



PowerPC Microprocessor Family:
Vector/SIMD Multimedia Extension Technology
Programming Environments Manual

Version 2.07c

October 26, 2006



© Copyright International Business Machines Corporation 1998, 2003, 2004, 2005, 2006

All Rights Reserved
Printed in the United States of America October 2006

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both.

IBM	PowerPC
IBM Logo	PowerPC Architecture
ibm.com	

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

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

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

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

The IBM home page can be found at ibm.com

The IBM semiconductor solutions home page can be found ibm.com/chips

Version 2.07c
October 26, 2006



Contents

List of Tables	7
List of Figures	9
About This Book	15
Audience	16
Organization	16
Suggested Reading	16
General Information	17
PowerPC Documentation	17
Conventions	19
Acronyms and Abbreviations	20
Terminology Conventions	22
1. Overview	23
1.1 Vector Processing Technology Overview	25
1.1.1 64-Bit Vector Processing Technology and the 32-Bit Subset	26
1.1.2 Levels of the Vector ISA	26
1.1.3 Features Not Defined by the Vector ISA	27
1.2 Vector Processing Architectural Model	27
1.2.1 Vector Registers and Programming Model	27
1.2.2 Operand Conventions	28
1.2.2.1 Byte Ordering	28
1.2.2.2 Floating-Point Conventions	29
1.2.3 Vector Addressing Modes	30
1.2.4 Vector Instruction Set	31
1.2.5 Vector Cache Model	32
1.2.6 Vector Exception Model	32
1.2.7 Memory Management Model	32
2. Vector Register Set	33
2.1 Overview of the Vector and PowerPC Registers	33
2.2 Registers Defined by Vector ISA	35
2.2.1 Vector Register File	35
2.2.2 Vector Status and Control Register	36
2.2.3 VRSRVE Register (VRSRVE)	38
2.3 Additions to the PowerPC UISA Registers	39
2.3.1 PowerPC Condition Register	39
2.4 Additions to the PowerPC OEA Registers	40
2.4.1 VPU Bit in the PowerPC Machine State Register (MSR)	40
2.4.2 Machine Status Save/Restore Registers (SRR)	41
2.4.2.1 Machine Status Save/Restore Register 0 (SRR0)	41
2.4.2.2 Machine Status Save/Restore Register 1 (SRR1)	42



Vector/SIMD Multimedia Extension Technology

3. Operand Conventions	43
3.1 Data Organization in Memory	43
3.1.1 Aligned and Misaligned Accesses	43
3.1.2 Vector Processing Unit Byte Ordering	44
3.1.3 Vector Register and Memory Access Alignment	44
3.1.4 Quadword Data Alignment	44
3.1.4.1 Loading an Unaligned Quadword Using Vector Permute	45
3.1.4.2 Storing an Unaligned Quadword Using Vector Permute	46
3.1.4.3 Loading an Unaligned Quadword Using Load Vector Left/Right	47
3.1.4.4 Storing an Unaligned Quadword Using Store Vector Left/Right Indexed	48
3.1.4.5 Scalar Loads and Stores	48
3.1.4.6 Misaligned Scalar Loads and Stores	48
3.2 Vector Floating-Point Instructions—UISA	49
3.2.1 Floating-Point Modes	49
3.2.1.1 Java Mode	49
3.2.1.2 Non-Java Mode	50
3.2.2 Floating-Point Infinities	50
3.2.3 Floating-Point Rounding	50
3.2.4 Floating-Point Exceptions	50
3.2.4.1 NaN Operand Exception	51
3.2.4.2 Invalid Operation Exception	52
3.2.4.3 Zero Divide Exception	52
3.2.4.4 Log of Zero Exception	52
3.2.4.5 Overflow Exception	52
3.2.4.6 Underflow Exception	53
3.2.5 Floating-Point NaNs	53
3.2.5.1 NaN Precedence	53
3.2.5.2 SNaN Arithmetic	53
3.2.5.3 QNaN Arithmetic	53
3.2.5.4 NaN Conversion to Integer	54
3.2.5.5 NaN Production	54
4. Addressing Modes and Instruction Set Summary	55
4.1 Conventions	56
4.1.1 Execution Model	56
4.1.2 Computation Modes	56
4.1.3 Classes of Instructions	56
4.1.4 Memory Addressing	57
4.1.4.1 Memory Operands	57
4.1.4.2 Effective Address Calculation	57
4.2 Vector UISA Instructions	58
4.2.1 Vector Integer Instructions	58
4.2.1.1 Saturation Detection	58
4.2.1.2 Vector Integer Arithmetic Instructions	59
4.2.1.3 Vector Integer Compare Instructions	64
4.2.1.4 Vector Integer Logical Instructions	66
4.2.1.5 Vector Integer Rotate and Shift Instructions	66
4.2.2 Vector Floating-Point Instructions	67
4.2.2.1 Floating-Point Division and Square-Root	68
4.2.2.2 Floating-Point Arithmetic Instructions	69



4.2.2.3 Vector Floating-Point Multiply-Add Instructions	70
4.2.2.4 Vector Floating-Point Rounding and Conversion Instructions	71
4.2.2.5 Vector Floating-Point Compare Instructions	72
4.2.2.6 Vector Floating-Point Estimate Instructions	75
4.2.3 Vector Load and Store Instructions	75
4.2.3.1 Alignment	75
4.2.3.2 Vector Load and Store Address Generation	76
4.2.3.3 Vector Load Instructions	78
4.2.3.4 Vector Store Instructions	81
4.2.4 Control Flow	82
4.2.5 Vector Permutation and Formatting Instructions	82
4.2.5.1 Vector Pack Instructions	82
4.2.5.2 Vector Unpack Instructions	84
4.2.5.3 Vector Merge Instructions	85
4.2.5.4 Vector Splat Instructions	86
4.2.5.5 Vector Permute Instructions	86
4.2.5.6 Vector Select Instruction	87
4.2.5.7 Vector Shift Instructions	87
4.2.6 Processor Control Instructions—UISA	89
4.2.6.1 Vector Status and Control Register Instructions	90
4.3 Vector VEA Instructions	91
4.3.1 Memory Control Instructions—VEA	91
4.3.2 User-Level Cache Instructions—VEA	91
4.3.3 Recommended Simplified Mnemonics	93
5. Cache, Exceptions, and Memory Management	95
5.1 PowerPC Shared Memory	95
5.1.1 PowerPC Memory Access Ordering	95
5.2 VPU Memory Bandwidth Management	96
5.2.1 Software-Directed Prefetch	96
5.2.1.1 Data Stream Touch (dst)	96
5.2.1.2 Transient Streams (dstt)	98
5.2.1.3 Storing to Streams (dstst)	99
5.2.1.4 Stopping Streams (dss)	99
5.2.1.5 Exception Behavior of Prefetch Streams	100
5.2.1.6 Synchronization Behavior of Streams	101
5.2.1.7 Address Translation for Streams	101
5.2.1.8 Stream Usage Notes	101
5.2.1.9 Stream Implementation Assumptions	103
5.2.2 Prioritizing Cache Block Replacement	103
5.2.3 Partially Executed Vector Instructions	103
5.3 DSI Exception—Data Address Breakpoint	104
5.4 VPU Unavailable Exception (0x00F20)	104
6. Vector Processing Instructions	107
6.1 Instruction Formats	107
6.1.1 Instruction Fields	107
6.1.2 Notation and Conventions	108
6.2 Vector Instruction Set	113

**Vector/SIMD Multimedia Extension Technology**

Appendix A. Vector Processing Instruction Set Listings	291
A.1 Instructions Sorted by Mnemonic	291
A.2 Instructions Sorted by Opcode	296
A.3 Instructions Sorted by Form	302
A.4 Instruction Set Legend	309
Glossary	315
Revision Log	327



List of Tables

Table i.	Acronyms and Abbreviated Terms	20
Table ii.	Terminology Conventions	22
Table iii.	Instruction Field Conventions	22
Table 2-1.	VSCR Field Descriptions	37
Table 2-2.	VRSAVE Bit Settings	38
Table 2-3.	CR6 Field Bit Settings for Vector Compare Instructions	39
Table 2-5.	MSR Bit Settings	40
Table 2-4.	MSR Bit Settings Affected by the VPU	41
Table 3-1.	Memory Operand Alignment	43
Table 4-1.	Vector Integer Arithmetic Instructions	59
Table 4-2.	CR6 Field Bit Settings for Vector Integer Compare Instructions	65
Table 4-3.	Vector Integer Compare Instructions	65
Table 4-4.	Vector Integer Logical Instructions	66
Table 4-5.	Vector Integer Rotate Instructions	66
Table 4-6.	Vector Integer Shift Instructions	67
Table 4-7.	Floating-Point Arithmetic Instructions	69
Table 4-8.	Vector Floating-Point Multiply-Add Instructions	70
Table 4-9.	Vector Floating-Point Rounding and Conversion Instructions	71
Table 4-10.	Common Mathematical Predicates	72
Table 4-11.	Other Useful Predicates	72
Table 4-12.	Vector Floating-Point Compare Instructions	73
Table 4-13.	Vector Floating-Point Estimate Instructions	75
Table 4-14.	Effective Address Alignment	76
Table 4-15.	Vector Load Instructions	78
Table 4-16.	Vector Load Instructions Supporting Alignment	80
Table 4-17.	Vector Store Instructions	81
Table 4-18.	Vector Pack Instructions	83
Table 4-19.	Vector Unpack Instructions	84
Table 4-20.	Vector Merge Instructions	85
Table 4-21.	Vector Splat Instructions	86
Table 4-22.	Vector Permute Instruction	86
Table 4-23.	Vector Select Instruction	87
Table 4-24.	Vector Shift Instructions	87
Table 4-25.	Coding Various Shifts and Rotates with the vsidoi Instruction	88
Table 4-26.	Move to/from Condition Register Instructions	90
Table 4-27.	User-Level Cache Instructions	92
Table 4-28.	Simplified Mnemonics for Data Stream Touch (dst)	94
Table 5-1.	VPU Unavailable Exception—Register Settings	104

**Vector/SIMD Multimedia Extension Technology**

Table 5-2.	Exception Priorities (Synchronous/Precise Exceptions)	105
Table 6-1.	Instruction Syntax Conventions	107
Table 6-2.	Notation and Conventions	108
Table 6-3.	Instruction Field Conventions	112
Table 6-4.	Precedence Rules	112
Table 6-5.	vexptefp Operation with Various Values of Element vB	180
Table 6-6.	vlogefp with Special Values	182
Table 6-7.	vrefp —Special Values of the Element in vB	235
Table 6-8.	vrsqrtefp —Special Values	244
Table A-1.	Complete Instruction List Sorted by Mnemonic	291
Table A-2.	Instructions Sorted by Opcode	296
Table A-3.	VA-Form	302
Table A-4.	VX-Form	303
Table A-5.	X-Form	307
Table A-6.	VXR-Form	308
Table A-7.	Vector Instruction Set Legend	309



List of Figures

Figure 1-1.	High-Level Structural Overview of PowerPC with Vector Processing Technology	25
Figure 1-2.	Vector /SIMD Multimedia Extension Top-Level Diagram	28
Figure 1-3.	Big-Endian Byte Ordering for a Vector Register	29
Figure 1-4.	Intraelement Example, vaddsws	30
Figure 1-5.	Interelement Example, vperm	30
Figure 2-1.	Programming Model—All Registers	34
Figure 2-2.	Vector Register File (VSCR)	35
Figure 2-3.	Vector Status and Control Register (VSCR)	36
Figure 2-4.	VSCR Moved to a Vector Register	36
Figure 2-5.	Saving/Restoring the Vector Context Register (VRSAVE)	38
Figure 2-6.	Condition Register (CR)	39
Figure 2-7.	Machine State Register (MSR)	40
Figure 2-8.	Machine Status Save/Restore Register 0 (SRR0)	41
Figure 2-9.	Machine Status Save/Restore Register 1 (SRR1)	42
Figure 3-1.	Misaligned Vector	45
Figure 4-1.	Register Indirect with Index Addressing for Loads/Stores	77
Figure 5-1.	Format of rB in dst Instruction	97
Figure 5-2.	Data Stream Touch	98
Figure 6-2.	vaddsb s— Add Saturating Sixteen Signed Integer Elements	113
Figure 6-1.	Format for Instruction Description Page	113
Figure 6-3.	dst —Data Stream Touch	115
Figure 6-4.	dstst —Data Stream Touch for Store	117
Figure 6-5.	Effects of Example Load/Store Instructions	120
Figure 6-6.	Load Vector for Shift Left	127
Figure 6-7.	Instruction vperm Used in Aligning Data	128
Figure 6-8.	vaddcuw —Determine Carries of Four Unsigned Integer Adds	143
Figure 6-9.	vaddfp —Add Four Floating-Point Elements	144
Figure 6-10.	vaddsb s— Add Saturating Sixteen Signed Integer Elements	145
Figure 6-11.	vaddsh s— Add Saturating Eight Signed Integer Elements	146
Figure 6-12.	vaddsw s—Add Saturating Four Signed Integer Elements	147
Figure 6-13.	vaddubm —Add Sixteen Integer Elements	148
Figure 6-14.	vaddubs —Add Saturating Sixteen Unsigned Integer Elements	149
Figure 6-15.	vadduhm —Add Eight Integer Elements	150
Figure 6-16.	vadduhs —Add Saturating Eight Unsigned Integer Elements	151
Figure 6-17.	vadduw m—Add Four Integer Elements	152
Figure 6-18.	vadduws —Add Saturating Four Unsigned Integer Elements	153
Figure 6-19.	vand —Logical Bitwise AND	154
Figure 6-20.	vandc —Logical Bitwise AND with Complement	155



Vector/SIMD Multimedia Extension Technology

Figure 6-21. vavgbsb — Average Sixteen Signed Integer Elements	156
Figure 6-22. vavgsh —Average Eight Signed Integer Elements	157
Figure 6-23. vavgsw — Average Four Signed Integer Elements	158
Figure 6-24. vavgub —Average Sixteen Unsigned Integer Elements	159
Figure 6-25. vavgsh — Average Eight Signed Integer Elements	160
Figure 6-26. vavguw —Average Four Unsigned Integer Elements	161
Figure 6-27. vcfsx —Convert Four Signed Integer Elements to Four Floating-Point Elements	162
Figure 6-28. vcfux —Convert Four Unsigned Integer Elements to Four Floating-Point Elements	163
Figure 6-29. vcmpbfp —Compare Bounds of Four Floating-Point Elements	165
Figure 6-30. vcmpeqfp —Compare Equal of Four Floating-Point Elements	166
Figure 6-31. vcmpequb —Compare Equal of Sixteen Integer Elements	167
Figure 6-32. vcmpequh —Compare Equal of Eight Integer Elements	168
Figure 6-33. vcmpequw —Compare Equal of Four Integer Elements	169
Figure 6-34. vcmpgefp —Compare Greater-Than-or-Equal of Four Floating-Point Elements	170
Figure 6-35. vcmpgtfp —Compare Greater-Than of Four Floating-Point Elements	171
Figure 6-36. vcmpgtsb —Compare Greater-Than of Sixteen Signed Integer Elements	172
Figure 6-37. vcmpgtsh —Compare Greater-Than of Eight Signed Integer Elements	173
Figure 6-38. vcmpgtsw —Compare Greater-Than of Four Signed Integer Elements	174
Figure 6-39. vcmpgtub —Compare Greater-Than of Sixteen Unsigned Integer Elements	175
Figure 6-40. vcmpgtuh —Compare Greater-Than of Eight Unsigned Integer Elements	176
Figure 6-41. vcmpgtuw —Compare Greater-Than of Four Unsigned Integer Elements	177
Figure 6-42. vctsxs —Convert Four Floating-Point Elements to Four Signed Integer Elements	178
Figure 6-43. vctuxs —Convert Four Floating-Point Elements to Four Unsigned Integer Elements	179
Figure 6-44. vexptefp —2 Raised to the Exponent Estimate Floating-Point for Four Floating-Point Elements	181
Figure 6-45. vlogefp —Log2 Estimate Floating-Point for Four Floating-Point Elements	183
Figure 6-46. vmaddfp —Multiply-Add Four Floating-Point Elements	184
Figure 6-47. vmaxfp —Maximum of Four Floating-Point Elements	185
Figure 6-48. vmaxsb —Maximum of Sixteen Signed Integer Elements	186
Figure 6-49. vmaxsh —Maximum of Eight Signed Integer Elements	187
Figure 6-50. vmaxsw —Maximum of Four Signed Integer Elements	188
Figure 6-51. vmaxub —Maximum of Sixteen Unsigned Integer Elements	189
Figure 6-52. vmaxuh —Maximum of Eight Unsigned Integer Elements	190
Figure 6-53. vmaxuw —Maximum of Four Unsigned Integer Elements	191
Figure 6-54. vhaddshs —Multiply-High and Add Eight Signed Integer Elements	192
Figure 6-55. vhhraddshs —Multiply-High Round and Add Eight Signed Integer Elements (16-Bit)	193
Figure 6-56. vminfp —Minimum of Four Floating-Point Elements	194
Figure 6-57. vminsbt —Minimum of Sixteen Signed Integer Elements	195
Figure 6-58. vminsh —Minimum of Eight Signed Integer Elements	196



Figure 6-59.	vminsw —Minimum of Four Signed Integer Elements	197
Figure 6-60.	vminub —Minimum of Sixteen Unsigned Integer Elements	198
Figure 6-61.	vminuh —Minimum of Eight Unsigned Integer Elements	199
Figure 6-62.	vminuw —Minimum of Four Unsigned Integer Elements	200
Figure 6-63.	vmladduhm —Multiply-Add of Eight Integer Elements	201
Figure 6-64.	vmrghb —Merge Eight High-Order Elements	202
Figure 6-65.	vmrghh —Merge Four High-Order Elements	203
Figure 6-66.	vmrghw —Merge Two High-Order Elements	204
Figure 6-67.	vmrglb —Merge Eight Low-Order Elements	205
Figure 6-68.	vmrglh —Merge Four Low-Order Elements	206
Figure 6-69.	vmrglw —Merge Two Low-Order Elements	207
Figure 6-70.	vmsummbm —Multiply-Sum of Integer Elements	208
Figure 6-71.	vmsumshm —Multiply-Sum of Signed Integer Elements	209
Figure 6-72.	vmsumshs —Multiply-Sum of Signed Integer Elements	210
Figure 6-73.	vmsumubm —Multiply-Sum of Unsigned Integer Elements	211
Figure 6-74.	vmsumuhm —Multiply-Sum of Unsigned Integer Elements	212
Figure 6-75.	vmsumuhs —Multiply-Sum of Unsigned Integer Elements	213
Figure 6-76.	vmulesb —Even Multiply of Eight Signed Integer Elements	214
Figure 6-77.	vmulesb —Even Multiply of Four Signed Integer Elements	215
Figure 6-78.	vmuleub —Even Multiply of Eight Unsigned Integer Elements	216
Figure 6-79.	vmuleuh —Even Multiply of Four Unsigned Integer Elements	217
Figure 6-80.	vmulosb —Odd Multiply of Eight Signed Integer Elements	218
Figure 6-81.	vmuleuh —Odd Multiply of Four Unsigned Integer Elements	219
Figure 6-82.	vmuloub —Odd Multiply of Eight Unsigned Integer Elements	220
Figure 6-83.	vmulouh —Odd Multiply of Four Unsigned Integer Elements	221
Figure 6-84.	vnmsubfp —Negative Multiply-Subtract of Four Floating-Point Elements	222
Figure 6-85.	vnor —Bitwise NOR of 128-Bit Vector	223
Figure 6-86.	vor —Bitwise OR of 128-Bit Vector	224
Figure 6-87.	vperm —Concatenate Sixteen Integer Elements	225
Figure 6-88.	vpkpx —Pack Eight Elements (32-Bit) to Eight Elements (16-Bit)	226
Figure 6-89.	vpkshus —Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)	228
Figure 6-90.	vpkswss —Pack Eight Signed Integer Elements (32-Bit) to Eight Signed Integer Elements (16-Bit)	229
Figure 6-91.	vpkswus —Pack Eight Signed Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)	230
Figure 6-92.	vpkuhum —Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)	231
Figure 6-93.	vpkuhus —Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)	232



