15← SItoUIsat(temp0:16,16)
end
```

Each element is a half word. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 16 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-134 shows the usage of the **vsubuhs** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits in length.

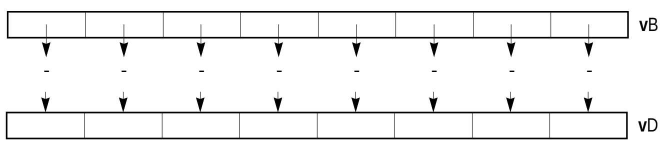


Figure 6-134. vsubuhs—Subtract Eight Signed Integer Elements (16-Bit)

vsubuwm vsubuwm

Vector Subtract Unsigned Word Modulo

vsubuwm **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1152
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  vDi:i+31← (vA)i:i+31 +int −(vB)i:i+31
end
```

Each element of **vsubuwm** is a word.

Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The integer result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Note the **vsubuwm** instruction can be used for unsigned or signed integers.

Figure 6-135 shows the usage of the **vsubuwm** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

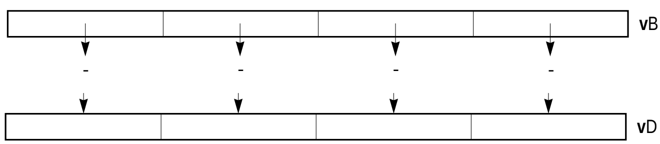


Figure 6-135. vsubuwm—Subtract Four Integer Elements (32-Bit)

vsubuws vsubuws

Vector Subtract Unsigned Word Saturate

vsubuws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1664
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  aop_{0:32}← ZeroExtend((vA)_{i:i+31},33)
  bop_{0:32}← ZeroExtend((vB)_{i:i+31},33)
  temp_{0:32}← aop_{0:32} +_{int} ¬bop_{0:32} +_{int} 1
  vD_{i:i+31}← SItoUIsat(temp_{0:32},32)
end
```

Each element is a word. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 32 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-135 shows the usage of the **vsubuws** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

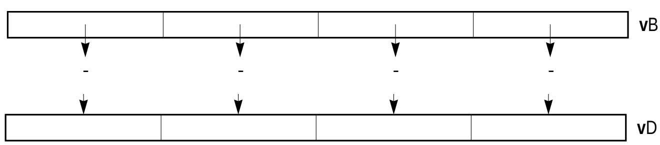


Figure 6-136. vsubuws—Subtract Four Signed Integer Elements (32-Bit)

vsumsws

vsumsws

Vector Sum Across Signed Word Saturate

vsumsws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1928
0 5	6 10	11 15	16 20	21 31

```
temp0:34 ← SignExtend((vB)96:127,35)
do i=0 to 127 by 32
  temp0:34 ← temp0:34 +int SignExtend((vA)i:i+31,35)
  vD ← 960 ∥ SItoSIsat(temp0:34,32)
end
```

The signed-integer sum of the four signed-integer word elements in **v**A is added to the signed-integer word element in bits of **v**B[96-127]. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into bits **v**D[96–127]. Bits **v**D[0–95] are cleared.

Other registers altered:

- SAT

Figure 6-137 shows the usage of the **vsumsws** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

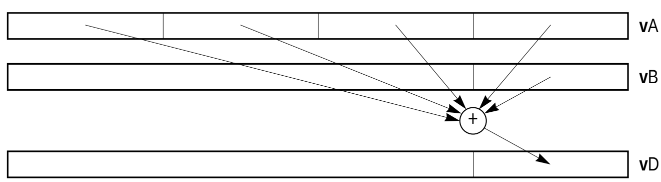


Figure 6-137. vsumsws—Sum Four Signed Integer Elements (32-Bit)

vsum2sws vsum2sws

Vector Sum Across Partial (1/2) Signed Word Saturate

vsum2sws **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1672
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 64
  temp0:33 ← SignExtend((vB)i+32:i+63,34)
  do j=0 to 63 by 32
    temp0:33 ← temp0:33 +int SignExtend((vA)i+j:i+j+31,34)
  end
  vDi:i+63 ← 320 || SItoSIsat(temp0:33,32)
end
```

The signed-integer sum of the first two signed-integer word elements in register **v**A is added to the signed-integer word element in **v**B[32–63]. If the intermediate result is greater than $(2^{31}-1)$ it saturates to $(2^{31}-1)$ and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into **v**D[32–63]. The signed-integer sum of the last two signed-integer word elements in register **v**A is added to the signed-integer word element in **v**B[96-127]. If the intermediate result is greater than $(2^{31}-1)$ it saturates to $(2^{31}-1)$ and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into **v**D[96–127]. The register **v**D[0–31,64–95] are cleared to 0.

Other registers altered:

- SAT

Figure 6-138 shows the usage of the **vsum2sws** command. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits in length.

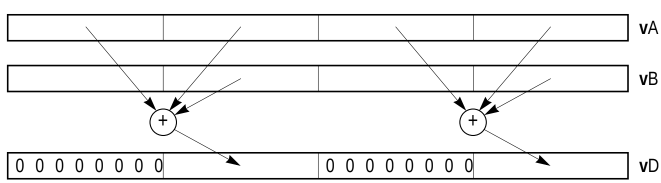


Figure 6-138. vsum2sws—Two Sums in the Four Signed Integer Elements (32-Bit)

vsum4sbs vsum4sbs

Vector Sum Across Partial (1/4) Signed Byte Saturate

vsum4sbs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1800
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  temp_{0:32} ← SignExtend((vB)_{i:i+31},33)
  do j=0 to 31 by 8
     temp_{0:32} ← temp_{0:32} +_{int} SignExtend((vA)_{i+j:i+j+7},33)
  end
  vD_{i:i+31} ← SItoSIsat(temp_{0:32},32)
end
```

For each word element in **v**B the following operations are performed in the order shown.

- The signed-integer sum of the four signed-integer byte elements contained in the corresponding word element of register **v**A is added to the signed-integer word element in register **v**B.
- If the intermediate result is greater than $(2^{31}-1)$ it saturates to $(2^{31}-1)$ and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-139 shows the usage of the **vsum4sbs** command. Each of the sixteen elements in the vector **v**A, is 8 bits in length. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

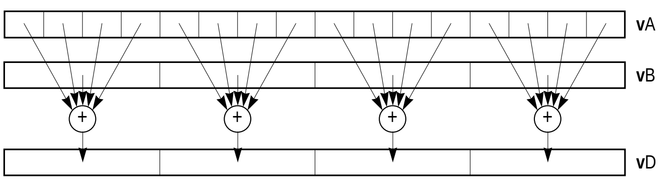


Figure 6-139. vsum4sbs—Four Sums in the Integer Elements (32-Bit)

vsum4shs

Vector Sum Across Partial (1/4) Signed Half Word Saturate

vsum4shs **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	1608
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  temp0:32 ← SignExtend((vB)i:i+31,33)
  do j=0 to 31 by 16
    temp0:32 ← temp0:32 +int SignExtend((vA)i+j:i+j+15,33)
  end
  vDi:i+31 ← SItoSIsat(temp0:32,32)
end
```

For each word element in register **v**B the following operations are performed, in the order shown.

- The signed-integer sum of the two signed-integer halfword elements contained in the corresponding word element of register **v**A is added to the signed-integer word element in **v**B.
- If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-140 shows the usage of the **vsum4shs** command. Each of the eight elements in the vector **v**A, is 16 bits in length. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

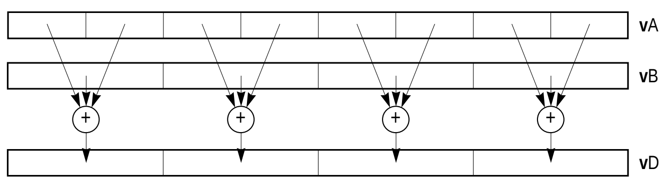


Figure 6-140. vsum4shs—Four Sums in the Integer Elements (32-Bit)