Vector/SIMD Multimedia Extension Technology

Figure 6-94. vpkuwum —Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)	233
Figure 6-95. vpkuwum —Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)	234
Figure 6-96. vrefp —Reciprocal Estimate of Four Floating-Point Elements	236
Figure 6-97. vrfim —Round to Minus Infinity of Four Floating-Point Integer Elements	237
Figure 6-98. vrfin —Nearest Round to Nearest of Four Floating-Point Integer Elements	238
Figure 6-99. vrfip —Round to Plus Infinity of Four Floating-Point Integer Elements	239
Figure 6-100. vrfiz —Round-to-Zero of Four Floating-Point Integer Elements	240
Figure 6-101. vrlb —Left Rotate of Sixteen Integer Elements	241
Figure 6-102. vrlh —Left Rotate of Eight Integer Elements	242
Figure 6-103. vrlw —Left Rotate of Four Integer Elements	243
Figure 6-104. vsqrtefp —Reciprocal Square Root Estimate of Four Floating-Point Elements	244
Figure 6-105. vsel —Bitwise Conditional Select of Vector Contents	245
Figure 6-106. vsl —Shift Bits Left in Vector	246
Figure 6-107. vslb —Shift Bits Left in Sixteen Integer Elements	247
Figure 6-108. vldoi —Shift Left by Bytes Specified	248
Figure 6-109. vsh —Shift Bits Left in Eight Integer Elements	249
Figure 6-110. vslo —Left Byte Shift of Vector	250
Figure 6-111. vslw —Shift Bits Left in Four Integer Elements	251
Figure 6-112. vspltb —Copy Contents to Sixteen Elements	252
Figure 6-113. vsplth —Copy Contents to Eight Elements	253
Figure 6-114. vspltisb —Copy Value into Sixteen Signed Integer Elements	254
Figure 6-115. vspltish —Copy Value to Eight Signed Integer Elements	255
Figure 6-116. vspltisw —Copy Value to Four Signed Elements	256
Figure 6-117. vspltw —Copy contents to Four Elements	257
Figure 6-118. vsr —Shift Bits Right for Vectors	259
Figure 6-119. vsrab —Shift Bits Right in Sixteen Integer Elements	260
Figure 6-120. vsrah —Shift Bits Right for Eight Integer Elements	261
Figure 6-121. vsraw —Shift Bits Right in Four Integer Elements	262
Figure 6-122. vsrb —Shift Bits Right in Sixteen Integer Elements	263
Figure 6-123. vsrh —Shift Bits Right for Eight Integer Elements	264
Figure 6-124. vsro —Vector Shift Right Octet	265
Figure 6-125. vsrw —Shift Bits Right in Four Integer Elements	266
Figure 6-126. vsubcuw —Subtract Carryout of Four Unsigned Integer Elements	267
Figure 6-127. vsubfp —Subtract Four Floating Point Elements	268
Figure 6-128. vsubsbs —Subtract Sixteen Signed Integer Elements	269
Figure 6-129. vsubshs —Subtract Eight Signed Integer Elements	270
Figure 6-130. vsubsws —Subtract Four Signed Integer Elements	271



Figure 6-131. vsububm —Subtract Sixteen Integer Elements	272
Figure 6-132. vsububs —Subtract Sixteen Unsigned Integer Elements	273
Figure 6-133. vsubuhm —Subtract Eight Integer Elements	274
Figure 6-134. vsubuhs —Subtract Eight Signed Integer Elements	275
Figure 6-135. vsubuwm —Subtract Four Integer Elements	276
Figure 6-136. vsubuws —Subtract Four Signed Integer Elements	277
Figure 6-137. vsumsws —Sum Four Signed Integer Elements	278
Figure 6-138. vsum2sws —Two Sums in the Four Signed Integer Elements	279
Figure 6-139. vsum4sbs —Four Sums in the Integer Elements	280
Figure 6-140. vsum4shs —Four Sums in the Integer Elements	281
Figure 6-141. vsum4ubs —Four Sums in the Integer Elements	282
Figure 6-142. vupkhpx —Unpack High-Order Elements (16 bit) to Elements (32-Bit)	283
Figure 6-143. vupkhsb —Unpack Hlgh-Order Signed Integer Elements (8-Bit) to Signed Integer Elements (16-Bit)	284
Figure 6-144. vupkhsh —Unpack Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)	285
Figure 6-145. vupklpx —Unpack Low-Order Elements (16-Bit) to Elements (32-Bit)	286
Figure 6-146. vupklsb —Unpack Low-Order Elements (8-Bit) to Elements (16-Bit)	287
Figure 6-147. vupklsh —Unpack Low-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)	288
Figure 6-148. vxor —Bitwise XOR (128-Bit)	289



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 the vector/single instruction stream, multiple data stream (SIMD) multimedia extension (vector processing) technology. This book describes how the vector processing technology relates to the 64-bit portion of the PowerPC Architecture.™

To locate any published errata or updates for this document, see <http://www.ibm.com/chips>.

The *Vector/SIMD Multimedia Extension Technology Programming Environments Manual (Vector Processing PEM)* is used as a reference guide for programmers. The *Vector Processing PEM* provides a description that includes the instruction format and figures to help in understanding how each 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 of the discussions about the vector processing technology are at the UI SA level.

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

- PowerPC user instruction set architecture (UI SA)—The UI SA defines the level of the architecture to which user-level software should conform. The UI SA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers.
- 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 UI SA, 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 UI SA 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 vector processing technology implementations on PowerPC processors. It must be kept in mind that each PowerPC processor is unique in its implementation of the vector 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.



Vector/SIMD Multimedia Extension Technology

Audience

This manual is intended for system software and hardware developers and application programmers who want to develop products using the vector/SIMD multimedia 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 reduced instruction set computer (RISC) processing.

This book describes how the vector processing technology interacts with the 64-bit portion of the PowerPC Architecture.

Organization

A summary and a brief description of the major sections of this manual follows:

- *Chapter 1, Overview* is useful for those who want a general understanding of the features and functions of the vector processing technology. This chapter provides an overview of how the vector technology defines the register set, operand conventions, addressing modes, instruction set, cache model, and exception model.
- *Chapter 2, Vector 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 vector processing technology registers and how they interact with the other PowerPC registers.
- *Chapter 3, Operand Conventions* describes how the vector technology interacts with the PowerPC conventions for storing data in memory, including information about alignment and single-precision floating-point conventions.
- *Chapter 4, Addressing Modes and Instruction Set Summary* provides an overview of the vector processing technology addressing modes and a brief description of the vector processing 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, Vector Processing Instructions* functions as a handbook for the vector 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., Vector Processing Instruction Set Listings* lists all the vector instructions. Instructions are grouped according to mnemonic, opcode, and form.
- This manual also includes a glossary.

Suggested Reading

This section lists additional reading that provides background for the information in this manual, as well as general information about the vector processing 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 via many online bookstores.
 - *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.1994.
Note: This book has been superseded by *PowerPC Architecture Books I-III, Version 2.02*, which are available at www.ibm.com/powerpc.
 - *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.
- *PowerPC Programming for Intel Programmers*, by Kip McClanahan; IDG Books Worldwide, Inc., 919 East Hillsdale Boulevard, Suite 400, Foster City, CA, 94404.

PowerPC Documentation

Some additional PowerPC documentation is available via the internet at <http://www.ibm.com/chips/techlib>.

- User's manuals—These books provide details about individual PowerPC implementations and are intended to be used in conjunction with the *Programming Environments Manuals*.
- Addenda/errata to user's manuals—Because some processors have follow-on parts, an addendum might be 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.
- Programming environments manuals (PEM)—These books provide information about resources defined by the PowerPC Architecture that are common to PowerPC processors. There are several versions, one that describes the functionality of the 32-bit architecture model and one that describes the 64-bit model.
 - *PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors*.
 - *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*.
- Datasheets—Datasheets provide specific data regarding bus timing, signal behavior, and ac, dc, and thermal characteristics, as well as other design considerations for each PowerPC implementation.
- *PowerPC Microprocessor Family: The Programmer's Reference Guide: MPRPPCPRG-01* is a concise reference that includes the register summary, memory control model, exception vectors, and the PowerPC instruction set.
- *PowerPC Quick Reference Guide*: This brochure is a quick reference guide to IBM's portfolio of industry-leading PowerPC technology. It includes highlights and specifications for the PowerPC 405, PowerPC 440, PowerPC 750, and PowerPC 970 based standard products.
- *PowerPC User Instruction Set Architecture, Book I* (Version 2.02)—This book defines the instructions, registers, and so forth, typically used by application programs (for example, Branch, Load, Store, and Arith-



Vector/SIMD Multimedia Extension Technology

metic instructions; general purpose and floating-point registers). All Book I facilities and instructions are nonprivileged (are available in problem state).

- *PowerPC Virtual Environment Architecture, Book II* (Version 2.02)–This book defines the memory model (caches, memory access ordering, and so forth) and related instructions, such as the instructions used to manage caches and to synchronize memory accesses when memory is shared among programs running on different processors. All Book II facilities and instructions are nonprivileged, but they are typically used via operating-system-provided library subroutines, which application programs call as needed.
- *PowerPC Operating Environment Architecture, Book III* (Version 2.02) –This book defines the privileged facilities and related instructions (address translation, memory protection, interruptions, and so forth). Nearly all Book III facilities and instructions are privileged. (Those that are nonprivileged are described also in Book I or II, but only at the level needed by application programmers.)
- 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.

Additional literature on PowerPC implementations is being released as new processors become available.

For a current list of PowerPC documentation, see <http://www.ibm.com/chips>.



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.
<i>italics</i>	Italics indicate variable instruction parameters, for example, bcctrx . Book titles in text are set in italics.
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, rB	Instruction syntax used to identify a source general-purpose register (GPR)
rD	Instruction syntax used to identify a destination GPR
frA, frB, frC	Instruction syntax used to identify a source floating-point register (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[DR] refers to the data address translation enable bit in the machine state register.
vA, vB, vC	Instruction syntax used to identify a source Vector Register (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 numeric value
¬	NOT logical operator
&	AND logical operator
	OR logical operator
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 *Chapter 6, Vector Processing Instructions*.

Vector/SIMD Multimedia Extension Technology**Acronyms and Abbreviations**

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

Table i. Acronyms and Abbreviated Terms (Page 1 of 2)

Term	Meaning
ALU	Arithmetic logic unit
ASR	Address Space Register
BPU	Branch processing unit
CR	Condition Register
CTR	Count Register
DAR	Data Address Register
DEC	Decrementer Register
DSISR	Register used for determining the source of a data storage interrupt (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 quadword
lsq	Least-significant quadword
MERSI	Modified/exclusive/reserved/shared/invalid—cache coherency protocol
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSQ	Most-significant quadword
msq	Most-significant quadword
MSR	Machine State Register
NaN	Not a number



Table i. Acronyms and Abbreviated Terms (Page 2 of 2)

Term	Meaning
NIA	Next instruction address
No-op	No operation
OEA	Operating environment architecture
PTEG	Page table entry group
RISC	Reduced instruction set computing
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
VPU	Vector processing unit
VR	Vector Register

Vector/SIMD Multimedia Extension Technology**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 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	Doubleword 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	crbA, crbB, crbD (respectively)
BF, BFA	crfD, crfS (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	frA, frB, frC, frD, frS (respectively)
FXM	CRM
RA, RB, RT, RS	rA, rB, rD, rS (respectively)
SI	SIMM
U	IMM
UI	UIMM
VA, VB, VT, VS	vA, vB, vD, vS (respectively)
VEC	Vector processing technology
/, //, ///	0...0 (shaded)



1. Overview

The vector/single instruction stream, multiple data stream (SIMD) multimedia extension technology provides a software model that accelerates the performance of various software applications and runs on reduced instruction set computing (RISC) microprocessors. The vector processing technology is a short vector parallel architecture that extends the instruction set architecture (ISA) of the PowerPC Architecture. The Vector ISA is based on separate vector/SIMD-style execution units that have high data parallelism. That is, the vector processing 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 not to impede additional parallelism through superscalar dispatch to multiple execution units or multithreaded execution unit pipelines.

The Vector ISA supports the following audio and visual applications:

- 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 plain old telephone service (POTS) lines and inserts it into the Internet. A digital subscriber line access multiplexer (DSLAM) pulls data off at a switch and immediately routes it to the Internet. This allows it to concentrate asymmetric digital subscriber line (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:
 - Multichannel modems.
 - Software modem: V.34, 56K.
 - Data encryption: RSA.
 - Modem banks can use the vector processing technology to replace signal processors in digital signal processor (DSP) farms.
- 2D and 3D graphics: QuickDraw, OpenGL, virtual reality modeling language (VRML), games, entertainment, high-precision computer aided design (CAD).
- Virtual reality.
- High-fidelity audio: 3D audio, AC-3. Hi-Fi audio uses the vector processor's floating-point unit (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.
- Base station processing. Cellular base station 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.



Vector/SIMD Multimedia Extension Technology

All vector 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 that 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 vector processing 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 vector 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. Vector Processing Instruction Set Listings*.
 - Register set. The vector programming model defines new vector registers, additions to the PowerPC register set, and how existing PowerPC registers are affected by the vector processing technology. The model also discusses memory conventions, including details regarding the byte ordering for quadwords.
- Memory model. The vector processing technology specifies additional cache management instructions. That is, a program can execute vector software instructions that indicate when a sequence of memory units (data stream/stream) are likely to be accessed.
- Exception model. To ensure efficiency, the vector processing technology provides only a Vector Processing Unit (VPU) Unavailable Interrupt (VUI) exception, a data storage interrupt (DSI) exception, and a trace exception (if implemented). There are no exceptions other than DSI exceptions on loads and stores. The vector instructions can cause PowerPC exceptions.
- Memory management model. The memory model for the vector processing technology is the same as it is implemented for the PowerPC Architecture. Vector processing memory accesses are always assumed to be aligned. If an operand is unaligned, additional vector 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 vector processing technology.

This chapter provides an overview of the major characteristics of the vector processing 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

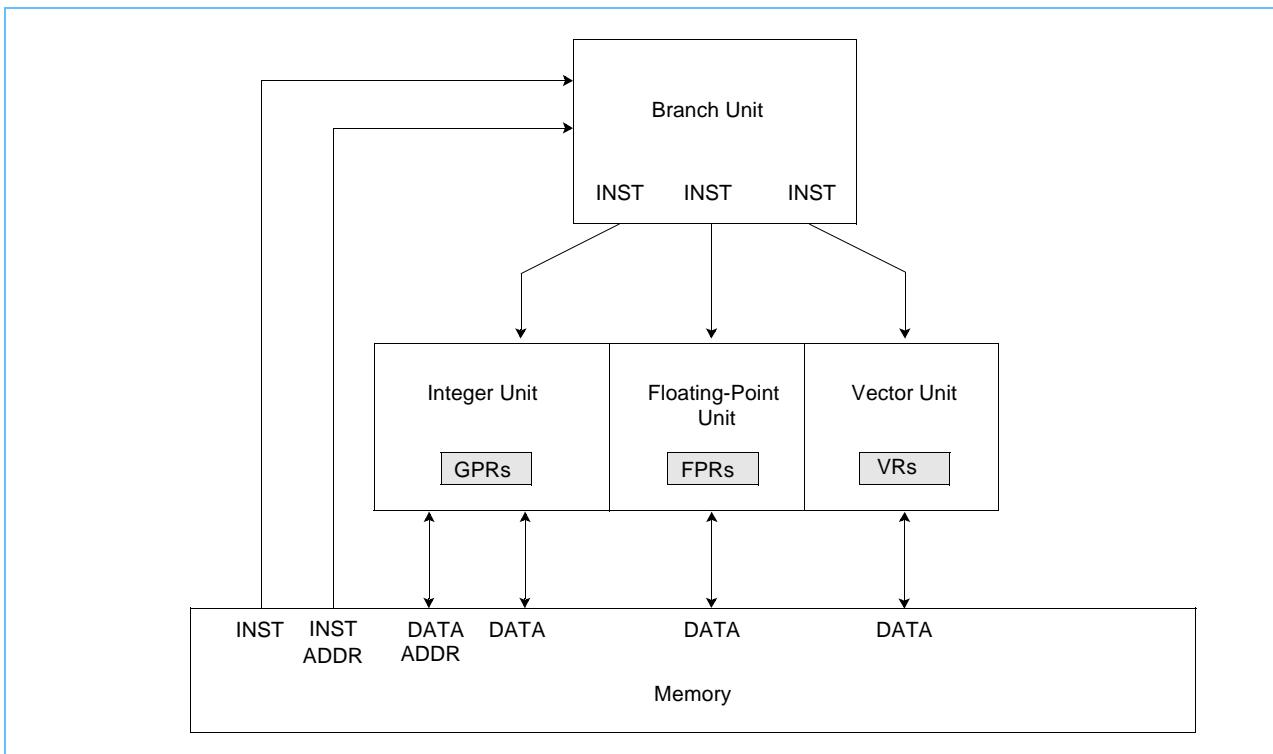
1.1 Vector Processing Technology Overview

The vector processing technology's SIMD-style extension provides an approach to accelerating the processing of data streams. Using the vector 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 vector processing unit (VPU), and minimizing bandwidth and latency memory access bottlenecks.

Vector processing 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. The new vector execution unit provides highly parallel operations, allowing for simultaneous execution of multiple operations in a single clock cycle.

The vector processing 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 vector processing technology is added 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 vector processing technology.

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



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



Vector/SIMD Multimedia Extension Technology

The vector processing technology defines the following features:

- Fixed 128-bit wide vector length that can be subdivided into sixteen 8-bit bytes, eight 16-bit halfwords, 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 the result 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).
- Vector instructions provide a vector compare and select mechanism to implement conditional execution as the preferred way to control data flow in vector processing programs.
- Enhanced cache/memory interface.

1.1.1 64-Bit Vector Processing Technology and the 32-Bit Subset

The vector processing technology supports the following modes of PowerPC operations:

- 64-bit implementations/64-bit mode—The vector processing technology defines interactions with the PowerPC 64-bit registers.
- 64-bit implementations/32-bit mode—The vector processing 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, see Chapter 1, “Overview,” in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*. This book describes the 64-bit PowerPC Architecture mode. 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 Levels of the Vector ISA

The Vector 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 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.
- 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.



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 vector processing technology defines instructions at the UISA and VEA levels.

1.1.3 Features Not Defined by the Vector ISA

Because flexibility is an important design goal of the vector processing technology, there are many aspects of the microprocessor design, typically relating to the hardware implementation, that the Vector ISA does not define; for example, the number and the nature of execution units. The Vector 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 Vector 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.