vsum4ubs vsum4ubs

Vector Sum Across Partial (1/4) Unsigned Byte Saturate

vsum4ubs **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1544
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  temp0:32 ← ZeroExtend((vB)i:i+31,33)
  do j=0 to 31 by 8
    temp0:32 ← temp0:32 +int ZeroExtend((vA)i+j:i+j+7,33)
  end
  vDi:i+31 ← UItoUIsat(temp0:32,32)
end
```

For each word element in **v**B the following operations are performed in the order shown.

- The unsigned-integer sum of the four unsigned-integer byte elements contained in the corresponding word element of register **v**A is added to the unsigned-integer word element in register **v**B.
- If the intermediate result is greater than $(2^{32}-1)$ it saturates to $(2^{32}-1)$.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-141 shows the usage of the **vsum4ubs** command. Each of the four elements in the vector **v**A, is 8 bits in length. Each of the four elements in the vectors **v**B and **v**D is 32 bits in length.

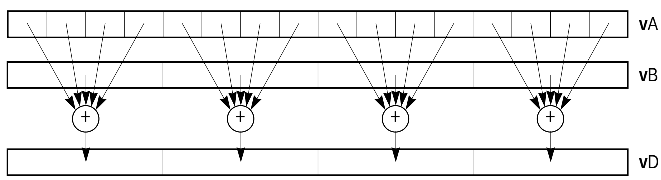


Figure 6-141. vsum4ubs—Four Sums in the Integer Elements (32-Bit)

vupkhpx

vupkhpx

Vector Unpack High Pixel16

vupkhpx **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	846
0 5	6 10	11 15	16 20	21 31

do i=0 to 63 by 16

$\quad \mathbf{v}D_{i*2:(i*2)+7} \leftarrow \mathrm{SignExtend}((\mathbf{v}B)_{i},8)$

$\quad \mathbf{v}D_{(i*2)+8:(i*2)+15} \leftarrow \mathrm{ZeroExtend}((\mathbf{v}B)_{i+1:i+5},8)$

$\quad \mathbf{v}D_{(i*2)+16:(i*2)+23} \leftarrow \mathrm{ZeroExtend}((\mathbf{v}B)_{i+6:i+10},8)$

$\quad \mathbf{v}D_{(i*2)+24:(i*2)+31} \leftarrow \mathrm{ZeroExtend}((\mathbf{v}B)_{i+11:i+15},8)$

end

Each halfword element in the high-order half of register **v**B is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of **v**D.

A halfword is unpacked to 32 bits by concatenating, in order, the results of the following operations.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1–5 of the halfword to 8 bits
- zero-extend bits 6–10 of the halfword to 8 bits
- zero-extend bits 11–15 of the halfword to 8 bits

Other registers altered:

- None

The source and target elements can be considered to be 16-bit and 32-bit "pixels" respectively, having the formats described in the programming note for the Vector Pack Pixel instruction.

Figure 6-142 shows the usage of the **vupkhpx** command. Each of the eight elements in the vectors, **v**B, is 16 bits in length. Each of the four elements in the vectors, **v**D, is 32 bits in length.

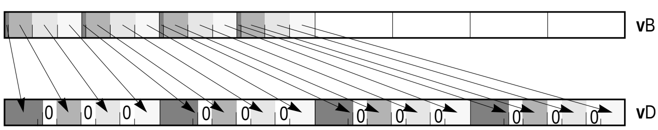


Figure 6-142. vupkhpx—Unpack High-Order Elements (16 bit) to Elements (32-Bit)

vupkhsb vupkhsb

Vector Unpack High Signed Byte

vupkhsb **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	526
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
  vDi*2:(i*2)+15 ← SignExtend((vB)i:i+7,16)
end
```

Each signed integer byte element in the high-order half of register **v**B is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight halfwords of register **v**D.

Other registers altered:

- None

Figure 6-143 shows the usage of the **vupkhsb** command. Each of the sixteen elements in the vectors, **v**B, is 8 bits in length. Each of the eight elements in the vectors, **v**D, is 16 bits in length.

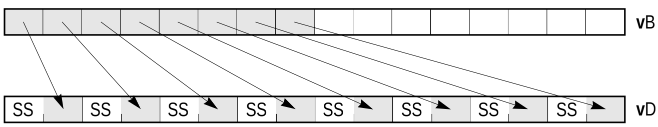


Figure 6-143. vupkhsb—Unpack HIgh-Order Signed Integer Elements (8-Bit) to Signed Integer Elements (16-Bit)

vupkhsh

vupkhsh

Vector Unpack High Signed Half Word

vupkhsh **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	590
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
  vDi*2:(i*2)+31 ← SignExtend((vB)i:i+15,32)
end
```

Each signed integer halfword element in the high-order half of register **v**B is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-143 shows the usage of the **vupkhsh** command. Each of the eight elements in the vectors **v**B and **v**D is 16 bits in length.

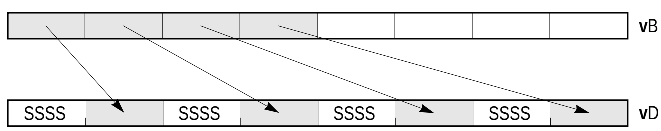


Figure 6-144. vupkhsh—Unpack Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

vupklpx

vupklpx

Vector Unpack Low Pixel16

vupklpx **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	974
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 63 by 16
  vD_{i*2:(i*2)+7} ← SignExtend((vB)_{i+64},8)
  vD_{(i*2)+8:(i*2)+15} ← ZeroExtend((vB)_{i+65:i+69},8)
  vD_{(i*2)+16:(i*2)+23} ← ZeroExtend((vB)_{i+70:i+74},8)
  vD_{(i*2)+24:(i*2)+31} ← ZeroExtend((vB)_{i+75:i+79},8)
end
```

Each halfword element in the low-order half of register **v**B is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of register **v**D.

A halfword is unpacked to 32 bits by concatenating, in order, the results of the following operations.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1–5 of the halfword to 8 bits
- zero-extend bits 6–10 of the halfword to 8 bits
- zero-extend bits 11–15 of the halfword to 8 bits

Other registers altered:

- None

Programming note: Notice that the unpacking done by the Vector Unpack Pixel instructions does not reverse the packing done by the Vector Pack Pixel instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, Vector Unpack Pixel inserts high-order bits while Vector Pack Pixel discards low-order bits).

Figure 6-142 shows the usage of the **vupklpx** command. Each of the eight elements in the vectors, **v**B, is 16 bits in length. Each of the four elements in the vectors, **v**D, is 32 bits in length.

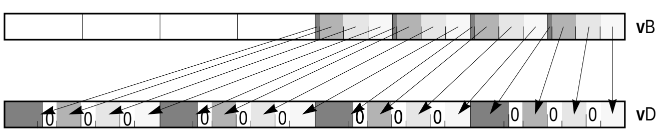


Figure 6-145. vupklpx—Unpack Low-order Elements (16-Bit) to Elements (32-Bit)

vupklsb vupklsb

Vector Unpack Low Signed Byte

vupklsb **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	654
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
  vD_{i*2:(i*2)+15} ← SignExtend((vB)_{i+64:i+71},16)
end
```

Each signed integer byte element in the low-order half of register **v**B is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight halfwords of register **v**D.

Other registers altered:

- None

Figure 6-14 shows the usage of the **vaddubs** command. Each of the sixteen elements in the vectors **v**B and **v**D is 8 bits in length.

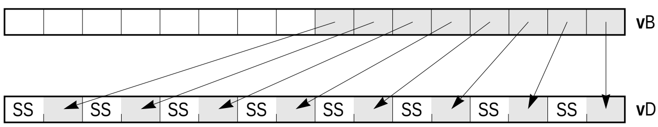


Figure 6-146. vupklsb—Unpack Low-Order Elements (8-Bit) to Elements (16-Bit)

vupklsh vupklsh

Vector Unpack Low Signed Half Word

vupklsh **v**D,**v**B Form: VX

04	**v**D	0 0 0 0 0	**v**B	718
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
  vDi*2:(i*2)+31 ← SignExtend((vB)i+64:i+79,32)
end
```

Each signed integer half word element in the low-order half of register **v**B is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-147 shows the usage of the **vupklpx** command. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits in length.

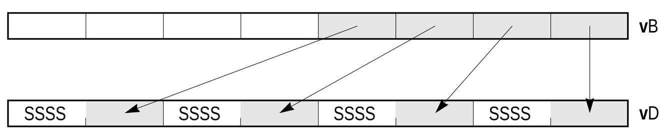


Figure 6-147. vupklsh—Unpack Low-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

vxor vxor

Vector Logical XOR

vxor **v**D,**v**A,**v**B Form: VX

04	**v**D	**v**A	**v**B	1220
0 5	6 10	11 15	16 20	21 31

vD ← (**v**A) ⊕ (**v**B)

The contents of **v**A are XORed with the contents of register **v**B and the result is placed into register **v**D.

Other registers altered:

- None
- Figure 6-148 shows the usage of the **vxor** command.

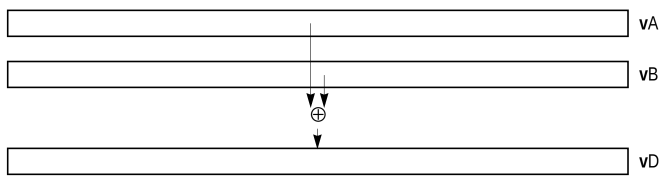


Figure 6-148. vxor—Bitwise XOR (128-Bit)

Appendix A
AltiVec Instruction Set Listings

This appendix lists the instruction set for the AltiVec™ technology. Instructions are sorted by mnemonic, opcode, and form. Also included in this appendix is a quick reference table that contains general information, such as the architecture level, privilege level, and form, and indicates if the instruction is 64-bit and/or optional.

Note that split fields, which represent the concatenation of sequences from left to right, are shown in lowercase.

A.1 Instructions Sorted by Mnemonic

Table A-1 lists the instructions implemented in the AltiVec architecture in alphabetical order by mnemonic.

Key:

Reserved bits

Table A-1. Complete Instruction List Sorted by Mnemonic

Name	0–5	6	7–8	9–10	11–15	16–20	21–25	26–30	31
dss	31	A	0 0	STRM	0 0 0 0 0	0 0 0 0 0	822		0
dssall	31	A	0 0	STRM	0 0 0 0 0	0 0 0 0 0	822		0
dst	31	T	0 0	STRM	A	B	342		0
dstst	31	T	0 0	STRM	A	B	374		0
dststt	31	1	0 0	tag	A	B	11	22	0
dstt	31	1	0 0	tag	A	B	0		0
lvebx	31	**v**D			A	B	7		0
lvehx	31	**v**D			A	B	39		0
lvewx	31	**v**D			A	B	71		0
lvsl	31	**v**D			A	B	6		0
lvsr	31	**v**D			A	B	38		0
lvx	31	**v**D			A	B	103		0

<table>
<tr><th>Name</th><th>0 5</th><th>6 7 8 9 10</th><th>11 12 13 14 15</th><th>16 17 18 19 20</th><th>21</th><th>22 23 24 25 26 27 28 29 30</th><th>31</th></tr>
<tr><td>**lvxl**</td><td>31</td><td>**v**D</td><td>A</td><td>B</td><td colspan="2">359</td><td>0</td></tr>
<tr><td>**mfvscr**</td><td>04</td><td>**v**D</td><td>0 0 0 0 0</td><td>0 0 0 0 0</td><td colspan="2">1540</td><td>0</td></tr>
<tr><td>**mtvscr**</td><td>04</td><td>///</td><td>0 0 0 0 0</td><td>**v**D</td><td colspan="2">1604</td><td>0</td></tr>
<tr><td>**stvebx**</td><td>31</td><td>S</td><td>A</td><td>B</td><td colspan="2">135</td><td>0</td></tr>
<tr><td>**stvehx**</td><td>31</td><td>S</td><td>A</td><td>B</td><td colspan="2">167</td><td>0</td></tr>
<tr><td>**stvewx**</td><td>31</td><td>S</td><td>A</td><td>B</td><td colspan="2">199</td><td>0</td></tr>
<tr><td>**stvx**</td><td>31</td><td>S</td><td>A</td><td>B</td><td colspan="2">231</td><td>0</td></tr>
<tr><td>**stvxl**</td><td>31</td><td>S</td><td>A</td><td>B</td><td colspan="2">487</td><td>0</td></tr>
<tr><td>**vaddcuw**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">384</td><td>0</td></tr>
<tr><td>**vaddfp**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">10</td><td>0</td></tr>
<tr><td>**vaddsbs**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">768</td><td>0</td></tr>
<tr><td>**vaddshs**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">832</td><td>0</td></tr>
<tr><td>**vaddsws**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">896</td><td>0</td></tr>
<tr><td>**vaddubm**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">0</td><td>0</td></tr>
<tr><td>**vaddubs**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">512</td><td>0</td></tr>
<tr><td>**vadduhm**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">64</td><td>0</td></tr>
<tr><td>**vadduhs**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">576</td><td>0</td></tr>
<tr><td>**vadduwm**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">128</td><td>0</td></tr>
<tr><td>**vadduws**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">640</td><td>0</td></tr>
<tr><td>**vand**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1028</td><td>0</td></tr>
<tr><td>**vandc**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1092</td><td>0</td></tr>
<tr><td>**vavgsb**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1282</td><td>0</td></tr>
<tr><td>**vavgsh**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1346</td><td>0</td></tr>
<tr><td>**vavgsw**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1410</td><td>0</td></tr>
<tr><td>**vavgub**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1026</td><td>0</td></tr>
<tr><td>**vavguh**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1090</td><td>0</td></tr>
<tr><td>**vavguw**</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td colspan="2">1154</td><td>0</td></tr>
<tr><td>**vcfsx**</td><td>04</td><td>**v**D</td><td>UIMM</td><td>**v**B</td><td colspan="3">842</td></tr>
<tr><td>**vcfux**</td><td>04</td><td>**v**D</td><td>UIMM</td><td>**v**B</td><td colspan="2">778</td><td>0</td></tr>
<tr><td>**vcmpbfp*x***</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td>Rc</td><td colspan="2">966</td></tr>
<tr><td>**vcmpeqfp*x***</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td>Rc</td><td colspan="2">198</td></tr>
<tr><td>**vcmpequb*x***</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td>Rc</td><td colspan="2">6</td></tr>
<tr><td>**vcmpequh*x***</td><td>04</td><td>**v**D</td><td>**v**A</td><td>**v**B</td><td>Rc</td><td colspan="2">70</td></tr>
</table>

Name	0–5	6–10	11–15	16–20	21 / 21–25	22–31 / 26–31
vcmpequw*x*	04	**v**D	**v**A	**v**B	Rc	134
vcmpgefp*x*	04	**v**D	**v**A	**v**B	Rc	454
vcmpgtfp*x*	04	**v**D	**v**A	**v**B	Rc	710
vcmpgtsb*x*	04	**v**D	**v**A	**v**B	Rc	774
vcmpgtsh*x*	04	**v**D	**v**A	**v**B	Rc	838
vcmpgtsw*x*	04	**v**D	**v**A	**v**B	Rc	902
vcmpgtub*x*	04	**v**D	**v**A	**v**B	Rc	518
vcmpgtuh*x*	04	**v**D	**v**A	**v**B	Rc	582
vcmpgtuw*x*	04	**v**D	**v**A	**v**B	Rc	646
vctsxs	04	**v**D	UIMM	**v**B		970
vctuxs	04	**v**D	UIMM	**v**B		906
vexptefp	04	**v**D	0 0 0 0 0	**v**B		394
vlogefp	04	**v**D	0 0 0 0 0	**v**B		458
vmaddfp	04	**v**D	**v**A	**v**B	**v**C	46
vmaxfp	04	**v**D	**v**A	**v**B		1034
vmaxsb	04	**v**D	**v**A	**v**B		258
vmaxsh	04	**v**D	**v**A	**v**B		322
vmaxsw	04	**v**D	**v**A	**v**B		386
vmaxub	04	**v**D	**v**A	**v**B		2
vmaxuh	04	**v**D	**v**A	**v**B		66
vmaxuw	04	**v**D	**v**A	**v**B		130
vmhaddshs	04	**v**D	**v**A	**v**B	**v**C	32
vmhraddshs	04	**v**D	**v**A	**v**B	**v**C	33
vminfp	04	**v**D	**v**A	**v**B		1098
vminsb	04	**v**D	**v**A	**v**B		770
vminsh	04	**v**D	**v**A	**v**B		834
vminsw	04	**v**D	**v**A	**v**B		898
vminub	04	**v**D	**v**A	**v**B		514
vminuh	04	**v**D	**v**A	**v**B		578
vminuw	04	**v**D	**v**A	**v**B		642
vmladduhm	04	**v**D	**v**A	**v**B	**v**C	34
vmrghb	04	**v**D	**v**A	**v**B		12
vmrghh	04	**v**D	**v**A	**v**B		76

Name	0–5	6–10	11–15	16–20	21–25	26–31
vmrghw	04	**v**D	**v**A	**v**B	140	
vmrglb	04	**v**D	**v**A	**v**B	268	
vmrglh	04	**v**D	**v**A	**v**B	332	
vmrglw	04	**v**D	**v**A	**v**B	396	
vmsummbm	04	**v**D	**v**A	**v**B	**v**C	37
vmsumshm	04	**v**D	**v**A	**v**B	**v**C	40
vmsumshs	04	**v**D	**v**A	**v**B	**v**C	41
vmsumubm	04	**v**D	**v**A	**v**B	**v**C	36
vmsumuhm	04	**v**D	**v**A	**v**B	**v**C	38
vmsumuhs	04	**v**D	**v**A	**v**B	**v**C	39
vmulesb	04	**v**D	**v**A	**v**B	776	
vmulesh	04	**v**D	**v**A	**v**B	840	
vmuleub	04	**v**D	**v**A	**v**B	520	
vmuleuh	04	**v**D	**v**A	**v**B	584	
vmulosb	04	**v**D	**v**A	**v**B	264	
vmulosh	04	**v**D	**v**A	**v**B	328	
vmuloub	04	**v**D	**v**A	**v**B	8	
vmulouh	04	**v**D	**v**A	**v**B	72	
vnmsubfp	04	**v**D	**v**A	**v**B	**v**C	47
vnor	04	**v**D	**v**A	**v**B	1284	
vor	04	**v**D	**v**A	**v**B	1156	
vperm	04	**v**D	**v**A	**v**B	**v**C	43
vpkpx	04	**v**D	**v**A	**v**B	782	
vpkshss	04	**v**D	**v**A	**v**B	398	
vpkshus	04	**v**D	**v**A	**v**B	270	
vpkswss	04	**v**D	**v**A	**v**B	462	
vpkuhum	04	**v**D	**v**A	**v**B	14	
vpkuhus	04	**v**D	**v**A	**v**B	142	
vpkuwum	04	**v**D	**v**A	**v**B	78	
vpkuwus	04	**v**D	**v**A	**v**B	206	
vrefp	04	**v**D	0 0 0 0 0	**v**B	266	
vrfim	04	**v**D	0 0 0 0 0	**v**B	714	
vrfin	04	**v**D	0 0 0 0 0	**v**B	522	