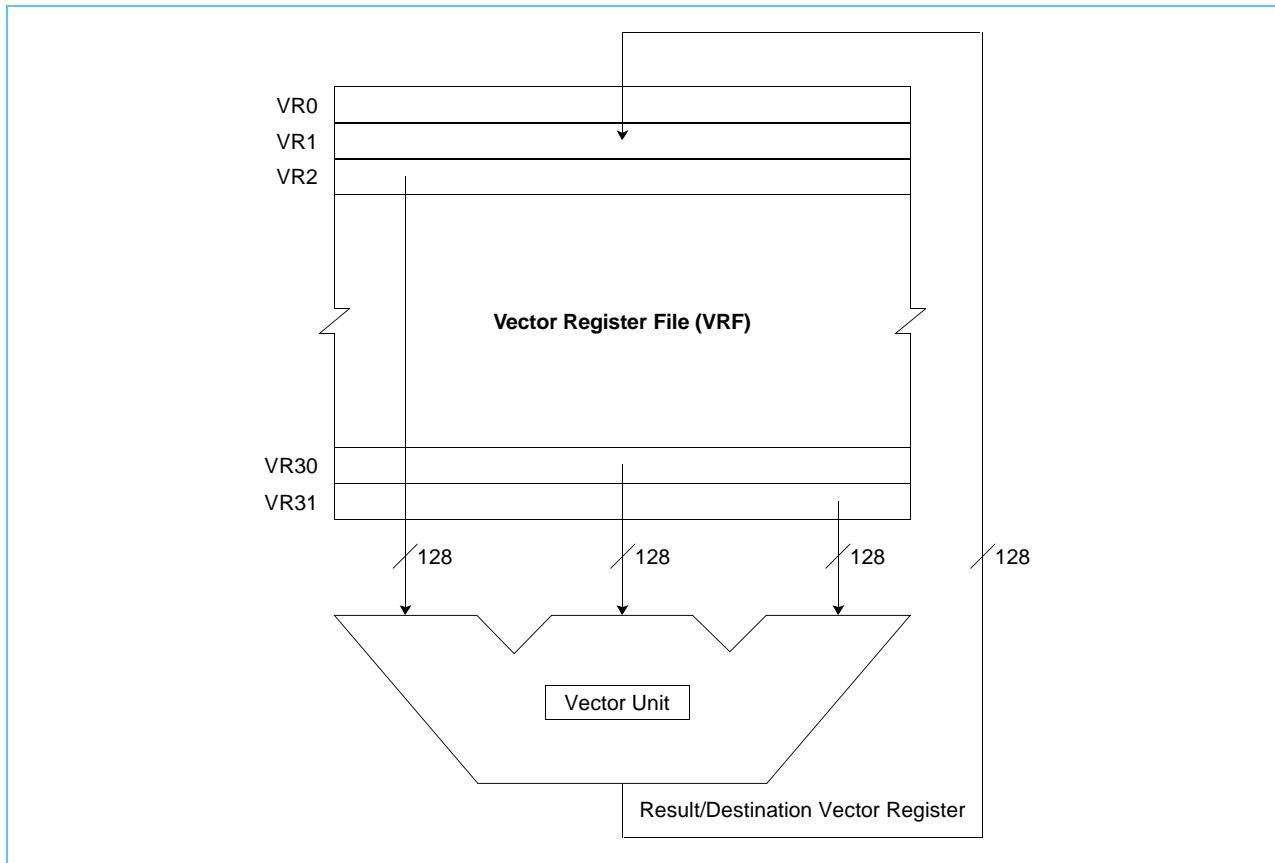
1.2 Vector Processing Architectural Model

This chapter provides overviews of aspects defined by the Vector 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 Vector Registers and Programming Model

In the vector processing 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* on page 28.

Vector/SIMD Multimedia Extension Technology*Figure 1-2. Vector /SIMD Multimedia Extension 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 vector 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 the Vector ISA is PowerPC big-endian. Big-endian byte ordering is shown in *Figure 1-3* on page 29.

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

Quadword															
Word 0				Word 1				Word 2				Word 3			
High Order Halfword 0		Low Order Halfword 1		Halfword 2		Halfword 3		Halfword 4		Halfword 5		Halfword 6		Halfword 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)

- 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 halfword for word 0 (bits [0–15]), would be halfword 0 (bits [0–7]), and the low-order halfword for word 0 would be halfword 1 (bits [8–15]).
- In big-endian mode, a vector quadword load instruction for which the effective address (EA) is quadword 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, a vector quadword store instruction for which the EA is quadword-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

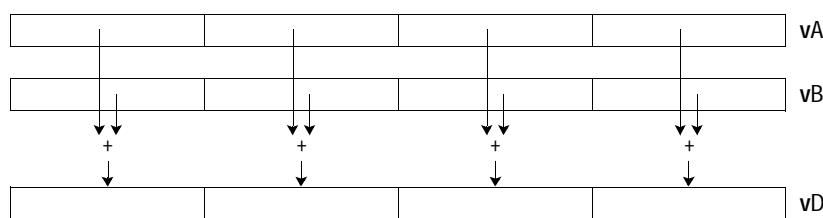
The Vector ISA basically has two modes for floating-point; that is, a Java™/IEEE/C9X-compliant mode or a possibly faster non-Java/non-IEEE mode. The Vector ISA conforms to the Java Language Specification 1 (hereafter referred to as Java), which is a subset of the default environment specified by the 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, the Vector 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), the Vector ISA conforms to C9X when in Java-compliant mode.

Vector/SIMD Multimedia Extension Technology**1.2.3 Vector Addressing Modes**

As with PowerPC instructions, vector 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. Vector load, store, and stream prefetch instructions use secondary opcodes in primary opcode 31 (0b011111). Vector ALU-type instructions use primary opcode point 4 (0b000100).

Vector 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*.

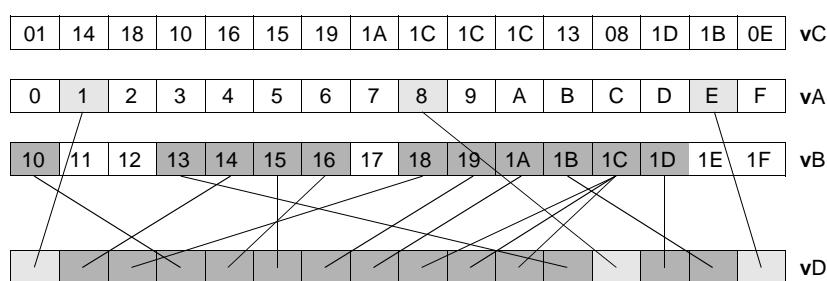
Figure 1-4. Intraelement Example, vaddsws



In this example, the four signed integer (32 bits) elements in register **vA** are added to the corresponding four signed integer (32 bits) elements in register **vB**, and the four results are placed in the corresponding elements in register **vD**.

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 (**vA** and **vB**) to be copied to any byte in the destination vector register, **vD**. The bytes in a third source vector register (**vC**) 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, and so forth) 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 Vector Instruction Set

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

- Vector integer arithmetic instructions—These instructions are defined by the UIISA. 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 UIISA.
 - 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 UIISA.



Vector/SIMD Multimedia Extension Technology

- 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 Vector 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.

1.2.5 Vector Cache Model

The Vector 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 Vector ISA does not define hardware aspects of cache implementations.

1.2.6 Vector Exception Model

The vector unit never generates an exception. Data stream instructions 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 is just canceled and is ignored. Most vector instructions do not generate any kind of exception.

The vector processing unit (VPU) 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 primary functions of the Memory Management Unit (MMU) 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. The Vector ISA does not provide any additional instructions to the PowerPC memory management model, but the vector instructions have options to ensure that an operand is correctly placed in a vector register or in memory.



2. Vector Register Set

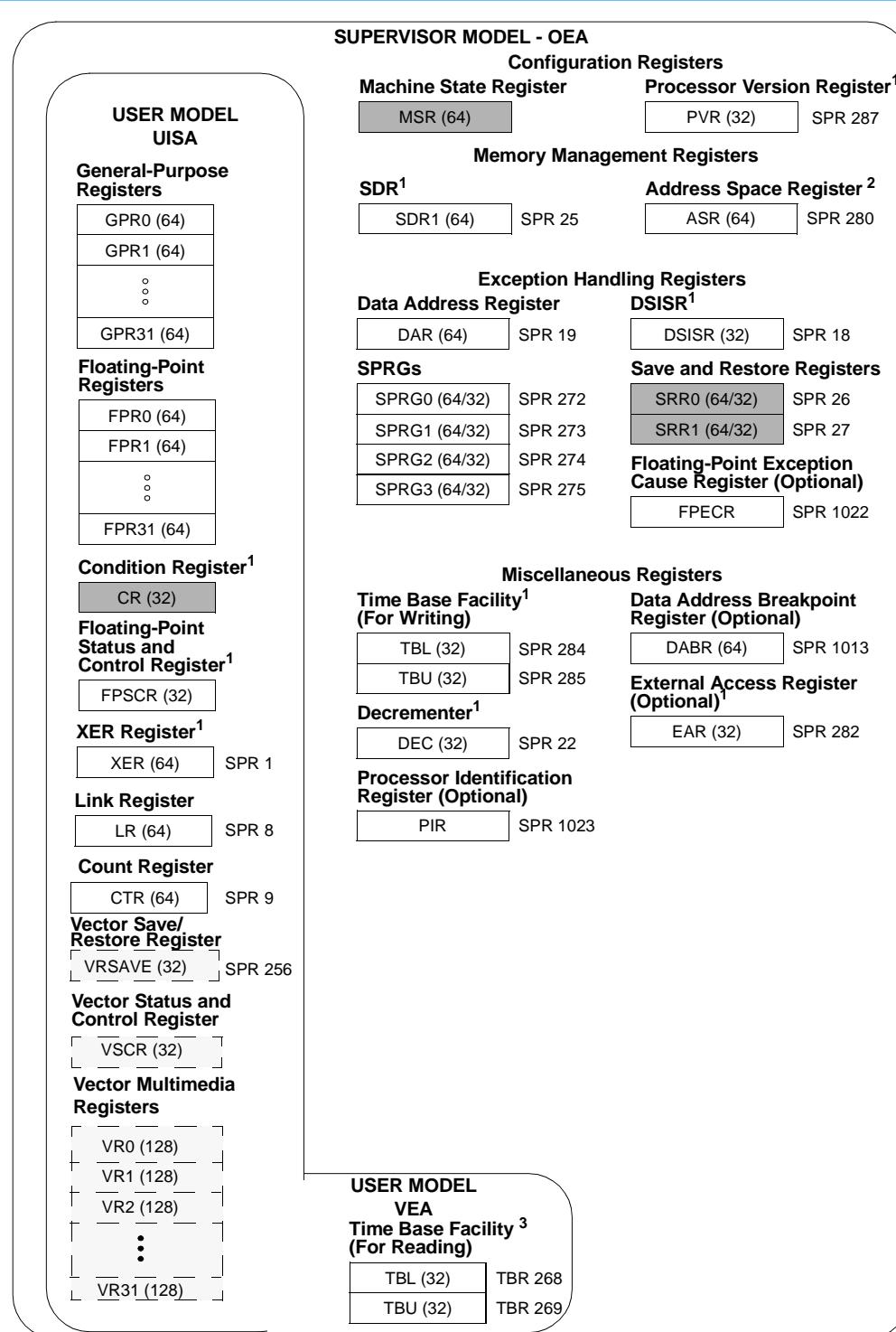
This chapter describes the register organization defined by the vector processing technology. It also describes how vector instructions affect some of the PowerPC registers. The vector instruction set architecture (ISA) defines register-to-register operations for all computational instructions. Source 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 the floating-point registers (FPRs). Data is transferred between memory and vector registers with explicit vector load and store instructions only.

Note: 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 Overview of the Vector and PowerPC Registers

Vector registers can be accessed by user-level and supervisor-level instructions as shown in *Figure 2-1* on page 34. The VRs are accessed as instruction operands. Access to the registers can be either implicit or explicit. The number to the right of the register name indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the Fixed-Point Exception Register (XER) is special purpose register [SPR] 1).

Note: The PowerPC registers affected by vector instructions are shaded.

Vector/SIMD Multimedia Extension Technology
Figure 2-1. Programming Model—All Registers


1 These registers are 32-bit registers only.

2 These registers are on 64-bit implementations only.

3 In 64-bit implementations, TBR268 is read as a 64-bit value.

2.2 Registers Defined by Vector ISA

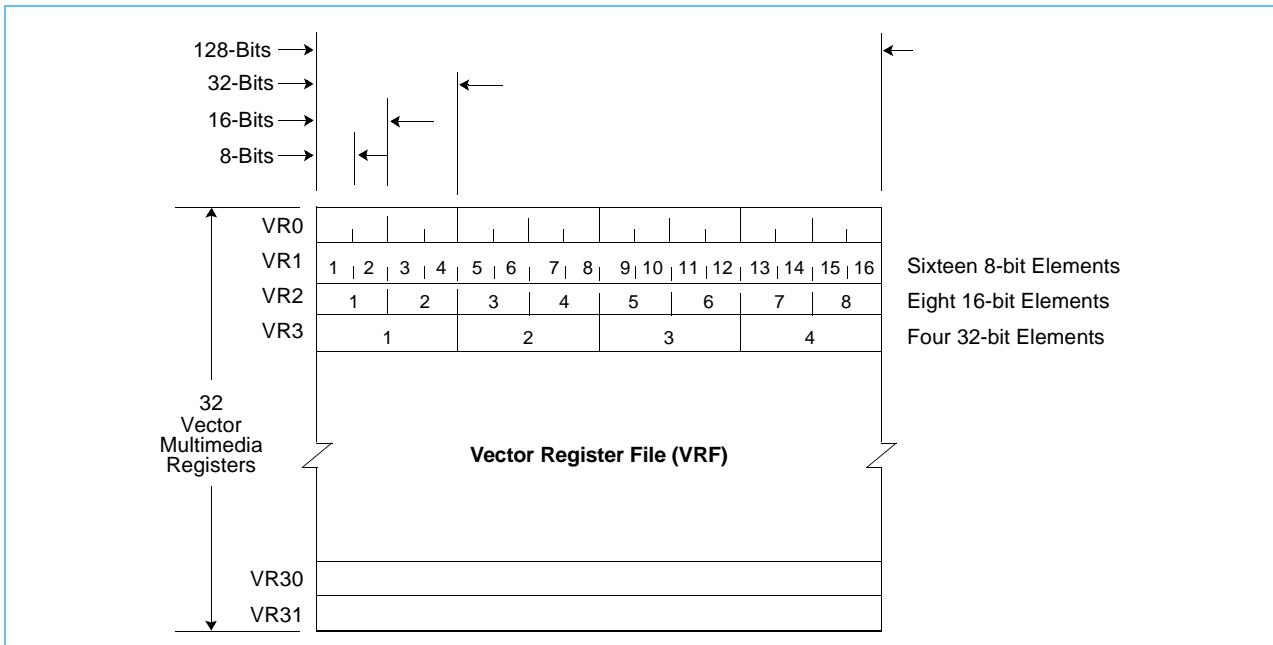
The user-level registers can be accessed by all software with either user or supervisor privileges. The user-level register set for the vector processing technology includes the following registers:

- Vector registers (VRs): the vector register file consists of 32 VRs (VR0-VR31). The VRs serve as vector source and vector destination registers for all vector instructions. See *Section 2.2.1 Vector Register File* for more details.
- Vector Status and Control Register (VSCR): the VSCR contains the non-Java and saturation bit with the remaining bits being reserved. See *Section 2.2.2 Vector Status and Control Register* for more details.
- Vector Save/Restore Register (VRSAVE): the VRSAVE assists the application and operating system software in saving and restoring the architectural state across context-switched events. See *Section 2.2.3 VRSAVE Register (VRSAVE)* for more details.

2.2.1 Vector Register File

The vector register file (VRF), as shown in *Figure 2-2*, 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 2-2. Vector Register File (VSCR)

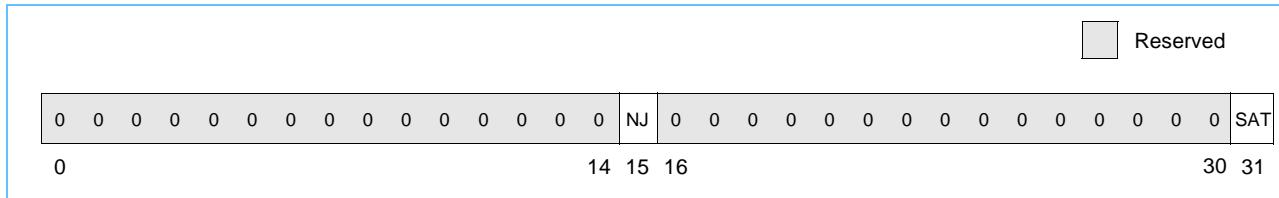


The vector registers are accessed as vector instruction operands. Access to registers is explicit as part of the execution of an instruction.

Vector/SIMD Multimedia Extension Technology**2.2.2 Vector Status and Control Register**

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 Floating-Point Status and Control Register (FPSCR) in the PowerPC scalar floating-point unit. The VSCR is shown in *Figure 2-3*.

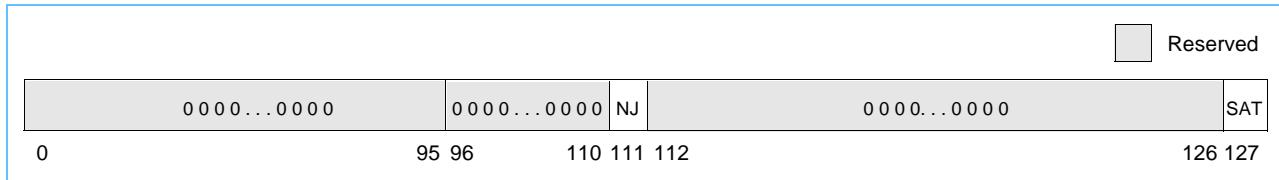
Figure 2-3. Vector Status and Control Register (VSCR)



The VSCR has two defined bits, the non-Java mode (NJ) bit (VSCR[15]) and the 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-aligned in the 128-bit vector register. When moved to a vector register, the upper 96 bits VRx[0–95] of the vector register are cleared, so that the VSCR in a vector register looks as shown in *Figure 2-4*.

Figure 2-4. 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	<p>Non-Java. A mode control bit that determines whether vector floating-point operations will be performed in a Java-IEEE-C9X-compliant mode or a possibly faster non-Java/non-IEEE mode.</p> <p>0 The Java-IEEE-C9X-compliant mode is selected. Denormalized values are handled as specified by Java, IEEE, and C9X standard.</p> <p>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.</p> <p>This mode is described in detail in the floating-point overview <i>Section 3.2.1 Floating-Point Modes</i>.</p>
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	<p>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, see <i>Section 4.2.1.1 Saturation Detection</i>.</p> <p>1 The vector saturate instruction implicitly sets when saturation has occurred on the results one of the vector instructions having saturate in its name:</p> <ul style="list-style-type: none"> Move To VSCR (mtvscr) Vector Add Integer with Saturation (vaddubs, vadduhs, vadduws, vaddsbs, vaddshs, vaddsws) Vector Subtract Integer with Saturation (vsububs, vsubuhs, vsubuws, vsubsbs, vsubshs, vsubsws) Vector Multiply-Add Integer with Saturation (vmhaddshs, vmhraddshs) Vector Multiply-Sum with Saturation (vmsumuhhs, 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) <p>0 Indicates no saturation occurred; mtvscr can explicitly clear this bit.</p>

The **mtvscr** is context synchronizing. This implies that all vector instructions logically preceding an **mtvscr** in the program flow will execute in the architectural context (NJ mode) that existed before 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 **mtvscr** 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. 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 vector instructions. Therefore care must be taken in reading it to avoid performance problems.



Vector/SIMD Multimedia Extension Technology

2.2.3 VRSAVE Register (VRSAVE)

The VRSAVE register, shown in *Figure 2-5*, is a user-level, 32-bit SPR used to assist in application and operating system software in saving and restoring the architectural state across process context-switched events. VRSAVE is SPR256 and is entirely maintained and managed by software.

Figure 2-5. Saving/Restoring the Vector Context Register (VRSAVE)

Field	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
Reset	0000_0000_0000_0000																0000_0000_0000_0000															
R/W	R/W with mfsp r or mtsp r instructions																															
SPR	SPR256																															
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	

VRSAVE bit settings are shown in *Table 2-2*.

Table 2-2. VRSAVE Bit Settings

Bits	Name	Description
0-31	VR n	0 VR n is not being used for the current process. 1 VR n is being used for VR n for the current process.

The VRSAVE register can be accessed only by the **mfsp**r or **mtsp**r instruction.

Note: An application binary interface (ABI) can use VRSAVE to indicate the usage of the vector registers. Any application use of VRSAVE must be in compliance with the ABI to ensure correct program operation.

2.3 Additions to the PowerPC UISA Registers

The PowerPC user instruction set architecture (UISA) registers can be accessed by either user or supervisor-level instructions. The Condition Register (CR) is the only register affected by the vector processing architecture. The CR is a 32-bit register, divided into eight 4-bit fields, CR0-CR7, that reflect the results of certain arithmetic operations and provide a mechanism for testing and branching.

2.3.1 PowerPC Condition Register

The PowerPC CR is a 32-bit register that reflects the result of certain operations and provides a mechanism for testing and branching. For the Vector ISA, the CR6 field can optionally be used. If a vector instruction field's record bit (Rc) is set in a vector compare instruction, then the CR6 field is updated. The bits in the PowerPC CR are grouped into eight 4-bit fields, CR0–CR7, as shown in *Figure 2-6*.

Figure 2-6. Condition Register (CR)

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

For more details on the CR see Chapter 2, “PowerPC Register Set,” in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors*.

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	1 All fields are in bounds for the vcmpbfp instruction; therefore, the result code of all fields is '00'
		0 All fields were in bounds	0 One of the fields is out of bounds for the vcmpbfp instruction
27	3	0	0

The Rc bit should be used sparingly. In some implementations, instructions with Rc bit = '1' could have a 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.

Vector/SIMD Multimedia Extension Technology

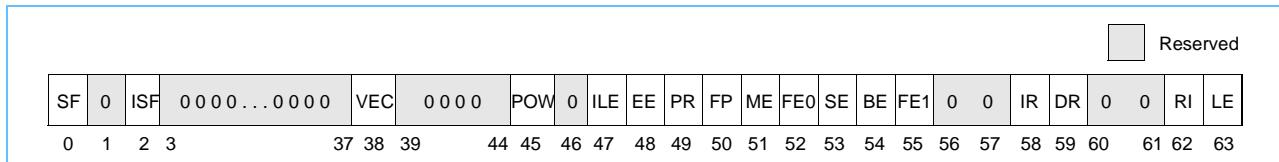
2.4 Additions to the PowerPC OEA Registers

The PowerPC operating environment architecture (OEA) can be accessed only with supervisor-level instructions. Any attempt to access these SPRs with user-level instructions results in a supervisor-level exception. For more details on the Machine State Register (MSR) and the Machine Status Save/Restore Register (SRR), see Chapter 2, “PowerPC Register Set,” in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors*.

2.4.1 VPU Bit in the PowerPC Machine State Register (MSR)

A vector processing unit (VPU) available bit is added to the PowerPC Machine State Register (MSR[VEC]). The MSR is 64 bits wide as shown in *Figure 2-7*.

Figure 2-7. Machine State Register (MSR)



Vector 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.

Note: The Cell Broadband Engine processor treats the data stream instructions as no-ops.

Table 2-4 on page 41 shows the vector bit definitions for the MSR, as well as how the PR and DR bits are affected by the vector data stream instructions.

Table 2-4. MSR Bit Settings Affected by the VPU

Bits	Name	Description
38	VEC	<p>VPU Available 0 When the bit is cleared to zero, the processor executes a “VPU Unavailable Exception” when any attempt is made to execute a vector instruction that accesses the VRF or the VSCR register. 1 The VRF and VSCR registers are accessible to vector instructions.</p> <p>Note: The VRSAVE register is not protected by MSR[VEC].</p> <p>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.</p>
49	PR	<p>Privilege level 0 The processor can execute both user and supervisor-level instructions. 1 The processor can only execute user-level instructions.</p> <p>Note: Care should be taken if data-stream prefetching is used in a 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.</p>
59	DR	<p>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’.</p>

For more detailed information including the other bit settings for MSR, see Chapter 2, “PowerPC Register Set,” in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors*.

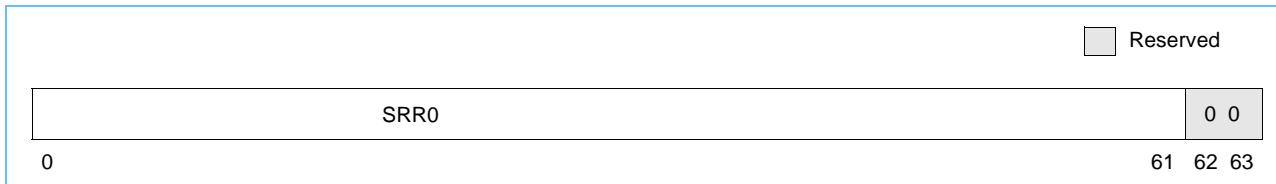
2.4.2 Machine Status Save/Restore Registers (SRR)

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 **rfd** instruction is executed. For more detailed information, see Chapter 2, “PowerPC Register Set,” in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors*.

2.4.2.1 Machine Status Save/Restore Register 0 (SRR0)

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

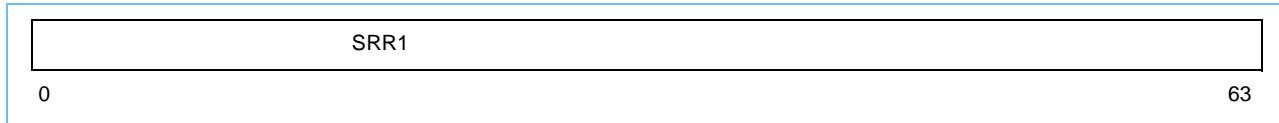
Figure 2-8. Machine Status Save/Restore Register 0 (SRR0)



Vector/SIMD Multimedia Extension Technology**2.4.2.2 Machine Status Save/Restore Register 1 (SRR1)**

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

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



When a VPU 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, reflecting the state of the MSR as it was just before the exception.



3. Operand Conventions

This section describes the operand conventions as they are represented in the vector processing technology at the user instruction set architecture (UISA) level. Detailed descriptions are provided of conventions used for transferring data between vector registers and memory, and for representing data in these vector registers. Additionally, the floating-point default conditions for exceptions are described.

3.1 Data Organization in Memory

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

The following sections describe the concepts of alignment and byte ordering of data for quadwords; otherwise, alignment is the same as described in Chapter 3, “Operand Conventions,” in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*.

3.1.1 Aligned and Misaligned Accesses

Vectors are accessed from memory with instructions such as Load Vector 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. Vector instructions are 4 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
Halfword	2 bytes	xxx0	xx00
Word	4 bytes	xx00	xx00
Quadword	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, a data item is said to be halfword aligned if its address is a multiple of two.

It is important to understand that vector memory operands are assumed to be aligned, and vector 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. Therefore, for the Vector ISA, the low-order bit of the effective address is ignored for halfword vector memory access instructions, and the low-order 4 bits of the effective address are ignored for quadword vector 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.

Vector/SIMD Multimedia Extension Technology

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

3.1.2 Vector Processing Unit Byte Ordering

For processors using the PowerPC and vector/SIMD architecture, the smallest addressable memory unit is the byte (8 bits), and scalars are composed of one or more sequential bytes. The Vector ISA supports big-endian byte ordering.

For 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.

3.1.3 Vector Register and Memory Access Alignment

When loading an aligned byte, halfword, or word memory operand into a vector register, the element that receives the data is the element that would have received the data if the entire aligned quadword containing the memory operand addressed by the effective address had 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 if the entire vector register had been stored to the aligned quadword containing the memory operand addressed by the effective address. (Byte memory operands are always aligned.)

For aligned byte, halfword, 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 instruction will take the contents of an element in a vector register and replicate it 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 3.1.4. Quadword 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 quadword in memory.

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

3.1.4 Quadword Data Alignment

The vector instruction set does not provide for alignment exceptions for loading and storing data. When performing vector loads and stores, with the exception of the Load/Store Vector Left/Right Indexed instructions implemented in the Cell Broadband Engine processor, the effect is as if the low-order 4 bits of the address are 0b0000, regardless of the actual effective address generated. Because vectors can often be misaligned due to the nature of the algorithm, the vector instruction set provides support for postalignment of quadword loads and for prealignment of quadword stores.

Consider the following array, B, consisting of five word elements.

```
int B[5];
B[0] = 0x01234567;
```

```
B[1] = 0x00112233;
B[2] = 0x44556677;
B[3] = 0x8899AABB;
B[4] = 0xCCDDEEFF;
```

Figure 3-1 illustrates the storage image of array B.

Figure 3-1. Misaligned Vector

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
	Most-Significant Quadword (MSQ)																Least-Significant Quadword (LSQ)															
Contents	01	23	45	67	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF												
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															

While this example shows the array starting at a quadword-aligned address, if the data of interest are elements one through four, accessing elements one through four of array B involves an unaligned quadword storage access that spans two aligned quadwords.

In the remainder of this section, the aligned quadword that contains the most-significant bytes of the unaligned quadword is called the most-significant quadword (MSQ) and the aligned quadword that contains the least-significant bytes of the unaligned quadword is called the least-significant quadword (LSQ). Because the Vector Storage Access instructions ignore the low-order bits of the effective address, the unaligned quadword cannot be transferred between storage and a vector register using a single instruction. The remainder of this section gives examples of accessing unaligned quadword storage operands. Similar sequences can be used to access unaligned byte and halfword storage operands.

3.1.4.1 Loading an Unaligned Quadword Using Vector Permute

Loading the contents of elements one through four of array B (see *Figure 3-1*) into VR[Vt] involves an unaligned quadword storage access. One approach that can be used to load an unaligned quadword into a vector register employs the Vector Permute instruction to extract the subject quadword from the two aligned quadwords that contain it. The following sequence of instructions copies the contents of an unaligned quadword in storage into VR[Vt].

# Assumptions	MEM (B, 16):	01	23	45	67	00	11	22	33	44	55	66	77	88	99	AA	BB												
	MEM (B + 16, 16):	CC	DD	EE	FF																								
	GPR (Ra):	00	00	00	00	80	00	00	00																				
	GPR (Rb):	00	00	00	00	00	00	00	00	04																			
lvs1	Vp, Ra, Rb	VR (Vp):	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13											
lvx	Vhi, Ra, Rb	VR (Vhi):	01	23	45	67	00	11	22	33	44	55	66	77	88	99	AA	BB											
addi	Rb, Rb, 16	GPR (Rb):	00	00	00	00	00	00	00	00	14																		
lvx	Vlo, Ra, Rb	VR (Vlo):	CC	DD	EE	FF																							
vperm	Vt, Vhi, Vlo, Vp	VR (Vt):	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF											



Vector/SIMD Multimedia Extension Technology

After the two quadwords have been loaded into VR[Vhi] and VR[Vlo], using Load Vector Indexed instructions, where the address for loading VR[Vlo] is 16 greater than the address for loading VR[Vhi], the alignment is performed by shifting the 32-byte quantity VR[Vhi] || VR[Vlo] to the left by an amount determined by the address of the first byte of the required data. The shifting is done using a Vector Permute instruction for which the permute control vector is generated by a Load Vector for Shift Left instruction. The Load Vector for Shift Left instruction uses the same address specification as the Load Vector Indexed instruction that loads VR[Vhi]; this is the address of the required unaligned quadword.

Note: The Load Vector for Shift Left instruction is used to generate the permute control vector for unaligned loads.

3.1.4.2 Storing an Unaligned Quadword Using Vector Permute

Storing the contents of VR[Vs] to elements one through four of array B (see *Figure 3-1 Misaligned Vector* on page 45) involves an unaligned quadword storage access.

The procedure for storing an unaligned quadword is essentially the reverse of the procedure for loading an unaligned quadword. However, a read-modify-write sequence is required that inserts the source quadword into two aligned quadwords in storage. The quadword to be stored is assumed to be in VR[Vs]. The contents of VR[Vs] are shifted right and split into two parts, each of which is merged (using a Vector Select instruction) with the current contents of the two aligned quadwords (MSQ and LSQ). The MSQ will contain the most significant bytes, and the LSQ will contain the least significant bytes, of the unaligned quadword. The resulting two quadwords are stored using Store Vector Indexed instructions. A Load Vector for Shift Right instruction is used to generate the permute control vector that is used for the shifting. A single register is used for the shifted contents; this is possible because the shifting is done by means of a right rotation. The rotation is accomplished by specifying VR[Vs] for both components of the Vector Permute instruction. In addition, the same permute control vector is used on a sequence of ones and zeros to generate the mask used by the Vector Select instructions that do the merging.

The following sequence of instructions copies the contents of VR[Vs] into an unaligned quadword in storage.



Note: The Load Vector for Shift Right instruction is used to generate the permute control vector for unaligned stores.

3.1.4.3 Loading an Unaligned Quadword Using Load Vector Left/Right

The Cell Broadband Engine processor provides a set of storage access instructions that more efficiently handles unaligned loads. These instructions are Load Vector Left Indexed (**lvlx**), Load Vector Left Indexed Last (**lvlxl**), Load Vector Right Indexed (**rvrx**), and Load Vector Right Indexed Last (**rvrxl**). The following sequence of instructions is used.

# Assumptions	MEM (B, 16):	01	23	45	67	00	11	22	33	44	55	66	77	88	99	AA	BB	
	MEM (B + 16, 16):	CC	DD	EE	FF													
	GPR (Ra):	00	00	00	00	80	00	00	00	(address of B[0])								
	GPR (Rb):	00	00	00	00	00	00	00	04	(index to B[1])								
lvlx	Vhi, Ra, Rb	VR (Vhi):	00	11	22	33	44	55	66	77	88	99	AA	BB	00	00	00	
addi	Rb, Rb, 16	GPR (Rb):	00	00	00	00	00	00	00	14								
lvrx	Vlo, Ra, Rb	VR (Vlo):	00	00	00	00	00	00	00	00	00	00	00	CC	DD	EE	FF	
vor	Vt, Vhi, Vlo	VR (Vt):	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF

Ivlx and **Ivrx** are supported only on the Cell Broadband Engine processor and are not portable to other implementations of the vector/SIMD multimedia extension technology.

Vector/SIMD Multimedia Extension Technology

3.1.4.4 Storing an Unaligned Quadword Using Store Vector Left/Right Indexed

The Cell Broadband Engine processor provides a set of storage access instructions that more efficiently handles unaligned store. These instructions are Store Vector Left Indexed (**stvlx**), Store Vector Left Indexed Last (**stvlxl**), Store Vector Right Indexed (**stvrx**), and Store Vector Right Indexed Last (**stvrxl**). The following sequence of instructions is used.

# Assumptions	MEM (B, 16):	01 23 45 67													
	MEM (B + 16, 16):														
	GPR (Ra):	00 00 00 00	80 00 00 00												(address of B[0])
	GPR (Rb):	00 00 00 00	00 00 00 04												(index to B[1])
	VR (Vs):	00 11 22 33	44 55 66 77	88 99 AA BB CC DD EE FF											
stvlx	Vs, Ra, Rb	MEM (B, 16):	01 23 45 67	00 11 22 33	44 55 66 77	88 99 AA BB CC DD EE FF									
addi	Rb, Rb, 16	GPR (Rb):	00 00 00 00	00 00 00 14											
stvrx	Vs, Ra, Rb	MEM (B + 16, 16)	CC DD EE FF												

Note: **stvlx** and **stvrx** are supported only on the Cell Broadband Engine processor and are not portable to other implementations of the vector/SIMD multimedia extension technology.

3.1.4.5 Scalar Loads and Stores

No alignment is performed for scalar load or store instructions in the Vector 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.

3.1.4.6 Misaligned Scalar Loads and Stores

Although no direct support of misaligned scalars is provided, the load-aligning sequence for the vectors described in *Section 3.1.4.1 Loading an Unaligned Quadword Using Vector Permute* on page 45 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 quadword at that address. Similarly, the read-modify-write sequences, with the mask adjusted for the scalar size, can be used to store misaligned scalars.



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

3.2 Vector Floating-Point Instructions—UISA

There are two kinds of floating-point instructions defined for the PowerPC and Vector 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 for 64-bit Microprocessors* apply to vector floating-point except as follows:

- In general, no status bits are set to reflect the results of floating-point operations. The only exception is that VSCR[SAT] can 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 vector 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 Vector ISA, and cannot be emulated satisfactorily using the functions that are supported by Vector ISA, the functions provided by the floating-point processor should be used; see Chapter 4, “Addressing Modes and Instruction Set Summary,” in *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*.

3.2.1 Floating-Point Modes

Vector 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, and thus eliminating enabling control flags
- Requiring only round-to-nearest rounding mode eliminates directed rounding modes and the associated rounding control flags



Vector/SIMD Multimedia Extension Technology

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 a 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 vector 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 vector floating-point instructions.

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



If an exception occurs, a result is placed into the corresponding target element as described in the following subsections. This result is the default result specified by Java, the IEEE standard, or C9X, as applicable. Recall that denormalized source values are treated as if they were zero when VSCR[NJ] = '1'. 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 nonzero, then the value is a Not a Number (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:

- A vector 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 vector instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is 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[.]**, **vcmpfbfp[.]**), the corresponding result is 0x0000_0000.

Vector/SIMD Multimedia Extension Technology**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 the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*.

3.2.4.3 Zero Divide Exception

A zero divide exception occurs when a Vector Reciprocal Estimate Floating-Point (**vrepf**) 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** $\log_2(-x) \rightarrow \text{QNaN}$, where $x \neq 0$

3.2.4.5 Overflow Exception

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

- For a vector instruction that would normally produce floating-point results, the magnitude of what would have been the result if the exponent range were unbounded exceeds that of the largest finite 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 two unsigned immediate values (UIMMs) is a number too large to be represented in the target integer format.



The following actions are taken:

1. If the vector 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 vector instructions that would normally produce floating-point results. It is detected before rounding. It occurs when a nonzero intermediate result, computed as though both the precision and the exponent range were unbounded, is less in magnitude than the smallest normalized 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 vector 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 vA, then from vB, and then from vC. 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.



Vector/SIMD Multimedia Extension Technology

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 vector 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'.



4. Addressing Modes and Instruction Set Summary

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

- Vector integer arithmetic instructions—These include arithmetic, logical, compare, rotate and shift instructions, and are described in *Section 4.2.1 Vector Integer Instructions*.
- Vector floating-point arithmetic instructions—These include floating-point arithmetic instructions, and are described in *Section 4.2.2 Vector Floating-Point Instructions*, which also discusses floating-point modes.
- Vector load and store instructions—These include load and store instructions for vector registers, and are described in *Section 4.2.3 Vector Load and Store Instructions*.
- Vector permutation and formatting instructions—These include pack, unpack, merge, splat, permute, select and shift instructions, and are described in *Section 4.2.5 Vector Permutation and Formatting Instructions*.
- Processor control instructions—These instructions are used to read and write from the Vector Status and Control Register, and are described in *Section 4.2.6 Processor Control Instructions—UISA*.
- Memory control instructions—These instructions are used to manage caches (user level and supervisor level), and are 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.