Name	0–5	6–10	11–15	16–20	21–25	26–31
vrfip	04	**v**D	0 0 0 0 0	**v**B	650	
vrfiz	04	**v**D	0 0 0 0 0	**v**B	586	
vrlb	04	**v**D	**v**A	**v**B	4	
vrlh	04	**v**D	**v**A	**v**B	68	
vrlw	04	**v**D	**v**A	**v**B	132	
vrsqrtefp	04	**v**D	0 0 0 0 0	**v**B	330	
vsel	04	**v**D	**v**A	**v**B	**v**C	42
vsl	04	**v**D	**v**A	**v**B	452	
vslb	04	**v**D	**v**A	**v**B	260	
vsldoi	04	**v**D	**v**A	**v**B	0 SH	44
vslh	04	**v**D	**v**A	**v**B	324	
vslo	04	**v**D	**v**A	**v**B	1036	
vslw	04	**v**D	**v**A	**v**B	388	
vspltb	04	**v**D	UIMM	**v**B	524	
vsplth	04	**v**D	UIMM	**v**B	588	
vspltisb	04	**v**D	SIMM	**v**B	780	
vspltish	04	**v**D	SIMM	0 0 0 0 0	844	
vspltisw	04	**v**D	SIMM	0 0 0 0 0	908	
vspltw	04	**v**D	UIMM	**v**B	652	
vsr	04	**v**D	**v**A	**v**B	708	
vsrab	04	**v**D	**v**A	**v**B	772	
vsrah	04	**v**D	**v**A	**v**B	836	
vsraw	04	**v**D	**v**A	**v**B	900	
vsrb	04	**v**D	**v**A	**v**B	516	
vsrh	04	**v**D	**v**A	**v**B	580	
vsro	04	**v**D	**v**A	**v**B	1100	
vsrw	04	**v**D	**v**A	**v**B	644	
vsubcuw	04	**v**D	**v**A	**v**B	1408	
vsubfp	04	**v**D	**v**A	**v**B	74	
vsubsbs	04	**v**D	**v**A	**v**B	1792	
vsubshs	04	**v**D	**v**A	**v**B	1856	
vsubsws	04	**v**D	**v**A	**v**B	1920	
vsububm	04	**v**D	**v**A	**v**B	1024	

Name	0–5	6–10	11–15	16–20	21–31
vsububs	04	**v**D	**v**A	**v**B	1536
vsubuhm	04	**v**D	**v**A	**v**B	1088
vsubuhs	04	**v**D	**v**A	**v**B	1600
vsubuwm	04	**v**D	**v**A	**v**B	1152
vsubuws	04	**v**D	**v**A	**v**B	1664
vsumsws	04	**v**D	**v**A	**v**B	1928
vsum2sws	04	D	A	B	1672
vsum4sbs	04	D	A	B	1800
vsum4shs	04	D	A	B	1608
vsum4ubs	04	D	A	B	1544
vupkhpx	04	D	0 0 0 0 0	B	846
vupkhsb	04	D	0 0 0 0 0	B	526
vupkhsh	04	D	0 0 0 0 0	B	590
vupklpx	04	D	0 0 0 0 0	B	974
vupklsb	04	D	0 0 0 0 0	B	654
vupklsh	04	D	0 0 0 0 0	B	718
vxor	04	D	A	B	1220

A.2 Instructions Sorted by Opcode

Table A-2 lists the Altivec instructions grouped by opcode.

Key:

Reserved bits

Table A-2. Instructions Sorted by Opcode

Name	0–5	6–10	11–15	16–20	21–25	26–31
vmhaddshs	000100	vD	vA	vB	vC	10 0000
vmhraddshs	000100	vD	vA	vB	vC	10 0001
vmladduhm	000100	vD	vA	vB	vC	10 0010
vmsumubm	000100	vD	vA	vB	vC	10 0100
vmsummbm	000100	vD	vA	vB	vC	10 0101
vmsumuhm	000100	vD	A	vB	vC	10 0110
vmsumuhs	000100	vD	vA	vB	vC	10 0111
vmsumshm	000100	vD	vA	vB	vC	10 1000
vmsumshs	000100	vD	vA	vB	vC	10 1001
vsel	000100	vD	vA	vB	vC	10 1010
vperm	000100	vD	vA	vB	vC	10 1011
vsldoi	000100	vD	vA	vB	0 SH	10 1100
vmaddfp	000100	vD	vA	vB	000 0010 1110	
vnmsubfp	000100	vD	vA	vB	vC	10 1111
vaddubm	000100	vD	vA	vB	000 0000 0000	
vadduhm	000100	vD	vA	vB	000 0100 0000	
vadduwm	000100	vD	vA	vB	000 1000 0000	
vaddcuw	000100	vD	vA	vB	001 1000 0000	
vaddubs	000100	vD	vA	vB	010 0000 0000	
vadduhs	000100	vD	vA	vB	010 0100 0000	
vadduws	000100	vD	vA	vB	010 1000 0000	
vaddsbs	000100	vD	vA	vB	011 0000 0000	
vaddshs	000100	vD	vA	vB	011 0100 0000	
vaddsws	000100	vD	vA	vB	011 1000 0000	
vsububm	000100	vD	vA	vB	100 0000 0000	
vsubuhm	000100	vD	vA	vB	100 0100 0000	
vsubuwm	000100	vD	vA	vB	100 1000 0000	
vsubcuw	000100	vD	vA	vB	101 1000 0000	
vsububs	000100	vD	vA	vB	110 0000 0000	
vsubuhs	000100	vD	vA	vB	110 0100 0000	

Name	0–5	6–10	11–15	16–20	21–31
vsubuws	000100	**v**D	**v**A	**v**B	110 1000 0000
vsubsbs	000100	**v**D	**v**A	**v**B	111 0000 0000
vsubshs	000100	**v**D	**v**A	**v**B	111 0100 0000
vsubsws	000100	**v**D	**v**A	**v**B	111 1000 0000
vmaxub	000100	**v**D	**v**A	**v**B	000 0000 0010
vmaxuh	000100	**v**D	**v**A	**v**B	000 0100 0010
vmaxuw	000100	**v**D	**v**A	**v**B	000 1000 0010
vmaxsb	000100	**v**D	**v**A	**v**B	001 0000 0010
vmaxsh	000100	**v**D	**v**A	**v**B	001 0100 0010
vmaxsw	000100	**v**D	**v**A	**v**B	001 1000 0010
vminub	000100	**v**D	**v**A	**v**B	010 0000 0010
vminuh	000100	**v**D	**v**A	**v**B	010 0100 0010
vminuw	000100	**v**D	**v**A	**v**B	010 1000 0010
vminsb	000100	**v**D	**v**A	**v**B	011 0000 0010
vminsh	000100	**v**D	**v**A	**v**B	011 0100 0010
vminsw	000100	**v**D	**v**A	**v**B	011 1000 0010
vavgub	000100	**v**D	**v**A	**v**B	100 0000 0010
vavguh	000100	**v**D	**v**A	**v**B	100 0100 0010
vavguw	000100	**v**D	**v**A	**v**B	100 1000 0010
vavgsb	000100	**v**D	**v**A	**v**B	101 0000 0010
vavgsh	000100	**v**D	**v**A	**v**B	101 0100 0010
vavgsw	000100	**v**D	**v**A	**v**B	101 1000 0010
vrlb	000100	**v**D	**v**A	**v**B	000 0000 0100
vrlh	000100	**v**D	**v**A	**v**B	000 0100 0100
vrlw	000100	**v**D	**v**A	**v**B	000 1000 0100
vslb	000100	**v**D	**v**A	**v**B	001 0000 0100
vslh	000100	**v**D	**v**A	**v**B	001 0100 0100
vslw	000100	**v**D	**v**A	**v**B	001 1000 0100
vsl	000100	**v**D	**v**A	**v**B	001 1100 0100
vsrb	000100	**v**D	**v**A	**v**B	010 0000 0100
vsrh	000100	**v**D	**v**A	**v**B	010 0100 0100
vsrw	000100	**v**D	**v**A	**v**B	010 1000 0100
vsr	000100	**v**D	**v**A	**v**B	010 1100 0100
vsrab	000100	**v**D	**v**A	**v**B	011 0000 0100
vsrah	000100	**v**D	**v**A	**v**B	011 0100 0100
vsraw	000100	**v**D	**v**A	**v**B	011 1000 0100
vand	000100	**v**D	**v**A	**v**B	100 0000 0100