Vector integer instructions operate on byte, halfword, and word operands. Floating-point instructions operate on single-precision operands. The Vector ISA uses instructions that are 4 bytes long and word-aligned. It provides for byte, halfword, 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 vector instruction set. Descriptions of memory addressing, synchronization, and the Vector Processing Unit (VPU) exception summary follow.

4.1.1 Execution Model

When used with the PowerPC instructions, vector instructions can be viewed by the programmer as 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 vector implementations may not allow out-of-order execution and completion. Non-data-dependent vector instructions may issue and execute while previously issued instructions with a longer latency are still in the execution stage. Register renaming is useful for vector instructions to avoid stalling dispatch on false dependencies and to allow maximum register name reuse in heavily unrolled loops. The execution of a sequence of instructions will not be interrupted by exceptions because 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 can only be interrupted by a vector load or store instruction; otherwise, vector instructions do not generate any exceptions.

4.1.2 Computation Modes

The Vector 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 the 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.

The Machine State Register bit '0', MSR[SF], is used to choose between 64-bit and 32-bit modes. When MSR[SF] = '0', the processor runs in 32-bit mode; when MSR[SF] = '1', the processor runs in the default 64-bit mode.

- 32-bit implementations, in that all registers except floating-point registers (FPRs) are 32 bits long, and effective addresses are 32 bits long.

Instructions defined in this chapter are provided for 64-bit implementations unless otherwise stated.

4.1.3 Classes of Instructions

Vector instructions follow 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 64-Bit Microprocessors*. For the Vector 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 can be bytes, halfwords, words, or quadwords for vector 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 Vector ISA supports big-endian byte ordering; see *Section 3.1.2 Vector Processing Unit 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-bit 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 for 64-Bit Microprocessors*.

A zero in the rA 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 (rA|0).

In all implementations (including 32-bit mode in 64-bit implementations), the doublewords of a quadword can be swapped.

Vector 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 Vector Load and Store Address Generation*.

4.2 Vector UIISA Instructions

Vector instructions can provide additional supporting instructions to the PowerPC Architecture. This section discusses the instructions defined in the vector UIISA.

4.2.1 Vector Integer Instructions

The following are categories for vector integer instructions:

- Arithmetic
- Compare
- Logical
- Rotate and shift

Integer instructions use the content of the 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 vector 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, halfword, word, quadword) and type of operand (unsigned, signed). The Vector 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 $0+0 \rightarrow 0$ and unsigned byte $1+254 \rightarrow 255$, 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**, **vsubuhs**, **vsubuws**, **vsubsbs**, **vsubshs**, **vsubsws**)
- Vector multiply-add integer with saturation (**vhaddshs**, **vhhraddshs**)
- 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: 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 2.2.2 Vector Status and Control Register*.

4.2.1.2 Vector Integer Arithmetic Instructions

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

Table 4-1. Vector Integer Arithmetic Instructions (Page 1 of 6)

Name	Mnemonic	Syntax	Operation
Vector Add Signed Integer Saturate	vaddsb vaddsh vaddsw	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vaddsb, element length = 8 bits (1 byte). For vaddsh, element length = 16 bits (1 halfword). For vaddsw, element length = 32 bits (1 word). <p>Each n-bit signed integer element in register vA is added to the corresponding n-bit signed integer element in register vB. If the intermediate result is greater than $2^{n-1}-1$, it saturates to $2^{n-1}-1$. If the intermediate result is less than -2^{n-1}, it saturates to -2^{n-1}. If saturation occurs, the SAT bit is set. The signed integer result is placed into the corresponding n-bit element of register vD.</p>
Vector Add Unsigned Integer Modulo	vaddubm vadduhm vadduwm	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vaddubm, element length = 8 bits (1 byte). For vadduhm, element length = 16 bits (1 halfword). For vadduwm, element length = 32 bits (1 word). <p>Each n-bit unsigned integer element in register vA is added to the corresponding n-bit unsigned integer element in register vB. The unsigned integer result is placed into the corresponding n-bit element of register vD.</p> <p>Note that these instructions can be used for unsigned or signed integers.</p>
Vector Add Unsigned Integer Saturate	vaddubs vadduhs vadduws	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vaddubs, element length = 8 bits (1 byte). For vadduhs, element length = 16 bits (1 halfword). For vadduws, element length = 32 bits (1 word). <p>Each n-bit unsigned integer element in register vA is added to the corresponding n-bit unsigned integer element in register vB. If the intermediate result is greater than 2^n-1, it saturates to 2^n-1. If saturation occurs, the SAT bit is set. The unsigned integer result is placed into the corresponding n-bit element of register vD.</p>
Vector Add and Write Carry-out Unsigned Word	vaddcuw	vD,vA,vB	<p>Each unsigned-integer word element in register vA is added to the corresponding unsigned-integer word element in register vB. 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 register vD.</p>
Vector Subtract Signed Integer Saturate	vsubsb vsubsh vsubsw	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vsubsb, element length = 8 bits (1 byte). For vsubsh, element length = 16 bits (1 halfword). For vsubsw, element length = 32 bits (1 word). <p>Each n-bit signed-integer element in register vB is subtracted from the corresponding n-bit signed-integer element in register vA. If the intermediate result is greater than $2^{n-1}-1$, it saturates to $2^{n-1}-1$. If the intermediate result is less than -2^{n-1}, it saturates to -2^{n-1}. If saturation occurs, the SAT bit is set. The signed-integer result is placed into the corresponding n-bit element in register vD.</p>

Vector/SIMD Multimedia Extension Technology**Table 4-1. Vector Integer Arithmetic Instructions (Page 2 of 6)**

Name	Mnemonic	Syntax	Operation
Vector Subtract Unsigned Integer Modulo	vsububm vsubuhm vsubuwm	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vsububm, element length = 8 bits (1 byte). For vsubuhm, element length = 16 bits (1 halfword). For vsubuwm, element length = 32 bits (1 word). <p>Each n-bit integer element in register vB is subtracted from the corresponding n-bit integer element in register vA. The integer result is placed into the corresponding n-bit element in register vD.</p> <p>Note that these instructions can be used for unsigned or signed integers.</p>
Vector Subtract Unsigned Integer Saturate	vsububs vsubuhs vsubuws	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vsububs, element length = 8 bits (1 byte). For vsubuhs, element length = 16 bits (1 halfword). For vsubuws, element length = 32 bits (1 word). <p>Each unsigned-integer element in register vB is subtracted from the corresponding unsigned-integer element in register vA. If the intermediate result is less than '0', it saturates to '0'. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding element in register vD.</p>
Vector Subtract and Write Carry-out Unsigned Word	vsubcuw	vD,vA,vB	<p>Each unsigned-integer word element in register vB is subtracted from the corresponding unsigned-integer word element in register vA. 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 register vD.</p>
Vector Multiply Odd Unsigned Integer	vmuloub vmulouh	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vmuloub, element length = 8 bits (1 byte), and For vmulouh, element length = 16 bits (1 halfword), and <p>Each odd-numbered n-bit unsigned-integer element in register vA is multiplied by the corresponding odd-numbered n-bit unsigned-integer element in register vB. The (2xn)-bit unsigned-integer products are placed, in the same order, into register vD.</p>
Vector Multiply Odd Signed Integer Modulo	vmulosb vmulosh	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vmulosb, element length = 8 bits (1 byte), and For vmulosh, element length = 16 bits (1 halfword), and <p>Each odd-numbered n-bit signed-integer element in register vA is multiplied by the corresponding odd-numbered n-bit signed-integer element in register vB. The (2xn)-bit signed-integer products are placed, in the same order, into register vD.</p>
Vector Multiply Even Unsigned Integer Modulo	vmuleub vmuleuh	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vmuleub, element length = 8 bits (1 byte), and For vmuleuh, element length = 16 bits (1 halfword), and <p>Each even-numbered n-bit unsigned-integer element in register vA is multiplied by the corresponding even-numbered n-bit unsigned-integer element in register vB. The (2xn)-bit unsigned-integer products are placed, in the same order, into register vD.</p>
Vector Multiply Even Signed Integer Modulo	vmulesb vmulesh	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vmulesb, element length = 8 bits (1 byte), and For vmulesh, element length = 16 bits (1 halfword), and <p>Each even-numbered n-bit signed-integer element in register vA is multiplied by the corresponding even-numbered n-bit signed-integer element in register vB. The (2xn)-bit signed-integer products are placed, in the same order, into register vD.</p>

Table 4-1. Vector Integer Arithmetic Instructions (Page 3 of 6)

Name	Mnemonic	Syntax	Operation
Vector Multiply-High and Add Signed Halfword Saturate	vmhaddshs	vD,vA,vB, vC	Each signed-integer halfword element in register vA is multiplied by the corresponding signed-integer halfword element in register vB, producing a 32-bit signed-integer product. The corresponding signed-integer halfword element in register vC is sign-extended to 17 bits and added to bits 0:16 of the product. If the intermediate result is greater than $2^{15}-1$, it saturates to $2^{15}-1$. If the intermediate result 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 halfword element of register vD.
Vector Multiply-High Round and Add Signed Halfword Saturate	vmhraddshs	vD,vA,vB,vC	Each signed-integer halfword element in register vA is multiplied by the corresponding signed-integer halfword element in register vB, producing a 32-bit signed-integer product. The product is rounded by adding the value 0x0000_4000. The corresponding signed-integer halfword element in register vC is sign-extended to 17 bits and added to bits 0:16 of the rounded product. If the intermediate result is greater than $2^{15}-1$, it saturates to $2^{15}-1$. If the intermediate result 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 halfword element of register vD.
Vector Multiply-Low and Add Unsigned Halfword Modulo	vmladduhm	vD,vA,vB,vC	Each integer halfword element in register vA is multiplied by the corresponding integer halfword element in register vB, producing a 32-bit integer product. The product is added to the corresponding integer halfword element in register vC. The integer result is placed into the corresponding halfword element of register vD. Note that vmladduhm can be used for unsigned or signed integers.
Vector Multiply-Sum Unsigned Integer Byte Modulo	vmsumubm	vD,vA,vB,vC	For each word element in register vC, the following operations are performed in the order shown: <ul style="list-style-type: none"> • Each of the four unsigned-integer byte elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer byte element in register vB, 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 register vC. • The unsigned-integer result is placed into the corresponding word element of register vD.
Vector Multiply-Sum Unsigned Integer Halfword Modulo	vmsumuhm	vD,vA,vB,vC	For each word element in register vC, the following operations are performed in the order shown: <ul style="list-style-type: none"> • Each of the two unsigned-integer halfword elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer halfword element in register vB, producing an unsigned-integer 32-bit product. • The unsigned-integer sum of these two products is added to the unsigned-integer word element in register vC. • The unsigned-integer result is placed into the corresponding word element of register vD.

Vector/SIMD Multimedia Extension Technology**Table 4-1. Vector Integer Arithmetic Instructions (Page 4 of 6)**

Name	Mnemonic	Syntax	Operation
Vector Multiply-Sum Signed Half-Word Saturate	vmsumshs	vD,vA,vB,vC	For each word element in register vC , the following operations are performed in the order shown: <ul style="list-style-type: none">• Each of the two signed-integer halfword elements in the corresponding word element of register vA is multiplied by the corresponding signed-integer halfword element in register vB, producing a signed-integer 32-bit product.• The signed-integer sum of these two products is added to the signed-integer word element in register vC.• If the intermediate result is greater than $2^{31}-1$, it saturates to $2^{31}-1$. If the intermediate result 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 word element of register vD.
Vector Multiply-Sum Unsigned Halfword Saturate	vmsumuhs	vD,vA,vB,vC	For each word element in register vC , the following operations are performed in the order shown: <ul style="list-style-type: none">• Each of the two unsigned-integer halfword elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer halfword element in register vB, producing an unsigned-integer 32-bit product.• The sum of the two 32-bit unsigned-integer products is added to the unsigned-integer word element in register vC.• If the intermediate result is greater than 2^{32}, it saturates to 2^{32}. If the intermediate result is less than '0', it saturates to '0'. If saturation occurs, the SAT bit is set.• The unsigned-integer result is placed into the corresponding word element of register vD.
Vector Multiply-Sum Mixed Byte Modulo	vmsummbm	vD,vA,vB,vC	For each word element in register vC , the following operations are performed in the order shown: <ul style="list-style-type: none">• Each of the four signed-integer byte elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer byte element in register vB, 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 register vC.• The signed-integer result is placed into the corresponding word element of register vD.
Vector Multiply-Sum Signed Halfword Modulo	vmsumshm	vD,vA,vB,vC	For each word element in register vC , the following operations are performed in the order shown: <ul style="list-style-type: none">• Each of the two signed-integer halfword elements contained in the corresponding word element of register vA is multiplied by the corresponding signed-integer halfword element in register vB, 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 register vC.• The signed-integer result is placed into the corresponding word element of register vD.
Vector Sum Across Signed Word Saturate	vsumsws	vD,vA,vB	The signed-integer sum of the four signed-integer word elements in register vA is added to the signed-integer word element in bits [96-127] of register vB . If the intermediate result is greater than $2^{31}-1$, it saturates to $2^{31}-1$. If the intermediate result 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 bits [96-127] of register vD . Bits [0-95] of register vD are set to '0'.

Vector/SIMD Multimedia Extension Technology

Table 4-1. Vector Integer Arithmetic Instructions (Page 5 of 6)

Name	Mnemonic	Syntax	Operation
Vector Sum Across Partial (1/2) Signed Word Saturate	vsum2sws	vD,vA,vB	<p>The signed-integer sum of the first two signed-integer word elements in register vA is added to the signed-integer word element in bits [32-63] of register vB. If the intermediate result is greater than $2^{31}-1$, it saturates to $2^{31}-1$. If the intermediate result 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 bits [32-63] of register vD.</p> <p>The signed-integer sum of the last two signed-integer word elements in register vA is added to the signed-integer word element in bits [96-127] of register vB. If the intermediate result is greater than $2^{31}-1$, it saturates to $2^{31}-1$. If the intermediate result 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 bits [96-127] of register vD. Bits[0-31] and bits [64-95] of register vD are set to '0'.</p>
Vector Sum Across Partial (1/4) Unsigned Integer Byte Saturate	vsum4ubs	vD,vA,vB	<p>For each word element in register vB, the following operations are performed in the order shown:</p> <ul style="list-style-type: none"> The unsigned-integer sum of the four unsigned-integer byte elements contained in the corresponding word element of register vA is added to the unsigned-integer word element in register vB. If the intermediate result is greater than $2^{32}-1$, it saturates to $2^{32}-1$. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding word element of register vD.
Vector Sum Across Partial (1/4) Signed Integer Byte Saturate	vsum4sbs	vD,vA,vB	<p>For each word element in register vB, the following operations are performed in the order shown:</p> <ul style="list-style-type: none"> The signed-integer sum of the four signed-integer byte elements contained in the corresponding word element of register vA is added to the signed-integer word element in register vB. If the intermediate result is greater than $2^{31}-1$, it saturates to $2^{31}-1$. If the intermediate result 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 word element of register vD.
Vector Sum Across Partial (1/4) Signed Integer Halfword Saturate	vsum4shs	vD,vA,vB	<p>For each word element in register vB, the following operations are performed, in the order shown:</p> <ul style="list-style-type: none"> The signed-integer sum of the two signed-integer halfword elements contained in the corresponding word element of register vA is added to the signed-integer word element in register vB. If the intermediate result is greater than $2^{31}-1$, it saturates to $2^{31}-1$. If the intermediate result 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 word element of register vD.
Vector Average Unsigned Integer	vavgub vavguh vavguw	vD,vA,vB	<p>Let n = element length. Each n-bit unsigned-integer element in register vA is added to the corresponding n-bit unsigned-integer element in register vB, producing an (n+1)-bit unsigned-integer sum. The sum is incremented by '1'. The high-order n bits of the result are placed into the corresponding n-bit element in register vD.</p> <p>For vavgub, element length = 8 bits (1 byte).</p> <p>For vavguh, element length = 16 bits (1 halfword).</p> <p>For vavguw, element length = 32 bits (1 word).</p>

Vector/SIMD Multimedia Extension Technology**Table 4-1. Vector Integer Arithmetic Instructions (Page 6 of 6)**

Name	Mnemonic	Syntax	Operation
Vector Average Signed Integer	vavgsb vavgsh vavgsw	vD,vA,vB	Let n = element length. Each n-bit signed-integer element in register vA is added to the corresponding n-bit signed-integer element in register vB , producing an (n+1)-bit signed-integer sum. The sum is incremented by '1'. The high-order n bits of the result are placed into the corresponding n-bit element in register vD . For vavgsb , element length = 8 bits (1 byte). For vavgsh , element length = 16 bits (1 halfword). For vavgsw , element length = 32 bits (1 word).
Vector Maximum Unsigned Integer	vmaxub vmaxuh vmaxuw	vD,vA,vB	Let n = element length. Each n-bit unsigned-integer element in register vA is compared to the corresponding n-bit unsigned-integer element in register vB . The larger of the two unsigned-integer values is placed into the corresponding n-bit element in register vD . For vmaxub , element length = 8 bits (1 byte). For vmaxuh , element length = 16 bits (1 halfword). For vmaxuw , element length = 32 bits (1 word).
Vector Maximum Signed Integer	vmaxsb vmaxsh vmaxsw	vD,vA,vB	Let n = element length. Each n-bit signed-integer element in register vA is compared to the corresponding n-bit signed-integer element in register vB . The larger of the two signed-integer values is placed into the corresponding n-bit element in register vD . For vmaxsb , element length = 8 bits (1 byte). For vmaxsh , element length = 16 bits (1 halfword). For vmaxsw , element length = 32 bits (1 word).
Vector Minimum Unsigned Integer	vminub vminuh vminuw	vD,vA,vB	Let n = element length. Each n-bit unsigned-integer element in register vA is compared to the corresponding n-bit unsigned-integer element in register vB . The smaller of the two unsigned-integer values is placed into the corresponding n-bit element in register vD . For vminub , element length = 8 bits (1 byte). For vminuh , element length = 16 bits (1 halfword). For vminuw , element length = 32 bits (1 word).
Vector Minimum Signed Integer	vminsbt vminsh vminsw	vD,vA,vB	Let n = element length. Each n-bit signed-integer element in register vA is compared to the corresponding n-bit signed-integer element in register vB . The smaller of the two signed-integer values is placed into the corresponding n-bit element in register vD . For vminsbt , element length = 8 bits (1 byte). For vminsh , element length = 16 bits (1 halfword). For vminsw , element length = 32 bits (1 word).

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 **vA** with the contents of the elements in **vB**. Each compare result vector consists of TRUE (0xFF, 0xFFFF, 0xFFFFFFFF) or FALSE (0x00, 0x0000, 0x00000000) elements of the size specified by the compare source operand element (byte, halfword, 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 or inverting result vectors.