Name	0–5	6–10	11–15	16–20	21–31	
vandc	000100	**v**D	**v**A	**v**B	100 0100 0100	
vor	000100	**v**D	**v**A	**v**B	100 1000 0100	
vxor	000100	**v**D	**v**A	**v**B	100 1100 0100	
vnor	000100	**v**D	**v**A	**v**B	101 0000 0100	
mfvscr	000100	**v**D	0 0 0 0 0	0 0 0 0 0	11 0000 0010	0
mtvscr	000100	0 0 0 0 0	0 0 0 0 0	**v**B	11 0010 0010	0
vcmpequb*x*	000100	**v**D	**v**A	**v**B	Rc	00 0000 0110
vcmpequh*x*	000100	**v**D	**v**A	**v**B	Rc	00 0100 0110
vcmpequw*x*	000100	**v**D	**v**A	**v**B	Rc	00 1000 0110
vcmpeqfp*x*	000100	**v**D	**v**A	**v**B	Rc	00 1100 0110
vcmpgefp*x*	000100	**v**D	**v**A	**v**B	Rc	01 1100 0110
vcmpgtub*x*	000100	**v**D	**v**A	**v**B	Rc	10 0000 0110
vcmpgtuh*x*	000100	**v**D	**v**A	**v**B	Rc	10 0100 0110
vcmpgtuw*x*	000100	**v**D	**v**A	**v**B	Rc	10 1000 0110
vcmpgtfp*x*	000100	**v**D	**v**A	**v**B	Rc	10 1100 0110
vcmpgtsb*x*	000100	**v**D	**v**A	**v**B	Rc	11 0000 0110
vcmpgtsh*x*	000100	**v**D	**v**A	**v**B	Rc	11 0100 0110
vcmpgtsw*x*	000100	**v**D	**v**A	**v**B	Rc	11 1000 0110
vcmpbfp*x*	000100	**v**D	**v**A	**v**B	Rc	11 1100 0110
vmuloub	000100	**v**D	**v**A	**v**B	000 0000 1000	
vmulouh	000100	**v**D	**v**A	**v**B	000 0100 1000	
vmulosb	000100	**v**D	**v**A	**v**B	001 0000 1000	
vmulosh	000100	**v**D	**v**A	**v**B	001 0100 1000	
vmuleub	000100	**v**D	**v**A	**v**B	010 0000 1000	
vmuleuh	000100	**v**D	**v**A	**v**B	010 0100 1000	
vmulesb	000100	**v**D	**v**A	**v**B	011 0000 1000	
vmulesh	000100	**v**D	**v**A	**v**B	011 0100 1000	
vsum4ubs	000100	**v**D	**v**A	**v**B	110 0000 1000	
vsum4sbs	000100	**v**D	**v**A	**v**B	111 0000 1000	
vsum4shs	000100	**v**D	**v**A	**v**B	110 0100 1000	
vsum2sws	000100	**v**D	**v**A	**v**B	110 1000 1000	
vsumsws	000100	**v**D	**v**A	**v**B	111 1000 1000	
vaddfp	000100	**v**D	**v**A	**v**B	000 0000 1010	
vsubfp	000100	**v**D	**v**A	**v**B	000 0100 1010	
vrefp	000100	**v**D	0 0 0 0 0	**v**B	001 0000 1010	
vrsqrtefp	000100	**v**D	0 0 0 0 0	**v**B	001 0100 1010	
vexptefp	000100	**v**D	0 0 0 0 0	**v**B	001 1000 1010	

Name	0–5	6–10	11–15	16–20	21–31
vlogefp	000100	**v**D	0 0 0 0 0	**v**B	001 1100 1010
vrfin	000100	**v**D	0 0 0 0 0	**v**B	010 0000 1010
vrfiz	000100	**v**D	0 0 0 0 0	**v**B	010 0100 1010
vrfip	000100	**v**D	0 0 0 0 0	**v**B	010 1000 1010
vrfim	000100	**v**D	0 0 0 0 0	**v**B	010 1100 1010
vcfux	000100	**v**D	UIMM	**v**B	011 0000 1010
vcfsx	000100	**v**D	UIMM	**v**B	011 0100 1010
vctuxs	000100	**v**D	UIMM	**v**B	011 1000 1010
vctsxs	000100	**v**D	UIMM	**v**B	011 1100 1010
vmaxfp	000100	**v**D	**v**A	**v**B	100 0000 1010
vminfp	000100	**v**D	**v**A	**v**B	100 0100 1010
vmrghb	000100	**v**D	**v**A	**v**B	000 0000 1100
vmrghh	000100	**v**D	**v**A	**v**B	000 0100 1100
vmrghw	000100	**v**D	**v**A	**v**B	000 1000 1100
vmrglb	000100	**v**D	**v**A	**v**B	001 0000 1100
vmrglh	000100	**v**D	**v**A	**v**B	001 0100 1100
vmrglw	000100	**v**D	**v**A	**v**B	001 1000 1100
vspltb	000100	**v**D	UIMM	**v**B	010 0000 1100
vsplth	000100	**v**D	UIMM	**v**B	010 0100 1100
vspltw	000100	**v**D	UIMM	**v**B	010 1000 1100
vspltisb	000100	**v**D	SIMM	0 0 0 0 0	011 0000 1100
vspltish	000100	**v**D	SIMM	0 0 0 0 0	011 0100 1100
vspltisw	000100	**v**D	SIMM	0 0 0 0 0	011 1000 1100
vslo	000100	**v**D	**v**A	**v**B	100 0000 1100
vsro	000100	**v**D	**v**A	**v**B	100 0100 1100
vpkuhum	000100	**v**D	**v**A	**v**B	000 0000 1110
vpkuwum	000100	**v**D	**v**A	**v**B	000 0100 1110
vpkuhus	000100	**v**D	**v**A	**v**B	000 1000 1110
vpkuwus	000100	**v**D	**v**A	**v**B	000 1100 1110
vpkshus	000100	**v**D	**v**A	**v**B	001 0000 1110
vpkswus	000100	**v**D	**v**A	**v**B	001 0100 1110
vpkshss	000100	**v**D	**v**A	**v**B	001 1000 1110
vpkswss	000100	**v**D	**v**A	**v**B	001 1100 1110
vupkhsb	000100	**v**D	0 0 0 0 0	**v**B	010 0000 1110
vupkhsh	000100	**v**D	0 0 0 0 0	**v**B	010 0100 1110
vupklsb	000100	**v**D	0 0 0 0 0	**v**B	010 1000 1110
vupklsh	000100	**v**D	0 0 0 0 0	**v**B	010 1100 1110

Name	0 – 5	6	7 – 8	9 – 10	11 – 15	16 – 20	21 – 30	31
vpkpx	000100	**v**D			**v**A	**v**B	011 0000 1110	
vupkhpx	000100	**v**D			0 0 0 0 0	**v**B	011 0100 1110	
vupklpx	000100	**v**D			0 0 0 0 0	**v**B	011 1100 1110	
lvsl	011111	**v**D			A	B	00 0000 0110	0
lvsr	011111	**v**D			A	B	00 0010 0110	0
dst	011111	T	0 0	STRM	A	B	01 0101 0110	0
dstt	011111	1	0 0 0	tag	A	B	00 0000 0000	0
dstst	011111	T	0 0	STRM	A	B	01 0111 0110	0
dststt	011111	1	0 0 0	tag	A	B	1011 1 0110	0
dss	011111	A	0 0	STRM	0 0 0 0 0	0 0 0 0 0	11 0011 0110	0
dssall	011111	A	0 0	STRM	0 0 0 0 0	0 0 0 0 0	11 0011 0110	0
lvebx	011111	**v**D			A	B	00 0000 0111	0
lvehx	011111	**v**D			A	B	00 0010 0111	0
lvewx	011111	**v**D			A	B	00 0100 0111	0
lvx	011111	**v**D			A	B	00 0110 0111	0
lvxl	011111	**v**D			A	B	01 0110 0111	0
stvebx	011111	**v**S			A	B	00 1000 0111	0
stvehx	011111	**v**S			A	B	00 1010 0111	0
stvewx	011111	**v**S			A	B	00 1100 0111	0
stvx	011111	**v**S			A	B	00 1110 0111	0
stvxl	011111	**v**S			A	B	01 1110 0111	0

A.3 Instructions Sorted by Form

Table A-3 through Table A-6 list the AltiVec instructions grouped by form.

Key:

Reserved bits

Table A-3. VA-Form

OPCD	**v**D	**v**A	**v**B	**v**C		XO
OPCD	**v**D	**v**A	**v**B	0	SH	XO

Specific Instructions

Name	0 ... 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25		26 27 28 29 30 31
vmhaddshs	04	**v**D	**v**A	**v**B	**v**C		32
vmhraddshs	04	**v**D	**v**A	**v**B	**v**C		33
vmladduhm	04	**v**D	**v**A	**v**B	**v**C		34
vmsumubm	04	**v**D	**v**A	**v**B	**v**C		36
vmsummbm	04	**v**D	**v**A	**v**B	**v**C		37
vmsumuhm	04	**v**D	A	**v**B	**v**C		38
vmsumuhs	04	**v**D	**v**A	**v**B	**v**C		39
vmsumshm	04	**v**D	**v**A	**v**B	**v**C		40
vmsumshs	04	**v**D	**v**A	**v**B	**v**C		41
vsel	04	**v**D	**v**A	**v**B	**v**C		42
vperm	04	**v**D	**v**A	**v**B	**v**C		43
vsldoi	04	**v**D	**v**A	**v**B	0	SH	44
vmaddfp	04	**v**D	**v**A	**v**B	**v**C		46
vnmsubfp	04	**v**D	**v**A	**v**B	**v**C		47

Table A-4. VX-Form

OPCD	**v**D	**v**A	**v**B	XO	
OPCD	**v**D	0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	0 0 0 0 0	0 0 0 0 0	**v**B	XO	0
OPCD	**v**D	0 0 0 0 0	**v**B	XO	
OPCD	**v**D	UIMM	**v**B	XO	
OPCD	**v**D	SIMM	0 0 0 0 0	XO	