Vector/SIMD Multimedia Extension Technology

If the record bit (Rc) is set in the integer compare instructions (shown in *Table 4-2*) 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, vD is set to all ones)
25	1	0
26	2	1 Relation is false for all element pairs (that is, register vD is cleared)
27	3	0

Table 4-3 describes 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[.]	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none">For vcmpgtub[.], element length = 8 bits (1 byte).For vcmpgtuh[.], element length = 16 bits (1 halfword).For vcmpgtuw[.], element length = 32 bits (1 word). <p>Each n-bit unsigned-integer byte element in register vA is compared to the corresponding n-bit unsigned-integer byte element in register vB. The corresponding n-bit element in register vD is set to all '1's if the element in register vA is greater than the element in register vB, and is set to all '0's otherwise.</p> <p>If Rc=1, CR Field 6 is set.</p> <p>CR6 = all_greater_than 0b0 none_greater_than 0b0.</p>
Vector Compare Greater Than Signed Integer	vcmpgtsb[.] vcmpgtsh[.] vcmpgtsw[.]	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none">For vcmpgtsb[.], element length = 8 bits (1 byte).For vcmpgtsh[.], element length = 16 bits (1 halfword).For vcmpgtsw[.], element length = 32 bits (1 word). <p>Each n-bit signed-integer byte element in register vA is compared to the corresponding n-bit signed-integer byte element in register vB. The corresponding n-bit element in register vD is set to all '1's if the element in register vA is greater than the element in register vB, and is set to all '0's otherwise.</p> <p>If Rc=1, CR Field 6 is set.</p> <p>CR6 = all_greater_than 0b0 none_greater_than 0b0.</p>
Vector Compare Equal To Unsigned Integer	vcmpequb[.] vcmpequh[.] vcmpequw[.]	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none">For vcmpequb[.], element length = 8 bits (1 byte).For vcmpequh[.], element length = 16 bits (1 halfword).For vcmpequw[.], element length = 32 bits (1 word). <p>Each n-bit unsigned-integer byte element in register vA is compared to the corresponding n-bit unsigned-integer byte element in register vB. The corresponding n-bit element in register vD is set to all '1's if the element in register vA is equal to the element in register vB, and is set to all '0's otherwise.</p> <p>If Rc=1, CR Field 6 is set.</p> <p>CR6 = all_equal_to 0b0 none_equal_to 0b0.</p>

Vector/SIMD Multimedia Extension Technology**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	vD,vA,vB	AND the contents of register vA with the contents register vB and place the result into register vD .
Vector Logical OR	vor	vD,vA,vB	OR the contents of register vA with the contents of register vB and place the result into register vD .
Vector Logical XOR	vxor	vD,vA,vB	XOR the contents of register vA with the contents of register vB and place the result into register vD .
Vector Logical AND with Complement	vandc	vD,vA,vB	AND the contents of register vA with the complement of the contents of register vB and place the result into register vD .
Vector Logical NOR	vnor	vD,vA,vB	NOR the contents of register vA with the contents of register vB and place the result into register vD .

4.2.1.5 Vector Integer Rotate and Shift Instructions

Table 4-5 describes the vector integer rotate instructions.

Table 4-5. Vector Integer Rotate Instructions

Name	Mnemonic	Syntax	Operation
Vector Rotate Left Integer	vrlb vrlh vrlw	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> • For vrlb, element length = 8 bits (1 byte). • For vrlh, element length = 16 bits (1 halfword). • For vrlw, element length = 32 bits (1 word). <p>Each n-bit element in register vA is rotated left by the number of bits specified in the low-order $\log_2(n)$ bits of the corresponding n-bit element in register vB. The result is placed into the corresponding n-bit element in register vD.</p>

Table 4-6 describes the vector integer shift instructions.

Table 4-6. Vector Integer Shift Instructions

Name	Mnemonic	Syntax	Operation
Vector Shift Left Integer	vslb vslh vslw	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vslb, element length = 8 bits (1 byte). For vslh, element length = 16 bits (1 halfword). For vslw, element length = 32 bits (1 word). <p>Each n-bit element in register vA is shifted left by the number of bits specified in the low-order $\log_2(n)$ bits of the corresponding n-bit element in register vB. Bits shifted out of bit 0 of the byte element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding n-bit element in register vD.</p>
Vector Shift Right Integer	vsrb vsrh vsrw	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vsrb, element length = 8 bits (1 byte). For vsrh, element length = 16 bits (1 halfword). For vsrw, element length = 32 bits (1 word). <p>Each n-bit element in register vA is shifted right by the number of bits specified in the low-order $\log_2(n)$ bits of the corresponding n-bit element in register vB. Bits shifted out of bit n-1 of the n-bit element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding n-bit element in register vD.</p>
Vector Shift Right Algebraic Integer	vsrab vsrah vsraw	vD,vA,vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vsrab, element length = 8 bits (1 byte). For vsrah, element length = 16 bits (1 halfword). For vsraw, element length = 32 bits (1 word). <p>Each n-bit element in register vA is shifted right by the number of bits specified in the low-order $\log_2(n)$ bits of the corresponding n-bit element in register vB. Bits shifted out of bit n-1 of the n-bit element are lost. Bit 0 of the n-bit element is replicated to fill the vacated bits on the left. The result is placed into the corresponding n-bit element in register vD.</p>

4.2.2 Vector Floating-Point Instructions

This section describes the vector floating-point instructions, which include the following types of instructions:

- Arithmetic
- Rounding and conversion
- Compare
- Floating-point estimate

The vector 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 Vector 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*.

Vector/SIMD Multimedia Extension Technology**4.2.2.1 Floating-Point Division and Square-Root**

Vector instructions do not have division or square-root instructions. Vector 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, just 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 Vector 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.

Floating-Point Division

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

$$y_1 = y_0 + y_0 * (1 - B * y_0), \text{ where } y_0 = \text{recip_est}(B)$$

This is implemented in the Vector ISA as follows:

$$y_0 = \text{vrefp}(B)$$

$$t = \text{vnmsubfp}(y_0, B, 1)$$

$$y_1 = \text{vmaddfp}(y_0, t, y_0)$$

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 (y_2) 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 $A * y_2$). The correctly rounded quotient can then be obtained.

$$Q' = Q + R * y_2$$

The additional accuracy provided by the fused nature of the vector 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 (unit in the last place), which is a requirement of this method.

Floating-Point Square-Root

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

$$y_1 = y_0 + 0.5 * y_0 * (1 - B * y_0 * y_0), \quad \text{where } y_0 = \text{recip_sqrt_est}(B)$$

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

Table 4-7 describes the floating-point arithmetic instructions.

Table 4-7. Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax	Operation
Vector Add Floating-Point	vaddfp	vD,vA,vB	Each single-precision floating-point element in register vA is added to the corresponding single-precision floating-point element in register vB. Each intermediate result is rounded and placed in the corresponding single-precision floating-point element in register vD.
Vector Subtract Floating-Point	vsubfp	vD,vA,vB	Each single-precision floating-point word element in register vB is subtracted from the corresponding single-precision floating-point word element in register vA. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of register vD.
Vector Maximum Floating-Point	vmaxfp	vD,vA,vB	Each single-precision floating-point word element in register vA is compared to the corresponding single-precision floating-point word element in register vB. The larger of the two single-precision floating-point values is placed into the corresponding word element in register vD. vmaxfp is sensitive to the sign of 0.0. When both operands are ± 0.0 : <ul style="list-style-type: none">• $\max(+0.0, \pm 0.0) = \max(\pm 0.0, +0.0) \Rightarrow +0.0$• $\max(-0.0, -0.0) \Rightarrow -0.0$• $\max(\text{NaN}, x) \Rightarrow \text{QNaN}$ where $x = \text{any value}$
Vector Minimum Floating-Point	vminfp	vD,vA,vB	Each single-precision floating-point word element in register vA is compared to the corresponding single-precision floating-point word element in register vB. The smaller of the two single-precision floating-point values is placed into the corresponding word element in register vD. vminfp is sensitive to the sign of 0.0. When both operands are ± 0.0 : <ul style="list-style-type: none">• $\min(-0.0, \pm 0.0) = \min(\pm 0.0, -0.0) \Rightarrow -0.0$• $\min(+0.0, +0.0) \Rightarrow +0.0$• $\min(\text{NaN}, x) \Rightarrow \text{QNaN}$ where $x = \text{any value}$

Vector/SIMD Multimedia Extension Technology**4.2.2.3 Vector Floating-Point Multiply-Add Instructions**

Vector multiply-add instructions are critically important to performance because a multiply followed by a data dependent addition is the most common idiom in Digital Signal Processor (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.

Vector 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 cannot 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 Vector ISA, use multiply-add instructions with a zero addend (for example, **vmaddfp vD,vA,vC,vB** where **(vB = 0.0)**).

Note: 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 ensure that the sign of a zero result is correct when the product is either +0.0 or -0.0 ($+0.0 + -0.0 \Rightarrow +0.0$, and $-0.0 + -0.0 \Rightarrow -0.0$). When the sign of a resulting 0.0 is not important, then use +0.0 as the addend. This may, in some cases, avoid the need for a second register to hold a -0.0 in addition to the integer 0/floating-point +0.0 that may already be available.

Table 4-8 describes the floating-point multiply-add instructions.

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

Name	Mnemonic	Syntax	Operation
Vector Multiply- Add Floating-Point	vmaddfp	vD,vA,vC,vB	Each single-precision floating-point word element in register vA is multiplied by the corresponding single-precision floating-point word element in register vC . The corresponding single-precision floating-point word element in register vB is added to the product. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element in register vD .
Vector Negative Multiply- Subtract Floating-Point	vnmsubfp	vD,vA,vC,vB	Each single-precision floating-point word element in register vA is multiplied by the corresponding single-precision floating-point word element in register vC . The corresponding single-precision floating-point word element in register vB 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 register vD . Note: Only one rounding occurs in this operation. Also note that a QNaN result is not negated.

4.2.2.4 Vector Floating-Point Rounding and Conversion Instructions

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

The Vector ISA provides separate instructions for converting floating-point numbers to integral floating-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). *Table 4-9* describes the floating-point rounding instructions.

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

Name	Mnemonic	Syntax	Operation
Vector Round to Floating-Point Integer Nearest	vrfin	vD,vB	Each single-precision floating-point word element in register vB is rounded to a single-precision floating-point integer, using the rounding mode Round to Nearest, and placed into the corresponding word element of register vD .
Vector Round to Floating-Point Integer toward Zero	vrfiz	vD,vB	Each single-precision floating-point word element in register vB is rounded to a single-precision floating-point integer, using the rounding mode Round toward Zero, and placed into the corresponding word element of register vD .
Vector Round to Floating-Point Integer toward Plus Infinity	vrfip	vD,vB	Each single-precision floating-point word element in register vB is rounded to a single-precision floating-point integer, using the rounding mode Round toward +Infinity, and placed into the corresponding word element of register vD .
Vector Round to Floating-Point Integer toward Minus Infinity	vrfim	vD,vB	Each single-precision floating-point word element in register vB is rounded to a single-precision floating-point integer, using the rounding mode Round toward -Infinity, and placed into the corresponding word element of register vD .
Vector Convert from Unsigned Fixed-Point Word	vcfux	vD,vB, UIIMM	Each unsigned fixed-point integer word element in register vB is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIIMM} (UIIMM = unsigned immediate value) and placed into the corresponding word element in register vD .
Vector Convert from Signed Fixed-Point Word	vcfsx	vD,vB, UIIMM	Each signed fixed-point integer word element in register vB is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIIMM} and placed into the corresponding word element in register vD .
Vector Convert to Unsigned Fixed-Point Word Saturate	vctuxs	vD,vB, UIIMM	Each single-precision floating-point word element in register vB is multiplied by 2^{UIIMM} . 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 $2^{32}-1$. If the intermediate result is less than '0', it saturates to '0'. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding word element in register vD .
Vector Convert to Signed Fixed-Point Word Saturate	vctsxs	vD,vB, UIIMM	Each single-precision floating-point word element in register vB is multiplied by 2^{UIIMM} . The product is converted to a signed fixed-point 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 the intermediate result 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 word element in register vD .

Vector/SIMD Multimedia Extension Technology**4.2.2.5 Vector Floating-Point Compare Instructions**

This section describes floating-point unordered compare instructions.

All vector 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(vX)` test. No explicit unordered compare instructions or traps are provided. However, the greater-than-or-equal-to predicate (\geq) (**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 vector/SIMD code.

Table 4-10. Common Mathematical Predicates

Case	Mathematical Predicate	Vector Realization	Relations			
			a>b	a<b	a=b	?
1	$a = b$	$a = b$	F	F	T	F
2	$a \neq b$ (?>)	$\neg(a = b)$	T	T	F	T
3	$a > b$	$a > b$	T	F	F	F
4	$a < b$	$b > a$	F	T	F	F
5	$a \geq b$	$\neg(b > a)$	T	F	T	*T
6	$a \leq b$	$\neg(a > b)$	F	T	T	*T
5a	$a \geq b$	$a \geq b$	T	F	T	F
6a	$a \leq b$	$b \geq a$	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 vector/SIMD code.

Table 4-11. Other Useful Predicates

Case	Predicate	Vector Realization	Relations			
			a>b	a<b	a=b	?
7	$a ? b$	$\neg((a=b) \vee (b>a) \vee (a>b))$	F	F	F	T
8	$a <> b$	$(a \geq b) \oplus (b \geq a)$	T	T	F	F
9	$a <=> b$	$(a \geq b) \vee (b \geq a)$	T	T	T	F
10	$a ?> b$	$\neg(b \geq a)$	T	F	F	T
11	$a ?\geq b$	$\neg(b > a)$	T	F	T	T
12	$a ?< b$	$\neg(a \geq b)$	F	T	F	T
13	$a ?\leq b$	$\neg(a > b)$	F	T	T	T
14	$a ?= b$	$\neg((a > b) \vee (b > a))$	F	F	T	T

Vector/SIMD Multimedia Extension Technology

- The vector floating-point compare instructions compare 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 **vA** are within the bounds specified by the corresponding element in **vB**, 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 ≤ x ≤ y).

Table 4-12 describes the floating-point compare instructions.

Table 4-12. Vector Floating-Point Compare Instructions (Page 1 of 2)

Name	Mnemonic	Syntax	Operation
Vector Compare Greater Than Floating- Point [Record]	vcmpgtfp[.]	vD,vA,vB	<p>Each single-precision floating-point word element in register vA is compared to the corresponding single-precision floating-point word element in register vB. The corresponding word element in register vD is set to all '1's if the element in register vA is greater than the element in register vB, and is set to all '0's otherwise.</p> <p>If Rc = 1, CR Field 6 is set.</p> <p>CR6 = all_greater_than 0b0 none_greater_than 0b0.</p> <p>Note: If a vA or vB element is a NaN, the corresponding results will be 0x0000_0000.</p>

Vector/SIMD Multimedia Extension Technology**Table 4-12. Vector Floating-Point Compare Instructions (Page 2 of 2)**

Name	Mnemonic	Syntax	Operation
Vector Compare Equal to Floating-Point [Record]	vcmpqfp[.]	vD,vA, vB	<p>Each single-precision floating-point word element in register vA is compared to the corresponding single-precision floating-point word element in register vB. The corresponding word element in register vD is set to all '1's if the element in register vA is equal to the element in register vB, and is set to all '0's otherwise.</p> <p>If $Rc=1$, CR Field 6 is set</p> $CR6 = \text{all_equal_to} \parallel 0b0 \parallel \text{none_equal_to} \parallel 0b0$ <p>Note: If a register vA or register vB element is a NaN, the corresponding result will be 0x0000_0000.</p>
Vector Compare Greater Than or Equal to Floating-Point [Record]	vcmpgeqfp[.]	vD,vA, vB	<p>Each single-precision floating-point word element in register vA is compared to the corresponding single-precision floating-point word element in register vB. The corresponding word element in register vD is set to all '1's if the element in register vA is greater than or equal to the element in register vB, and is set to all '0's otherwise.</p> <p>If $Rc = 1$, CR Field 6 is set:</p> $CR6 = \text{all_greater_or_equal} \parallel 0b0 \parallel \text{none_greater_or_equal} \parallel 0b0$ <p>Note: If a vA or vB element is a NaN, the corresponding results will be 0x0000_0000.</p>
Vector Compare Bounds Floating-Point [Record]	vcmpbfp[.]	vD,vA, vB	<p>Each single-precision floating-point word element in register vA is compared to the corresponding single-precision floating-point word element in register vB. A 2-bit value is formed that indicates whether the element in register vA is within the bounds specified by the element in register vB, as follows.</p> <p>Bit [0] of the 2-bit value is zero if the element in register vA is less than or equal to the element in register vB, and is one otherwise. Bit [1] of the 2-bit value is zero if the element in register vA is greater than or equal to the negative of the element in register vB, and is one otherwise.</p> <p>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 register vD and the remaining bits of the element are set to 0.</p> <p>If $Rc=1$, CR Field 6 is set to indicate whether all four elements in register vA are within the bounds specified by the corresponding element in register vB, as follows:</p> $CR6 = 0b00 \parallel \text{all_within_bounds} \parallel 0$ <p>Note: If any single-precision floating-point word element in register vB is negative, the corresponding element in register vA is out of bounds. Note that if a register vA or a register vB element is a NaN, the two high order bits of the corresponding result will both have the value 1.</p>

4.2.2.6 Vector Floating-Point Estimate Instructions

Table 4-13 describes the floating-point estimate instructions.

Table 4-13. Vector Floating-Point Estimate Instructions

Name	Mnemonic	Syntax	Operation
Vector Reciprocal Estimate Floating-Point	vrefp	vD,vB	The single-precision floating-point estimate of the reciprocal of each single-precision floating-point element in register vB is placed into the corresponding element in register vD .
Vector Reciprocal Square Root Estimate Floating-Point	vrsqrtefp	vD,vB	The single-precision estimate of the reciprocal of the square root of each single-precision element in register vB is placed into the corresponding word element of register vD .
Vector Log2 Estimate Floating-Point	vlogefp	vD,vB	The single-precision floating-point estimate of the base 2 logarithm of each single-precision floating-point word element in register vB is placed into the corresponding element in register vD .
Vector 2 Raised to the Exponent Estimate Floating-Point	vexptefp	vD,vB	The single-precision floating-point estimate of 2 raised to the power of each single-precision floating-point element in register vB is placed into the corresponding element in register vD .

4.2.3 Vector Load and Store Instructions

Only very basic load and store operations are provided in the Vector ISA. This keeps the circuitry in the memory path fast so that the latency of memory operations will be low. Instead, a powerful set of field manipulation instructions is provided to manipulate data into the required 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 quadword vector between memory and vector multimedia registers. Load vector left/right indexed (**lvlx**, **lvxl**, **lvrx**, **lvrxl**) and store vector left/right indexed (**stvlx**, **stvlxl**, **stvrx**, **stvrxl**) instructions are provided in the Cell Broadband Engine to transfer an unaligned quadword vector between memory and vector multimedia registers. Load vector element indexed (**lvebx**, **lvehx**, **lviewx**) and store vector element indexed instructions (**stvebx**, **stvehx**, **stviewx**) transfer byte, halfword, and word scalar elements between memory and vector multimedia registers.

All vector loads and vector stores use the index (**rA|0 + rB**) addressing mode to specify the target memory address. The Vector ISA does not provide any update forms. An **lvebx**, **lvehx**, or **lviewx** instruction transfers a scalar data element from memory into the destination vector register, leaving other elements in the vector with boundedly undefined values. An **stvebx**, **stvehx**, or **stviewx** instruction transfers a scalar data element from the source vector register to memory leaving other elements in the quadword 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. Quadword memory accesses made by **lvx**, **lvxl**, **stvx**, and **stvxl** instructions are not guaranteed to be atomic.

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 help 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



Vector/SIMD Multimedia Extension Technology

(**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 3.1.4 Quadword Data Alignment*.

The **lvx**, **lvxl**, **lvlx**, **lvlxl**, **lvrx**, **lvrxl**, **stvx**, **stvxl**, **stvlx**, **stvlxl**, **stvrx**, and **stvrxl** 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 quadword boundaries for maximum performance.

4.2.3.2 Vector Load and Store Address Generation

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

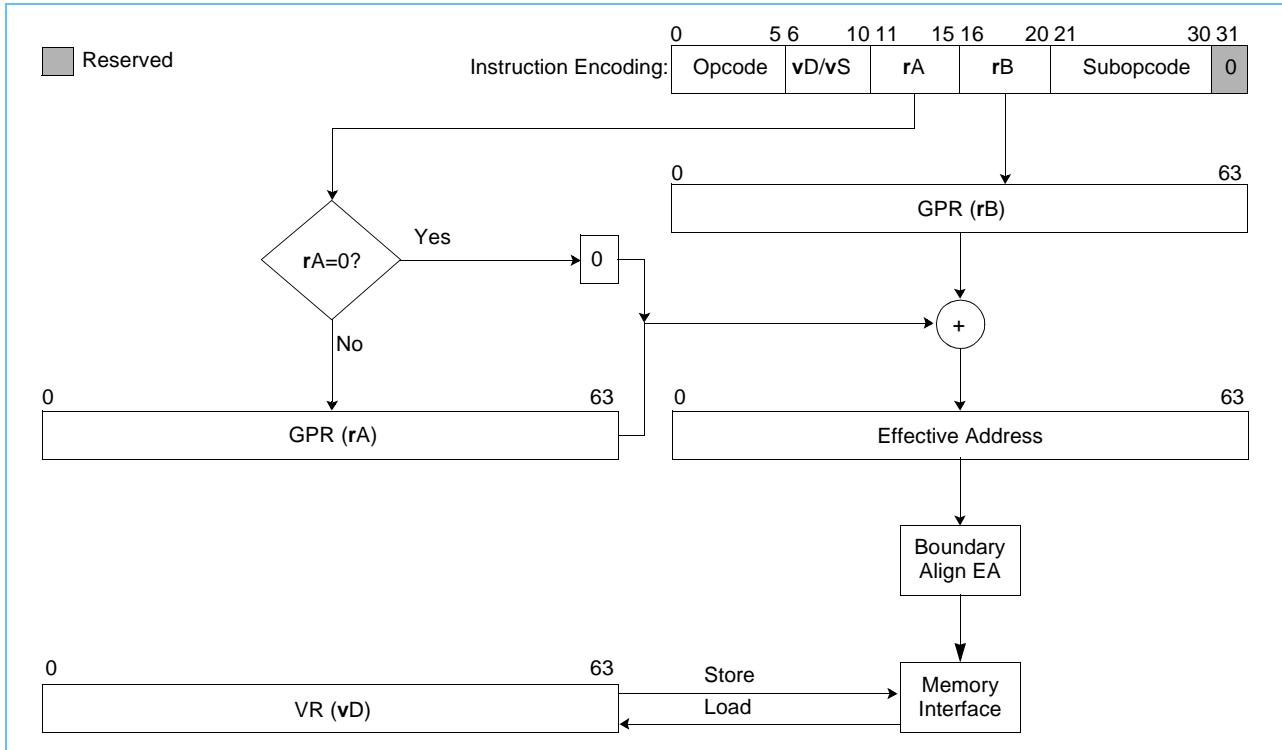
All vector load and store instructions use register indirect with index addressing mode that causes the contents of two general-purpose registers (specified as operands **rA** and **rB**) to be added in the generation of the effective address (EA). A zero in place of the **rA** operand causes a zero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

Table 4-14. Effective Address Alignment

Operand	Effective Address Bit	Setting
Indexed Halfword	EA[63]	'0'
Indexed Word	EA[62–63]	'00'
Indexed Quadword	EA[60–63]	'0000'

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



Vector/SIMD Multimedia Extension Technology**4.2.3.3 Vector Load Instructions**

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

Table 4-15 describes the vector load instructions.

Table 4-15. Vector Load Instructions (Page 1 of 2)

Name	Mnemonic	Syntax	Operation
Load Vector Element Byte Indexed	lvebx	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <ul style="list-style-type: none"> • For 64-bit implementations, let $eb = EA[60-63]$ <p>The byte in memory addressed by the EA is loaded into the byte element eb in register vD.</p> <p>The remaining bytes in register vD are set to undefined values.</p>
Load Vector Element Halfword Indexed	lvehx	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <p>For 64-bit implementations:</p> <ul style="list-style-type: none"> • Bit [63] of EA is set to 0. • Let $eb = EA[60-62]$. <p>The halfword in memory addressed by the EA is loaded into the halfword element eb in register vD.</p> <p>The remaining halfwords in register vD are set to undefined values.</p>
Load Vector Element Word Indexed	lvewx	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <ul style="list-style-type: none"> • For 64-bit implementations: <ul style="list-style-type: none"> • Bits [62-63] of EA are set to '0'. • Let $eb = EA[60-61]$. <p>The word in memory addressed by the EA is loaded into the word element eb in register vD.</p> <p>The remaining words in register vD are set to undefined values.</p>

Note:

1. The Load Vector Left/Right Indexed [Last] instructions described above are supported only on the Cell Broadband Engine processor and are not portable to other processors that support vector/SIMD processing.

Table 4-15. Vector Load Instructions (Page 2 of 2)

Name	Mnemonic	Syntax	Operation
Load Vector Indexed [Last]	lvx lvxl	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <ul style="list-style-type: none"> For 64-bit implementations, bits [60-63] of EA are set to '0'. <p>Load the quadword in memory addressed by the EA into vD.</p> <p>lvxl provides a hint that the quadword in memory addressed by EA will probably not be needed again by the program in the near future.</p>
Load Vector Left Indexed [Last] ¹	lvlx lvxl1	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <p>Let eb be the value of bits [60-63] of EA.</p> <p>The 16-eb bytes in memory addressed by EA are loaded into the left-most 16-eb byte elements of vD. The right-most eb byte elements of vD are set to '0'.</p> <p>lvxl1 provides a hint that the quadword in memory addressed by EA will probably not be needed again by the program in the near future.</p>
Load Vector Right Indexed [Last] ¹	lvrx lvrxl	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <p>Let eb be the value of bits [60-63] of EA.</p> <p>If eb is not equal to '0' (for example, EA is not quadword-aligned), then eb bytes in memory addressed by EA-eb are loaded into the right-most eb byte elements of vD and the left-most 16-eb byte elements of vD are set to '0'.</p> <p>If eb is equal to '0' (for example, EA is quadword-aligned), then the contents of vD are set to '0'.</p> <p>lvrxl provides a hint that the quadword in memory addressed by the EA will probably not be needed again by the program in the near future.</p>

Note:

1. The Load Vector Left/Right Indexed [Last] instructions described above are supported only on the Cell Broadband Engine processor and are not portable to other processors that support vector/SIMD processing.

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 vA and vB 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 **lvsr** 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 vA and the vB register for **vperm**. For shifting left, the vB register for **vperm** should be a register containing all zeros and vA should contain the value to be shifted, and vice versa for shifting right. For further examples on how to align the data, see *Section 3.1.4 Quadword Data Alignment*.

Vector/SIMD Multimedia Extension Technology

Table 4-16 describes the vector alignment instructions.

Table 4-16. Vector Load Instructions Supporting Alignment

Name	Mnemonic	Syntax	Operation
Load Vector for Shift Left	lvsl	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <ul style="list-style-type: none"> For 64-bit implementations, let sh = bits [60-63] of EA. <p>Let X be the 32-byte value 0x00 0x01 0x02 ... 0x1E 0x1F. Bytes sh:sh+15 of X are placed into register vD.</p>
Load Vector for Shift Right	lvsr	vD,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <ul style="list-style-type: none"> For 64-bit implementations, let sh = bits [60-63] of EA. <p>Let X be the 32-byte value 0x00 0x01 0x02 ... 0x1E 0x1F. Bytes (16-sh):(31-sh) of X are placed into register vD.</p>

4.2.3.4 Vector Store Instructions

For vector store instructions, the contents of a vector register used as a source (**vS**) are stored into the byte, halfword, word or quadword in memory addressed by the effective address (EA). *Table 4-17* describes the vector store instructions.

Table 4-17. Vector Store Instructions (Page 1 of 2)

Name	Mnemonic	Syntax	Operation
Store Vector Element Byte Indexed	stvebx	vS,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <ul style="list-style-type: none"> For 64-bit implementations, let $eb = EA[60-63]$ <p>The contents of byte element eb in register vS is stored into the byte in memory addressed by EA.</p>
Store Vector Element Halfword Indexed	stvehx	vS,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <p>For 64-bit implementations:</p> <ul style="list-style-type: none"> Bit [63] of EA is set to '0'. Let $eb = EA[60-62]$. <p>The contents of halfword element eb in register vS is stored into the halfword in memory addressed by EA.</p>
Store Vector Element Word Indexed	stvewx	vS,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <p>For 64-bit implementations, let $eb = EA[60:61]$:</p> <ul style="list-style-type: none"> Bits [62-63] of EA are set to '0'. Let $eb = EA[60:61]$. <p>The contents of word element eb in register vS is stored into the word in memory addressed by EA.</p>

Note:

1. The Store Vector Left/Right Indexed [Last] instructions described above are supported only on the Cell Broadband Engine processor and are not portable to other processors that support vector/SIMD processing.



Vector/SIMD Multimedia Extension Technology

Table 4-17. Vector Store Instructions (Page 2 of 2)

Name	Mnemonic	Syntax	Operation
Store Vector Left Indexed [Last] ¹	stvlx stvlxl	vS,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <p>Let eb be the value of bits [60-63] of EA.</p> <p>The contents of the left-most 16-eb byte elements of vS are stored into the 16-eb bytes of memory addressed by EA.</p> <p>stvlxl provides a hint that the quadword in memory addressed by EA will probably not be needed again by the program in the near future.</p>
Store Vector Right Indexed [Last] ¹	stvrx stvrxl	vS,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <p>Let eb be the value of bits [60-63] of EA.</p> <p>If eb is not equal to '0' (for example, EA is not quadword-aligned), then the contents of the right-most eb byte elements of vS are stored into the eb bytes of memory addressed by EA-eb.</p> <p>If eb is equal to '0' (for example, EA is quadword-aligned), then memory is not altered by this instruction.</p> <p>stvrxl provides a hint that the quadword in memory addressed by EA will probably not be needed again by the program in the near future.</p>
Store Vector Indexed [Last]	stvx stvxl	vS,rA,rB	<p>Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.</p> <ul style="list-style-type: none"> • For 64-bit implementations, bits [60-63] of EA are set to '0' <p>Store the contents of vS into the quadword in memory addressed by the EA.</p> <p>stvxl provides a hint that the quadword in memory addressed by EA will probably not be needed again by the program in the near future.</p>

Note:

1. The Store Vector Left/Right Indexed [Last] instructions described above are supported only on the Cell Broadband Engine processor and are not portable to other processors that support vector/SIMD processing.

4.2.4 Control Flow

Vector instructions can be freely intermixed with existing PowerPC instructions to form a complete program. Vector instructions do provide a vector compare and select mechanism to implement conditional execution as the preferred mechanism to control data flow in vector programs. In addition, vector compare instructions can update the Condition Register thus providing the communication from vector 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

Halfword vector pack instructions (**vpkuhum**, **vpkuhus**, **vpkshus**, **vpkshss**) truncate the 16 halfwords from two concatenated source operands producing a single result of 16 bytes (quadword) 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 8 words from two concatenated source operands producing a single result of 8 halfwords using modulo(2^{16}), 16-bit signed-saturation, or 16-bit unsigned-saturation to perform the truncation.

Vector/SIMD Multimedia Extension Technology

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 α RGB 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 least significant bits.

Table 4-18 describes the vector pack instructions.

Table 4-18. Vector Pack Instructions

Name	Mnemonic	Syntax	Operation
Vector Pack Unsigned Integer Unsigned Modulo	vpkuhum vpuwum	vD, vA, vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vpkuhum, let n = 16 bits. For vpuwum, let n = 32 bits. <p>Let the source vector be the concatenation of the contents of register vA followed by the contents of register vB.</p> <p>The low-order (n÷2) bits of each n-bit element in the source vector is placed into the corresponding (n÷2)-bit element of register vD.</p>
Vector Pack Unsigned Integer Unsigned Saturate	vpkuhus vpuwus	vD, vA, vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vpkuhus, let n = 16 bits. For vpuwus, let n = 32 bits. <p>Let the source vector be the concatenation of the contents of register vA followed by the contents of register vB.</p> <p>Each n-bit unsigned integer element in the source vector is converted to an (n÷2)-bit unsigned integer. If the value of the element is greater than $2^{(n-2)}-1$, the result saturates to $2^{(n-2)}-1$. If saturation occurs, the SAT bit is set. The result is placed into the corresponding (n÷2)-bit element of register vD.</p>
Vector Pack Signed Integer Unsigned Saturate	vpkshus vpkswus	vD, vA, vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vpkshus, let n = 16 bits. For vpkswus, let n = 32 bits. <p>Let the source vector be the concatenation of the contents of register vA followed by the contents of register vB.</p> <p>Each n-bit signed integer element in the source vector is converted to an (n÷2)-bit unsigned integer. If the value of the element is greater than $2^{(n-2)}-1$, the result saturates to $2^{(n-2)}-1$. If the value of the element is less than '0', the result saturates to '0'. If saturation occurs, the SAT bit is set. The result is placed into the corresponding (n÷2)-bit element of register vD.</p>
Vector Pack Signed Integer Signed Saturate	vpkshss vpkswss	vD, vA, vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vpkshss, let n = 16 bits. For vpkswss, let n = 32 bits. <p>Let the source vector be the concatenation of the contents of register vA followed by the contents of register vB.</p> <p>Each n-bit signed integer element in the source vector is converted to an (n÷2)-bit signed integer. If the value of the element is greater than $2^{(n-2)}-1$, the result saturates to $2^{(n-2)}-1$. If the value of the element is less than $-2^{(n-2)}+1$, the result saturates to $-2^{(n-2)}+1$. If saturation occurs, the SAT bit is set. The result is placed into the corresponding (n÷2)-bit element of register vD.</p>
Vector Pack Pixel	vpkpx	vD, vA, vB	<p>The source vector is the concatenation of the contents of register vA followed by the contents of register vB.</p> <p>Each word element in the source vector is packed to produce a 16-bit value as described below and placed into the corresponding halfword element of register vD. A word is packed to 16 bits by concatenating, in order, the following bits.</p> <ul style="list-style-type: none"> 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)

Vector/SIMD Multimedia Extension Technology**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 halfwords using sign extension to fill the most significant bytes (MSBs). Halfword vector unpack instructions unpack the 4 low halfwords (or 4 high halfwords) of one source operand into 4 words using sign extension to fill the msb(s).

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 α RGB pixels. The 1/5/5 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 on page 84 describes the unpack instructions.

Table 4-19. Vector Unpack Instructions

Name	Mnemonic	Syntax	Operation
Vector Unpack High Signed Integer	vupkhsb vupkhsh	vD, vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vupkhsb, let n = 8 bits. For vupkhsh, let n = 16 bits. <p>Each n-bit signed integer element in the high-order half of register vB is sign-extended to produce a (2xn)-bit signed integer and placed, in the same order, into register vD.</p>
Vector Unpack High Pixel	vupkhpx	vD, vB	<p>Each halfword element in the high-order half of register vB is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of register vD.</p> <p>A halfword is unpacked to 32 bits by concatenating, in order, the results of the following operations:</p> <ul style="list-style-type: none"> 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
Vector Unpack Low Signed Integer	vupklsb vupklsh	vD, vB	<p>Let n = element length.</p> <ul style="list-style-type: none"> For vupklsb, let n = 8 bits. For vupklsh, let n = 16 bits. <p>Each n-bit signed integer byte element in the low-order half of register vB is sign-extended to produce a (nx2)-bit signed integer and placed, in the same order, into register vD.</p>
Vector Unpack Low Pixel	vupklpx	vD, vB	<p>Each halfword element in the low-order half of register vB is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of register vD.</p> <p>A halfword is unpacked to 32 bits by concatenating, in order, the results of the following operations.</p> <ul style="list-style-type: none"> 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

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, halfword vector merge instructions interleave the 4 low halfwords (or 4 high halfwords) of two source operands producing a result of 8 halfwords. 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	vD, vA, vB	Let n = element length. <ul style="list-style-type: none">• For vmrghb, let n = 8 bits.• For vmrghh, let n = 16 bits.• For vmrghw, let n = 32 bits. The n-bit elements in the high-order half of register vA are placed, in the same order, into the even-numbered n-bit elements of register vD . The n-bit elements in the high-order half of register vB are placed, in the same order, into the odd-numbered n-bit elements of register vD .
Vector Merge Low Integer	vmrglb vmrglh vmrglw	vD, vA, vB	Let n = element length. <ul style="list-style-type: none">• For vmrglb, let n = 8 bits.• For vmrglh, let n = 16 bits.• For vmrglw, let n = 32 bits. The n-bit elements in the low-order half of register vA are placed, in the same order, into the even-numbered n-bit elements of register vD . The n-bit elements in the low-order half of register vB are placed, in the same order, into the odd-numbered n-bit elements of register vD .

Vector/SIMD Multimedia Extension Technology**4.2.5.4 Vector Splat Instructions**

When a program needs to perform an 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	vD, vB, UIMM	<p>Let n = element length.</p> <ul style="list-style-type: none"> • For vspltb, let n = 8 bits. • For vsplth, let n = 16 bits. • For vspltw, let n = 32 bits. <p>The contents of n-bit element UIMM in register vB are replicated into each n-bit element of register vD.</p>
Vector Splat Immediate Signed Integer	vspltsb vspltsih vspltsw	vD, SIMM	<p>Let n = element length.</p> <ul style="list-style-type: none"> • For vspltsb, let n = 8 bits. • For vspltsih, let n = 16 bits. • For vspltsw, let n = 32 bits. <p>The value of the signed immediate value (SIMM) field, sign-extended to the n bits, is replicated into each n-bit element of register vD.</p>

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. For an example of how to use the instruction in aligning data, see *Section 3.1.4 Quadword Data Alignment*. *Table 4-22* describes the vector permute instruction.

Table 4-22. Vector Permute Instruction

Name	Mnemonic	Syntax	Operation
Vector Permute	vperm	vD, vA,vB,vC	<p>Let the source vector be the concatenation of the contents of register vA followed by the contents of register vB.</p> <p>For each integer i in the range 0 to 15, the contents of the byte element in the source vector specified in bits [3:7] of byte element i in register vC are placed into byte element i of register vD.</p>

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	vD,vA,vB,vC	For each bit in register vC that contains the value 0, the corresponding bit in register vA is placed into the corresponding bit of register vD. For each bit in register vC that contains the value '1', the corresponding bit in register vB is placed into the corresponding bit of register vD.

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 interelement 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 (Page 1 of 2)

Name	Mnemonic	Syntax	Operation
Vector Shift Left	vsl	vD,vA,vB	<p>Let sh be equal to the contents of bits 125:127 of register vB; sh is the shift count in bits ($0 \leq \text{sh} \leq 7$).</p> <p>The contents of register vA are shifted left by sh bits. Bits shifted out of bit [0] are lost. Zeros are supplied to the vacated bits on the right. The result is placed into register vD.</p> <p>The contents of the low-order 3 bits of all byte elements in register vB must be identical to the contents of bits [125-127] of register vB; otherwise, the value placed into register vD is undefined.</p>
Vector Shift Left Double by Octet Immediate	vsldoi	vD,vA,vB,SH	Let the source vector be the concatenation of the contents of register vA followed by the contents of register vB. Bytes SHB:SHB + 15 of the source vector are placed into register vD.