Specific Instructions

Name	0–5	6–10	11–15	16–20	21–31
vaddubm	04	**v**D	**v**A	**v**B	0
vadduhm	04	**v**D	**v**A	**v**B	64
vadduwm	04	**v**D	**v**A	**v**B	128
vaddcuw	04	**v**D	**v**A	**v**B	384
vaddubs	04	**v**D	**v**A	**v**B	512
vadduhs	04	**v**D	**v**A	**v**B	576
vadduws	04	**v**D	**v**A	**v**B	640
vaddsbs	04	**v**D	**v**A	**v**B	768
vaddshs	04	**v**D	**v**A	**v**B	832
vaddsws	04	**v**D	**v**A	**v**B	896
vsububm	04	**v**D	**v**A	**v**B	1024
vsubuhm	04	**v**D	**v**A	**v**B	1088
vsubuwm	04	**v**D	**v**A	**v**B	1152
vsubcuw	04	**v**D	**v**A	**v**B	1408
vsububs	04	**v**D	**v**A	**v**B	1536
vsubuhs	04	**v**D	**v**A	**v**B	1600
vsubuws	04	**v**D	**v**A	**v**B	1664
vsubsbs	04	**v**D	**v**A	**v**B	1792
vsubshs	04	**v**D	**v**A	**v**B	1856
vsubsws	04	**v**D	**v**A	**v**B	1920
vmaxub	04	**v**D	**v**A	**v**B	2
vmaxuh	04	**v**D	**v**A	**v**B	66
vmaxuw	04	**v**D	**v**A	**v**B	130
vmaxsb	04	**v**D	**v**A	**v**B	258
vmaxsh	04	**v**D	**v**A	**v**B	322
vmaxsw	04	**v**D	**v**A	**v**B	386
vminub	04	**v**D	**v**A	**v**B	514
vminuh	04	**v**D	**v**A	**v**B	578
vminuw	04	**v**D	**v**A	**v**B	642
vminsb	04	**v**D	**v**A	**v**B	770
vminsh	04	**v**D	**v**A	**v**B	834
vminsw	04	**v**D	**v**A	**v**B	898
vavgub	04	**v**D	**v**A	**v**B	1026
vavguh	04	**v**D	**v**A	**v**B	1090
vavguw	04	**v**D	**v**A	**v**B	1154
vavgsb	04	**v**D	**v**A	**v**B	1282

Specific Instructions

Name	0–5	6–10	11–15	16–20	21–30	31
vavgsh	04	**v**D	**v**A	**v**B	1346	
vavgsw	04	**v**D	**v**A	**v**B	1410	
vrlb	04	**v**D	**v**A	**v**B	4	
vrlh	04	**v**D	**v**A	**v**B	68	
vrlw	04	**v**D	**v**A	**v**B	132	
vslb	04	**v**D	**v**A	**v**B	260	
vslh	04	**v**D	**v**A	**v**B	324	
vslw	04	**v**D	**v**A	**v**B	388	
vsl	04	**v**D	**v**A	**v**B	452	
vsrb	04	**v**D	**v**A	**v**B	516	
vsrh	04	**v**D	**v**A	**v**B	580	
vsrw	04	**v**D	**v**A	**v**B	644	
vsr	04	**v**D	**v**A	**v**B	708	
vsrab	04	**v**D	**v**A	**v**B	772	
vsrah	04	**v**D	**v**A	**v**B	836	
vsraw	04	**v**D	**v**A	**v**B	900	
vand	04	**v**D	**v**A	**v**B	1028	
vandc	04	**v**D	**v**A	**v**B	1092	
vor	04	**v**D	**v**A	**v**B	1156	
vnor	04	**v**D	**v**A	**v**B	1284	
mfvscr	04	**v**D	0 0 0 0 0	0 0 0 0 0	1540	0
mtvscr	04	0 0 0 0 0	0 0 0 0 0	**v**B	1604	0
vmuloub	04	**v**D	**v**A	**v**B	8	
vmulouh	04	**v**D	**v**A	**v**B	72	
vmulosb	04	**v**D	**v**A	**v**B	264	
vmulosh	04	**v**D	**v**A	**v**B	328	
vmuleub	04	**v**D	**v**A	**v**B	520	
vmuleuh	04	**v**D	**v**A	**v**B	584	
vmulesb	04	**v**D	**v**A	**v**B	776	
vmulesh	04	**v**D	**v**A	**v**B	840	
vsum4ubs	04	**v**D	**v**A	**v**B	1544	
vsum4sbs	04	**v**D	**v**A	**v**B	1800	
vsum4shs	04	**v**D	**v**A	**v**B	1608	
vsum2sws	04	**v**D	**v**A	**v**B	1672	
vsumsws	04	**v**D	**v**A	**v**B	1928	
vaddfp	04	**v**D	**v**A	**v**B	10	

Specific Instructions

Name	0–5	6–10	11–15	16–20	21–31
vsubfp	04	**v**D	**v**A	**v**B	74
vrefp	04	**v**D	0 0 0 0 0	**v**B	266
vrsqrtefp	04	**v**D	0 0 0 0 0	**v**B	330
vexptefp	04	**v**D	0 0 0 0 0	**v**B	394
vlogefp	04	**v**D	0 0 0 0 0	**v**B	458
vrfin	04	**v**D	0 0 0 0 0	**v**B	522
vrfiz	04	**v**D	0 0 0 0 0	**v**B	586
vrfip	04	**v**D	0 0 0 0 0	**v**B	650
vrfim	04	**v**D	0 0 0 0 0	**v**B	714
vcfux	04	**v**D	UIMM	**v**B	778
vcfsx	04	**v**D	UIMM	**v**B	842
vctuxs	04	**v**D	UIMM	**v**B	906
vctsxs	04	**v**D	UIMM	**v**B	970
vmaxfp	04	**v**D	**v**A	**v**B	1034
vminfp	04	**v**D	**v**A	**v**B	1098
vmrghb	04	**v**D	**v**A	**v**B	12
vmrghh	04	**v**D	**v**A	**v**B	76
vmrghw	04	**v**D	**v**A	**v**B	140
vmrglb	04	**v**D	**v**A	**v**B	268
vmrglh	04	**v**D	**v**A	**v**B	332
vmrglw	04	**v**D	**v**A	**v**B	396
vspltb	04	**v**D	UIMM	**v**B	524
vsplth	04	**v**D	UIMM	**v**B	588
vspltw	04	**v**D	UIMM	**v**B	652
vspltisb	04	**v**D	SIMM	0 0 0 0 0	780
vspltish	04	**v**D	SIMM	0 0 0 0 0	844
vspltisw	04	**v**D	SIMM	0 0 0 0 0	908
vslo	04	**v**D	**v**A	**v**B	1036
vsro	04	**v**D	**v**A	**v**B	1100
vpkuhum	04	**v**D	**v**A	**v**B	14
vpkuwum	04	**v**D	**v**A	**v**B	78
vpkuhus	04	**v**D	**v**A	**v**B	142
vpkuwus	04	**v**D	**v**A	**v**B	206
vpkshus	04	**v**D	**v**A	**v**B	270
vpkswus	04	**v**D	**v**A	**v**B	334
vpkshss	04	**v**D	**v**A	**v**B	398

Specific Instructions

Name	0–5	6–10	11–15	16–20	21–25	26–31
vpkswss	04	**v**D	**v**A	**v**B	462	
vupkhsb	04	**v**D	0 0 0 0 0	**v**B	526	
vupkhsh	04	**v**D	0 0 0 0 0	**v**B	590	
vupklsb	04	**v**D	0 0 0 0 0	**v**B	654	
vupklsh	04	**v**D	0 0 0 0 0	**v**B	718	
vpkpx	04	**v**D	**v**A	**v**B	12	782
vupkhpx	04	**v**D	0 0 0 0 0	**v**B	846	
vupklpx	04	**v**D	0 0 0 0 0	**v**B	974	
vxor	04	**v**D	**v**A	**v**B	1220	

Table A-5. X-Form

0–5	6	7–8	9–10	11–15	16–20	21–30	31
OPCD	**v**D			**v**A	**v**B	XO	0
OPCD	**v**S			**v**A	**v**B	XO	0
OPCD	T	0 0	STRM	A	B	XO	0

Specific Instructions

Name	0–5	6	7–8	9–10	11–15	16–20	21–25	26–30	31
dst	31	T	0 0	STRM	A	B	342		0
dstt	31	1	0 0 0	tag	A	B	0		0
dstst	31	T	0 0	STRM	A	B	374		0
dststt	31	1	0 0 0	tag	A	B	11	22	0
dss	31	A	0 0	STRM	0 0 0 0 0	0 0 0 0 0	822		0
dssall	31	A	0 0	STRM	0 0 0 0 0	0 0 0 0 0	822		0
lvebx	31	**v**D			**v**A	**v**B	7		0
lvehx	31	**v**D			A	B	39		0
lvewx	31	**v**D			A	B	71		0
lvsl	31	**v**D			A	B	6		0
lvsr	31	**v**D			A	B	38		0
lvx	31	**v**D			A	B	103		0
lvxl	31	**v**D			A	B	359		0
stvebx	31	**v**S			A	B	135		0
stvehx	31	**v**S			A	B	167		0
stvewx	31	**v**S			A	B	199		0
stvx	31	**v**S			A	B	231		0
stvxl	31	**v**S			A	B	487		0

Table A-6. VXR-Form

OPCD	**v**D	**v**A	**v**B	Rc	XO

Specific Instructions

Name	0–5	6–10	11–15	16–20	21	22–31
vcmpbfp*x*	04	**v**D	**v**A	**v**B	Rc	966
vcmpeqfp*x*	04	**v**D	**v**A	**v**B	Rc	198
vcmpequb*x*	04	**v**D	**v**A	**v**B	Rc	6
vcmpequh*x*	04	**v**D	**v**A	**v**B	Rc	70
vcmpequw*x*	04	**v**D	**v**A	**v**B	Rc	134
vcmpgefp*x*	04	**v**D	**v**A	**v**B	Rc	454
vcmpgtfp*x*	04	**v**D	**v**A	**v**B	Rc	710
vcmpgtsb*x*	04	**v**D	**v**A	**v**B	Rc	774
vcmpgtsh*x*	04	**v**D	**v**A	**v**B	Rc	838
vcmpgtsw*x*	04	**v**D	**v**A	**v**B	Rc	902
vcmpgtub*x*	04	**v**D	**v**A	**v**B	Rc	518
vcmpgtuh*x*	04	**v**D	**v**A	**v**B	Rc	582
vcmpgtuw*x*	04	**v**D	**v**A	**v**B	Rc	646

A.4 Instruction Set Legend

Table A-7 provides general information on the AltiVec instruction set such as the architectural level, privilege level, and form.

Table A-7. AltiVec Instruction Set Legend

	UISA	VEA	OEA	Supervisor Level	Optional	Form
dss		√				VX
dssall		√				VX
dst	√					VX
dstst		√				VX
dststt		√				VX
dstt		√				VX
lvebx	√					X
lvehx	√					X
lvewx	√					X
lvsl	√					X
lvsr	√					X
lvx	√					X
lvxl	√					X
mfvscr	√					VX
mtvscr	√					VX
stvebx	√					X
stvehx	√					X
stvewx	√					X
stvx	√					X
stvxl	√					X
vaddcuw	√					VX
vaddfp	√					VX
vaddsbs	√					VX
vaddshs	√					VX
vaddsws	√					VX
vaddubm	√					VX
vaddubs	√					VX
vadduhm	√					VX

Table A-7. AltiVec Instruction Set Legend (Continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form
vadduhs	√					VX
vadduwm	√					VX
vadduws	√					VX
vand	√					VX
vandc	√					VX
vavgsb	√					VX
vavgsh	√					VX
vavgsw	√					VX
vavgub	√					VX
vavguh	√					VX
vavguw	√					VX
vcfux	√					VX
vcfsx	√					VX
vcmpbfp*x*	√					VXR
vcmpeqf*x*	√					VXR
vcmpequb*x*	√					VXR
vcmpequh*x*	√					VXR
vcmpequw*x*	√					VXR
vcmpgefp*x*	√					VXR
vcmpgtfp*x*	√					VXR
vcmpgtsb*x*	√					VXR
vcmpgtsh*x*	√					VXR
vcmpgtsw*x*	√					VXR
vcmpgtub*x*	√					VXR
vcmpgtuh*x*	√					VXR
vcmpgtuw*x*	√					VXR
vctsxs	√					VX
vctuxs	√					VX
vexptefp	√					VX
vlogefp	√					VX
vmaddfp	√					VA
vmaxfp	√					VX

Table A-7. AltiVec Instruction Set Legend (Continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form
vmaxsb	√					VX
vmaxsh	√					VX
vmaxsw	√					VX
vmaxub	√					VX
vmaxuh	√					VX
vmaxuw	√					VX
vmhaddshs	√					VA
vmhraddshs	√					VA
vminfp	√					VX
vminsb	√					VX
vminsh	√					VX
vminsw	√					VX
vminub	√					VX
vminuh	√					VX
vminuw	√					VX
vmladduhm	√					VA
vmrghb	√					VX
vmrghh	√					VX
vmrghw	√					VX
vmrglb	√					VX
vmrglh	√					VX
vmrglw	√					VX
vmsummbm	√					VA
vmsumshm	√					VA
vmsumshs	√					VA
vmsumubm	√					VA
vmsumuhm	√					VA
vmsumuhs	√					VA
vmulesb	√					VX
vmulesh	√					VX
vmuleub	√					VX
vmuleuh	√					VX

Table A-7. AltiVec Instruction Set Legend (Continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form
vmulosb	√					VX
vmulosh	√					VX
vmuloub	√					VX
vmulouh	√					VX
vnmsubfp	√					VA
vnor	√					VX
vor	√					VX
vperm	√					VA
vpkpx	√					VX
vpkshss	√					VX
vpkshus	√					VX
vpkswss	√					VX
vpkuhum	√					VX
vpkuhus	√					VX
vpkswus	√					VX
vpkuwum	√					VX
vpkuwus	√					VX
vrefp	√					VX
vrfim	√					VX
vrfin	√					VX
vrfip	√					VX
vrfiz	√					VX
vrlb	√					VX
vrlh	√					VX
vrlw	√					VX
vrsqrtefp	√					VX
vsel	√					VA
vsl	√					VX
vslb	√					VX
vsldoi	√					VA
vslh	√					VX
vslo	√					VX

Table A-7. AltiVec Instruction Set Legend (Continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form
vslw	√					VX
vspltb	√					VX
vsplth	√					VX
vspltisb	√					VX
vspltish	√					VX
vspltisw	√					VX
vspltw	√					VX
vsr	√					VX
vsrab	√					VX
vsrah	√					VX
vsraw	√					VX
vsrb	√					VX
vsrh	√					VX
vsro	√					VX
vsrw	√					VX
vsubcuw	√					VX
vsubfp	√					VX
vsubsbs	√					VX
vsubshs	√					VX
vsubsws	√					VX
vsububm	√					VX
vsubuhm	√					VX
vsububs	√					VX
vsubuhs	√					VX
vsubuwm	√					VX
vsubuws	√					VX
vsumsws	√					VX
vsum2sws	√					VX
vsum4sbs	√					VX
vsum4shs	√					VX
vsum4ubs	√					VX
vupkhpx	√					VX

Table A-7. AltiVec Instruction Set Legend (Continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form
vupkhsb	√					VX
vupklsh	√					VX
vupkhpx	√					VX
vupklsb	√					VX
vupklsh	√					VX
vxor	√					VX

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*.

Asynchronous exception. *Exceptions* that are caused by events external to the processor's execution. In this document, the term 'asynchronous exception' is used interchangeably with the word *interrupt*.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC architecture implements atomic accesses through the **lwarx**/**stwcx.** instruction pair.

B

BAT (block address translation) mechanism. A software-controlled array that stores the available block address translations on-chip.

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.

Block. An area of memory that ranges from 128 Kbyte to 256 Mbyte, whose size, translation, and protection attributes are controlled by the *BAT mechanism.*

Boundedly undefined. A characteristic of results of certain operations that are not rigidly prescribed by the PowerPC architecture. Boundedly-undefined results for a given operation may vary among implementations, and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be boundedly undefined, the results of executing instructions in contexts where results are allowed to be boundedly undefined are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

C

Cache. High-speed memory component containing recently-accessed data and/or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a cache block may vary among processors; the maximum block size is one *page*. In PowerPC processors, *cache coherency* is maintained on a cache-block basis. Note that the term ‘cache block’ is often used interchangeably with ‘cache line’.

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor’s cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the *cache* is bypassed and the load or store is performed to or from main memory.

Cast-outs. *Cache blocks* that must be written to memory when a cache miss causes a cache block to be replaced.

Changed bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the changed bit if any store is performed into the *page*. *See also* Page access history bits and Referenced bit.

Clear. To cause a bit or bit field to register a value of zero. *See also* Set.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *exception*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an exception).

Copy-back. An operation in which modified data in a *cache block* is copied back to memory.

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.

Direct-mapped cache. A cache in which each main memory address can appear in only one location within the cache, operates more quickly when the memory request is a cache hit.

Direct-store. Interface available on PowerPC processors only to support direct-store devices from the POWER architecture. When the T bit of a *segment descriptor* is set, the descriptor defines the region of memory that is to be used as a direct-store segment. Note that this facility is being phased out of the architecture and will not likely be supported in future devices. Therefore, software should not depend on it and new software should not use it.

Double-word swap. AltiVec processors implement a double-word swap when moving quad words between vector registers and memory. The double word swap performs an additional swap to keep vector registers and memory consistent in little-endian mode. Double-word swap is referred to as ‘swizzling’ in the AltiVec technology architecture specification. This feature is not supported by the PowerPC architecture.

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.

Exception. A condition encountered by the processor that requires special, supervisor-level processing.

Exception handler. A software routine that executes when an exception is taken. Normally, the exception handler corrects the condition that caused the exception, or performs some other meaningful task (that may include aborting the program that caused the exception). The address for each exception handler is identified by an exception vector offset defined by the architecture and a prefix selected via the MSR.

Extended opcode. A secondary opcode field generally located in instruction bits 21–30, that further defines the instruction type. All PowerPC instructions are one word in length. The most significant 6 bits of the instruction are the *primary opcode*, identifying the type of instruction. *See also* Primary opcode.

Execution synchronization. A mechanism by which all instructions in execution are architecturally complete before beginning execution (appearing to begin execution) of the next instruction. Similar to context synchronization but doesn't force the contents of the instruction buffers to be deleted and refetched.

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

Fetch. Retrieving instructions from either the cache or main memory and placing them into the instruction queue.

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 values 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.

Fully-associative. Addressing scheme where every cache location (every byte) can have any possible address.

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.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H

Harvard architecture. An architectural model featuring separate caches for instruction and data.

Hashing. An algorithm used in the *page table* search process.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

Illegal instructions. A class of instructions that are not implemented for a particular PowerPC processor. These include instructions not defined by the PowerPC architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the PowerPC architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The PowerPC architecture has many different implementations.

Implementation-dependent. An aspect of a feature in a processor's design that is defined by a processor's design specifications rather than by the PowerPC architecture.

Implementation-specific. An aspect of a feature in a processor's design that is not required by the PowerPC architecture, but for which the PowerPC architecture may provide concessions to ensure that processors that implement the feature do so consistently.

Imprecise exception. A type of *synchronous exception* that is allowed not to adhere to the precise exception model (*see* Precise exception). The PowerPC architecture allows only floating-point exceptions to be handled imprecisely.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

In-order. An aspect of an operation that adheres to a sequential model. An operation is said to be performed in-order if, at the time that it is performed, it is known to be required by the sequential execution model. *See* Out-of-order.

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Instruction parallelism. A feature of PowerPC processors that allows instructions to be processed in parallel.

Interrupt. An *asynchronous exception*. On PowerPC processors, interrupts are a special case of exceptions. *See also* asynchronous exception.

Invalid state. State of a cache entry that does not currently contain a valid copy of a cache block from memory.

K

Key bits. A set of key bits referred to as Ks and Kp in each segment register and each BAT register. The key bits determine whether supervisor or user programs can access a *page* within that *segment* or *block*.

Kill. An operation that causes a *cache block* to be invalidated.

L

L2 cache. *See* Secondary cache.

Least-significant bit (lsb). The bit of least value in an address, register, data element, or instruction encoding.

Least-significant byte (LSB). The byte 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.

Loop unrolling. Loop unrolling provides a way of increasing performance by allowing more instructions to be issued in a clock cycle. The compiler replicates the loop body to increase the number of instructions executed between a loop branch.

M

MESI (modified/exclusive/shared/invalid). *Cache coherency* protocol used to manage caches on different devices that share a memory system. Note that the PowerPC architecture does not specify the implementation of a MESI protocol to ensure cache coherency.

Memory access ordering. The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory-mapped accesses. Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU). The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Microarchitecture. The hardware details of a microprocessor's design. Such details are not defined by the PowerPC architecture.

Mnemonic. The abbreviated name of an instruction used for coding.

Modified state. When a cache block is in the modified state, it has been modified by the processor since it was copied from memory. *See* MESI.

Munging. A modification performed on an *effective address* that allows it to appear to the processor that individual aligned scalars are stored as *little-endian* values, when in fact it is stored in *big-endian* order, but at different byte addresses within double words. Note that munging affects only the effective address and not the byte order. Note also that this term is not used by the PowerPC architecture.

Multiprocessing. The capability of software, especially operating systems, to support execution on more than one processor at the same time.

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

Most-significant byte (MSB). The highest-order byte 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).

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

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

OEA (operating environment architecture). The level of the architecture that describes PowerPC memory management model, supervisor-level registers, synchronization requirements, and the exception model. It also defines the time-base feature from a supervisor-level perspective. Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

Optional. A feature, such as an instruction, a register, or an exception, that is defined by the PowerPC architecture but not required to be implemented.

Out-of-order. An aspect of an operation that allows it to be performed ahead of one that may have preceded it in the sequential model, for example, speculative operations. An operation is said to be performed out-of-order if, at the time that it is performed, it is not known to be required by the sequential execution model. *See* In-order.

Out-of-order execution. A technique that allows instructions to be issued and completed in an order that differs from their sequence in the instruction stream.

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.

P

Page. A region in memory. The OEA defines a page as a 4-Kbyte area of memory, aligned on a 4-Kbyte boundary.

Page access history bits. The *changed* and *referenced* bits in the PTE keep track of the access history within the page. The referenced bit is set by the MMU whenever the page is accessed for a read or write operation. The changed bit is set when the page is stored into. *See* Changed bit and Referenced bit.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory*. On PowerPC processors, a page fault exception condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Page table. A table in memory is comprised of *page table entries*, or PTEs. It is further organized into eight PTEs per PTEG (page table entry group). The number of PTEGs in the page table depends on the size of the page table (as specified in the SDR1 register).

Page table entry (PTE). Data structures containing information used to translate *effective address* to physical address on a 4-Kbyte page basis. A PTE consists of 8 bytes of information in a 32-bit processor and 16 bytes of information in a 64-bit processor.

Persistent data stream. A data stream is considered to be persistent when it is expected to be loaded from frequently.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise exceptions. A category of exception for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete, and subsequent instructions can be flushed and redispatched after exception handling has completed. *See* Imprecise exceptions.

Primary opcode. The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction. S*ee* Secondary opcode.

Protection boundary. A boundary between *protection domains*.

Protection domain. A protection domain is a segment, a virtual page, a BAT area, or a range of unmapped effective addresses. It is defined only when the appropriate relocate bit in the MSR (IR or DR) is 1.

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

rA. The **r**A instruction field is used to specify a GPR to be used as a source or destination.

rB. The **r**B instruction field is used to specify a GPR to be used as a source.

rD. The **r**D instruction field is used to specify a GPR to be used as a destination.

rS. The **r**S instruction field is used to specify a GPR to be used as a source.

Real address mode. An MMU mode when no address translation is performed and the *effective address* specified is the same as the physical address. The processor's MMU is operating in real address mode if its ability to perform address translation has been disabled through the MSR registers IR and/or DR bits.

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.

Referenced bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the *referenced bit* whenever the page is accessed for a read or write. *See also* Page access history bits.

Register indirect addressing. A form of addressing that specifies one GPR that contains the address for the load or store.

Register indirect with immediate index addressing. A form of addressing that specifies an immediate value to be added to the contents of a specified GPR to form the target address for the load or store.

Register indirect with index addressing. A form of addressing that specifies that the contents of two GPRs be added together to yield the target address for the load or store.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** instruction to read a memory semaphore into a GPR.

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

Scalability. The capability of an architecture to generate *implementations* specific for a wide range of purposes, and in particular implementations of significantly greater performance and/or functionality than at present, while maintaining compatibility with current implementations.

Secondary cache. A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2, cache.

Segment. A 256-Mbyte area of *virtual memory* that is the most basic memory space defined by the PowerPC architecture. Each segment is configured through a unique *segment descriptor.*

Segment descriptors. Information used to generate the interim *virtual address*. The segment descriptors reside in 16 on-chip segment registers for 32-bit implementations. For 64-bit implementations, the segment descriptors reside as *segment table entries* in a hashed segment table in memory.

Set (*v*). To write a nonzero value to a bit or bit field; the opposite of *clear.* The term ‘set’ may also be used to generally describe the updating of a bit or bit field.

Set (*n*). A subdivision of a *cache*. Cacheable data can be stored in a given location in any one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. *See* Set-associative.

Set-associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

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.

SIMD. Single instruction stream, multiple data streams. A vector instruction can operate on several data elements within a single instruction in a single functional unit. SIMD is a way to work with all the data at once (in parallel), which can make execution faster.

Simplified mnemonics. Assembler mnemonics that represent a more complex form of a common operation.

Splat. A splat instruction will take one element and replicates (splats) that value into a vector register. The purpose being to have all elements have the same value so they can be used as a constant to multiply other vector registers.

Static branch prediction. Mechanism by which software (for example, compilers) can give a hint to the machine hardware about the direction a branch is likely to take.

Sticky bit. A bit that when *set* must be cleared explicitly.

Strong ordering. A memory access model that requires exclusive access to an address before making an update, to prevent another device from using stale data.

Superscalar machine. A machine that can issue multiple instructions concurrently from a conventional linear instruction stream.

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.

Synchronization. A process to ensure that operations occur strictly *in order*. *See* Context synchronization and Execution synchronization.

Synchronous exception. An *exception* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous exceptions, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

TLB (translation lookaside buffer) A cache that holds recently-used *page table entries*.

Throughput. The measure of the number of instructions that are processed per clock cycle.

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.

Transient stream. A data stream is considered to be transient when it is likely to be referenced from infrequently.

U

UISA (user instruction set architecture). The level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and exception model as seen by user programs, and the memory and programming models.

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.

Unified cache. Combined data and instruction cache.

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

vA. The **v**A instruction field is used to specify a vector register to be used as a source or destination.

vB. The **v**B instruction field is used to specify a vector register to be used as a source.

vC. The **v**C instruction field is used to specify a vector register to be used as a source.

vD. The **v**D instruction field is used to specify a vector register to be used as a destination.

vS. The **v**S instruction field is used to specify a vector register to be used as a source.

VEA (virtual environment architecture). The level of the *architecture* that describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time-base facility from a user-level perspective. *Implementations* that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

Vector. The spatial parallel processing of short, fixed-length one-dimensional matrices performed by an execution unit.

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.

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to virtual memory is possible only when it coincides with *physical memory.*

W

Weak ordering. A memory access model that allows bus operations to be reordered dynamically, which improves overall performance and in particular reduces the effect of memory latency on instruction throughput.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.

INDEX

INDEX

INDEX

INDEX

INDEX

X

Z

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.