Vector/SIMD Multimedia Extension Technology**Table 4-24. Vector Shift Instructions (Page 2 of 2)**

Name	Mnemonic	Syntax	Operation
Vector Shift Left by Octet	vslo	vD,vA,vB	The contents of register vA are shifted left by the number of bytes specified by the contents of bits [121-124] of register vB . Bytes shifted out of byte 0 are lost. Zeros are supplied to the vacated bytes on the right. The result is placed into register vD .
Vector Shift Left	vsr	vD,vA,vB	Let sh be equal to the contents of bits 125:127 of register vB ; sh is the shift count in bits ($0 \leq sh \leq 7$). The contents of register vA 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 register vD . The contents of the low-order 3 bits of all byte elements in register vB must be identical to the contents of bits [125-127] of register vB ; otherwise, the value placed into register vD is undefined.
Vector Shift Right by Octet	vsro	vD,vA,vB	The contents of register vA are shifted right by the number of bytes specified by the contents of bits [121-124] of register vB . Bytes shifted out of vA are lost. Zeros are supplied to the vacated bytes on the left. The result is placed into register vD .

Immediate Interelement Shifts/Rotates

The Vector Shift Left Double by Octet Immediate (**vsidoi**) instruction provides the basic mechanism that can be used to provide interelement shifts 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 **vA:vB**, 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

MSV = most significant vector
LSV = least significant vector

Computed Interelement Shifts/Rotates

The Load Vector for Shift Left (**lvsI**) instruction and Load Vector for Shift Right (**lvsR**) instruction are supplied to assist in shifting 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, they use register indirect with index addressing mode(**rA|0 +rB**). 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 an 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 vector.

Variable Interelement Shifts

A vector register can 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 least significant bits 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 **VX** by the number of bits specified in **VY** and place the results in **VZ**.

With these instructions, a full double-register shift can be performed in seven instructions. The following code will shift **VW|VX** left by the number of bits specified in **VY** placing the result in **VZ**:

```
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 Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors*, for information about the instructions used for reading from and writing to the MSR and SPRs.

Vector/SIMD Multimedia Extension Technology**4.2.6.1 Vector Status and Control Register Instructions**

Table 4-26 summarizes the instructions for reading from or writing to the Vector Status and Control Register (VSCR). For more information about VSCR, see section in *Section 2.2.2 Vector Status and Control Register*.

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

Name	Mnemonic	Syntax	Operation
Move to Vector Status and Control Register	mtvscr	vB	Place the contents of vB into VSCR.
Move from Vector Status and Control Register	mfvscr	vD	Place the contents of VSCR into vD .



4.3 Vector VEA Instructions

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 UIISA, but may not necessarily adhere to the OEA. For further details, see Chapter 4, “Addressing Mode and Instruction Set Summary,” in *PowerPC Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors*.

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

4.3.1 Memory Control Instructions—VEA

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 Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors* for more information about supervisor-level cache, segment register manipulation, and TLB management instructions.

4.3.2 User-Level Cache Instructions—VEA

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 Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors* 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 4-27 summarizes the directed prefetch cache instructions defined by the VEA.

Note: These instructions are accessible to user-level programs.

Vector/SIMD Multimedia Extension Technology**Table 4-27. User-Level Cache Instructions (Page 1 of 2)**

Name	Mnemonic	Syntax	Operation
Data Stream Touch (nontransient, T = 0)	dst	rA,rB,STRM	<p>This instruction associates the data stream specified by the contents of rA and rB with the stream ID specified by STRM.</p> <p>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 data stream is likely to be loaded from fairly frequently in the near future. The hint is ignored for blocks that are Caching Inhibited.</p> <p>The specified data stream is defined by the following values:</p> <ul style="list-style-type: none"> EA: (rA), where rA ≠ 0; if rA = 0, the instruction form is invalid. unit size: (rB)[35–39] if (rB)[35–39] ≠ 0; otherwise 32 count: (rB)[40–47] if (rB)[40–47] ≠ 0; otherwise 256 stride: (rB)[48–63] if (rB)[48–63] ≠ 0; otherwise 32768
Data Stream Touch (transient, T = 1)	dstt	rA,rB,STRM	<p>This instruction associates the data stream specified by the contents of rA and rB with the stream ID specified by STRM.</p> <p>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 data stream is likely to be transient and referenced very few times. The hint is ignored for blocks that are Caching Inhibited.</p> <p>The specified data stream is defined by the following values:</p> <ul style="list-style-type: none"> EA: (rA), where rA ≠ 0; if rA = 0, the instruction form is invalid. unit size: (rB)[35–39] if (rB)[35–39] ≠ 0; otherwise 32 count: (rB)[40–47] if (rB)[40–47] ≠ 0; otherwise 256 stride: (rB)[48–63] if (rB)[48–63] ≠ 0; otherwise 32768
Data Stream Touch for Store (nontransient, T = 0)	dstst	rA,rB,STRM	<p>This instruction associates the data stream specified by the contents of registers rA and rB with the stream ID specified by STRM.</p> <p>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 store into the stream, and that prefetching from any data stream that was previously associated with the specified stream ID is no longer needed. The data stream is likely to be stored into fairly frequently in the near future. The hint is ignored for blocks that are Caching Inhibited.</p> <p>The specified data stream is defined by the following values:</p> <ul style="list-style-type: none"> EA: (rA), where rA ≠ 0; if rA = 0, the instruction form is invalid. unit size: (rB)[35–39] if (rB)[35–39] ≠ 0; otherwise 32 count: (rB)[40–47] if (rB)[40–47] ≠ 0; otherwise 256 stride: (rB)[48–63] if (rB)[48–63] ≠ 0; otherwise 32768

Note: The Cell Broadband Engine processor treats the instructions described above as no-ops. Applications should use the Data Cache Block Touch instructions described in the *PowerPC Virtual Environment Architecture, Book II* to prefetch data on the Cell Broadband Engine processor.

Table 4-27. User-Level Cache Instructions (Page 2 of 2)

Name	Mnemonic	Syntax	Operation
Data Stream Touch for Store (transient, T = 1)	dsttt	rA,rB,STRM	<p>This instruction associates the data stream specified by the contents of rA and rB with the stream ID specified by STRM.</p> <p>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 store into the stream, and that prefetching from any data stream that was previously associated with the specified stream ID is no longer needed. The data stream is likely to be transient and referenced very few times. The hint is ignored for blocks that are Caching Inhibited.</p> <p>The specified data stream is defined by the following values:</p> <ul style="list-style-type: none"> EA: (rA), where rA ≠ 0; if rA = 0, the instruction form is invalid. unit size: (rB)[35–39] if (rB)[35–39] ≠ 0; otherwise 32 count: (rB)[40–47] if (rB)[40–47] ≠ 0; otherwise 256 stride: (rB)[48–63] if (rB)[48–63] ≠ 0; otherwise 32768
Data Stream Stop	dss	STRM,A	<p>If A = '0' and a data stream associated with the stream ID specified by STRM exists, this instruction terminates prefetching of that data stream.</p> <p>If A = '1', this instruction terminates prefetching of all existing data streams. (The STRM field is ignored.)</p> <p>In addition, executing a dss instruction ensures that all memory accesses associated with data stream prefetching caused by preceding dst and dsttt instructions that specified the same stream ID as that specified by the dss instruction (A = '0'), or by all preceding dst and dsttt instructions (A = '1'), will be in group G1 with respect to the memory barrier created by a subsequent sync instruction.</p> <p>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.</p> <p>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.</p>
Data Stream Stop All	dssall		<p>Terminates prefetching of all existing data streams. All active streams may be stopped.</p> <p>If the optional data stream prefetch facility is implemented, dssall (extended mnemonic for dss) terminates 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.</p>

Note: The Cell Broadband Engine processor treats the instructions described above as no-ops. Applications should use the Data Cache Block Touch instructions described in the *PowerPC Virtual Environment Architecture, Book II* to prefetch data on the Cell Broadband Engine processor.

4.3.3 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 unless they are described in this document.



Vector/SIMD Multimedia Extension Technology

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 4-28*.

*Table 4-28. Simplified Mnemonics for Data Stream Touch (**dst**)*

Operation	Simplified Mnemonic	Equivalent to
Data Stream Touch (nontransient)	dst rA, rB, STRM	dst rA, rB, STRM,0
Data Stream Touch (transient)	dstt rA, rB, STRM	dst rA, rB, STRM,1
Data Stream Touch for Store (nontransient)	dstst rA, rB, STRM	dstst rA, rB, STRM,0
Data Stream Touch for Store (transient)	dsttt rA, rB, STRM	dsttt rA, rB, STRM,1
Data Stream Stop (one stream)	dss STRM	dss STRM,0
Data Stream Stop All	dssall	dss 0,1

Note: The Cell Broadband Engine processor treats the instructions described above as no-ops. Applications should use the Data Cache Block Touch instructions described in the *PowerPC Virtual Environment Architecture, Book II* to prefetch data on the Cell Broadband Engine processor.



5. Cache, Exceptions, and Memory Management

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

Because the vector Instruction Set Architecture (ISA) uses 128-bit operands, additional instructions are provided to optimize cache and memory bus use.

5.1 PowerPC Shared Memory

To fully understand the data stream prefetch instructions for the vector processing unit (VPU), one needs to know 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, then 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 synchronize 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 will be performed with respect to any processor or mechanism, to the extent required by the memory coherence required attributes associated with the



Vector/SIMD Multimedia Extension Technology

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 the synchronize instruction is cumulative, and applies to all memory accesses except those associated with fetching instructions following the synchronize instruction. See the description of the **eieio** instruction in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors* 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 VPU Memory Bandwidth Management

The Vector 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 *Section 5.2.1.1*.

Notes:

- The Cell Broadband Engine processor treats data stream instructions as no-ops.
- The Cell Broadband Engine processor implements an extended definition of **dcbt**. See *PowerPC Virtual Environment Architecture, Book II* for more information.

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, just a stream. A data stream is defined by the following values:

- EA—The effective address of the first unit in the sequence
- Unit size—The number of quadwords in each unit; $0 < \text{unit size} \leq 32$
- Count—The number of units in the sequence; $0 < \text{count} \leq 256$

- 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 $EA + (n - 1) \times \text{stride}$; $-32768 \leq \text{stride} < 0$ or $0 < \text{stride} \leq 32768$)

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 **rA** (if **rA** = 0, the instruction form is invalid). **BlockSize**, **BlockCount**, and **BlockStride** are specified in **rB**. Do not confuse the term ‘cache block’; the term ‘block’ always indicates a PowerPC cache block.

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

Figure 5-1. Format of rB in dst Instruction

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

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 numeric value of the size, count, and stride. Do not specify strides smaller than 1 block (16 bytes).

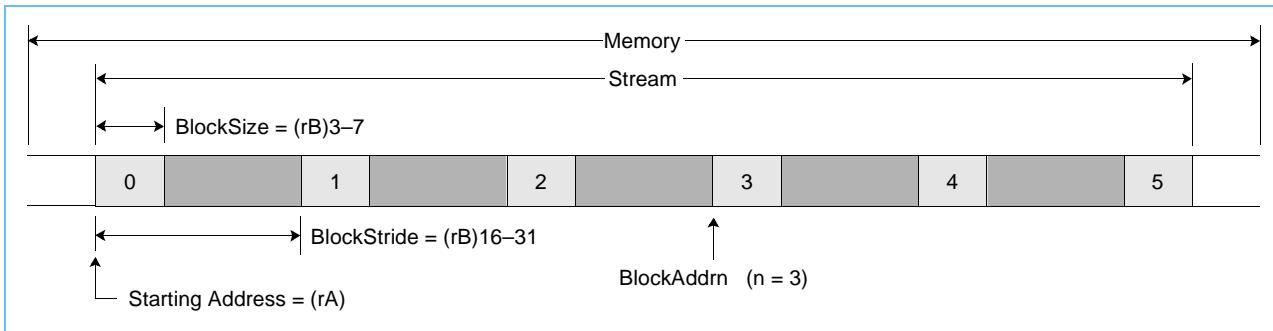
The programmer specifies block size in terms of vectors (16 bytes), regardless of the cache-block size. Hardware automatically optimizes the number of cache blocks it fetches to bring a block into the cache. The number of cache blocks fetched into the cache for each block is the fewest natural number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

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

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 can be any byte address. Each block’s address is computed as a full byte address from the following equation:

$$\text{BlockAddr}_n = (\text{rA}) + n (\text{rB})_{16-31} \quad \begin{aligned} \text{where } n &= \{0 \dots (\text{BlockCount} - 1)\} \\ \text{and } &\text{ if } ((\text{rB})_{16-31} = 0) \text{ then } ((\text{rB})_{16-31} \leftarrow 32768) \end{aligned}$$

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 $\log_2(\text{CacheBlockSize})$ 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* on page 98.

Vector/SIMD Multimedia Extension Technology*Figure 5-2. Data Stream Touch*

Executing a **dst** instruction notifies the cache/memory subsystem that the program will soon need the 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 cancels 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 cancel the stream at any time, or to prioritize other memory operations ahead of it. If a stream is canceled, 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 effectively manage the location and retention of transient data in the processor's cache/memory design. An implementation can 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 the 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 the state most efficient for subsequent writing to it (store). For example, in a modified/exclusive/shared/invalid (MESI) cache subsystem, a **dst** might bring a cache block in the shared (S) state. However, a **dstst** would bring the cache block in the exclusive (E) state to avoid a subsequent demand-driven bus transaction taking ownership of the cache block so that 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 that no more bandwidth is wasted. All active streams can 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 when 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 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
```



Vector/SIMD Multimedia Extension Technology

The **mts** instruction cannot be executed until the **lwz** loads the SR value into Ry. 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' causes 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 **dstst** 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 effects of supervisor-level data-stream prefetching can also adversely affect performance of applications that use data stream prefetching, because 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 canceled 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 **dsttt** 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 translation lookaside buffer (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 data storage interrupt (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 cannot 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.



Vector/SIMD Multimedia Extension Technology

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. By limiting the granularity of prefetches to small blocks, the adverse effect of an exception killing the prefetch is also limited to that smaller prefetch because the next prefetch will be encountered sooner.

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. Therefore, 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 required 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 that 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, including the Cell Broadband Engine processor, treat **dst**, **dstt**, **dstst**, **dss**, and **dssall** instructions as no-ops. However, if a processor implements these instructions, a minimum level of functionality will be provided to create as consistent a programming model across different machines as possible. Programs can assume the following functionality in these instructions:

- 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 cancel streams on page boundary crossings.
- Does not cancel streams on exceptions (except DSI exceptions caused by the stream).
- Does not cancel streams, or stall execution of subsequent instructions pending completion of streams, on the PowerPC synchronization instructions **sync**, **isync**, or **eieio**.
- Does not cancel 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 be canceled.

5.2.2 Prioritizing Cache Block Replacement

Load Vector Indexed Last (**lvxl**) and Store Vector Indexed Last (**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 Digital Signal Processor (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 least recently used (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 Vector 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 can be altered when floating-point load instructions cause a DSI exception, if the vector/SIMD multimedia extension facility is implemented, the target register (**vD**) can be altered when **lvx**, **lvxl**, **lvlx**, **lvlxl**, **lvrx**, or **lvrxl** is executed and the TLB entry is invalidated before the access completes. See the section about partially executed instructions in *PowerPC Operating Environment Architecture, Book III* for more information.

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 DS1 Exception—Data Address Breakpoint

The data address breakpoint mechanism provides a way to detect load and store accesses to a designated doubleword. The address comparison is done on an effective address (EA). The data address breakpoint mechanism is controlled by the Data Address Breakpoint Register (DABR) and the Data Address Breakpoint Register Extension (DABRX). See the Data Address Breakpoint section in *PowerPC Operating Environment Architecture, Book III* for more information.

5.4 VPU Unavailable Exception (0x00F20)

The vector facility includes an additional instruction-caused, precise exception to those defined by the OEA and discussed in Chapter 6, “Exceptions,” in the *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*. A VPU unavailable exception occurs when no higher priority exception exists (see *Table 5-2* on page 105), an attempt is made to execute a vector instruction, and MSR[VEC] = ‘0’.

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

Table 5-1. VPU Unavailable Exception—Register Settings

Register	Setting Description						
SRR0	Set to the effective address of the instruction that caused the exception						
SRR1	0–32 Loaded with equivalent bits from the Machine State Register (MSR) 33–36 Cleared 37–41 Loaded with equivalent bits from the MSR 42–47 Cleared 48–63 Loaded with equivalent bits from the MSR Note: Depending on the implementation, additional bits in the MSR may be copied to SRR1.						
MSR	SF ¹ ISF ¹ VEC POW ILE	1 — 0 0 —	EE PR FP ME FE0	0 0 0 — 0	SE BE FE1 IP IR	0 0 0 — 0	DR 0 RI 0 LE Set to value of ILE
1. 64-bit implementations only							

When a VPU unavailable exception is taken, instruction execution resumes at offset 0x00F20.

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* on page 105 (taken from the section “Exception Priorities,” in Chapter 6, “Exceptions,” *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*).



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

Priority	Exception
3 ¹	<p>Instruction dependent—When an instruction causes an exception, the exception mechanism waits for any instructions before 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.</p> <p>Note: A single instruction can cause multiple exceptions. When this occurs, those exceptions are ordered in priority as indicated in the following list:</p> <ul style="list-style-type: none">A. Integer loads and stores<ul style="list-style-type: none">a. Program illegal instructionb. DS1, Data Segment, or Alignmentc. Trace (if implemented)B. Floating-point loads and stores<ul style="list-style-type: none">a. Program illegal instructionb. Floating-point unavailablec. DS1, Data Segment, or Alignmentd. Trace (if implemented)C. Other floating-point instructions<ul style="list-style-type: none">a. Floating-point unavailableb. Program—Precise-mode floating-point enabled exceptionc. Trace (if implemented)D. Vector Loads and Stores (if VPU facility implemented)<ul style="list-style-type: none">a. VPU unavailablec. Trace (if implemented)E. rfid and mtmsrd (or mtmsr)<ul style="list-style-type: none">a. Program—Precise-mode floating-point enabled exceptionb. Trace (if implemented), for mtmsrd (or mtmsr) onlyIf 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.F. Other Vector instructions (if VPU facility implemented)<ul style="list-style-type: none">a. VPU unavailableb. Trace (if implemented)G. Other instructions<ul style="list-style-type: none">a. These exceptions are mutually exclusive and have the same priority:<ul style="list-style-type: none">— Program: Trap— System call (sc)— Program: Supervisor level instruction— Program: Illegal Instructionb. Trace (if implemented)c. VPU assist (if implemented)F. ISI exception<p>The ISI exception has the lowest priority in this category. It is only recognized when all instructions before 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.</p>

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

6. Vector Processing Instructions

This chapter lists the vector 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 bits and a description of the instruction fields and pseudocode conventions is also provided. For more information about the vector instruction set, see Chapter 4 “Addressing Modes and Instruction Set Summary,” for more information about the PowerPC instruction set, see Chapter 8, “Instruction Set,” in *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*.

6.1 Instruction Formats

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

Within a vector register, a byte, halfword, or word element, is defined as follows:

- Byte elements: Each element is 8 bits. Therefore, in the pseudocode, $n = 8$ and there are a total of 16 elements
- Halfword elements: Each element is 16 bits. Therefore, in the pseudocode, $n = 16$ and there are a total of 8 elements
- Word elements: Each element is 32 bits. Therefore, in the pseudocode, $n = 32$ and there are a total of 4 elements

See *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 (Page 1 of 2)

Field	Description
OPCD (0–5)	Primary opcode field
rA (11–15)	Specifies a general purpose register (GPR) to be used as a source.
rB (16–20)	Specifies a GPR to be used as a source.
Rc (21)	Record bit (as defined for vector/SIMD multimedia extension technology). 0 Does not update Condition Register (CR) field 6. 1 Set CR field 6 to control program flow as described in <i>Section 2.3.1 PowerPC Condition Register</i> .
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.

Vector/SIMD Multimedia Extension Technology**Table 6-1. Instruction Syntax Conventions (Page 2 of 2)**

Field	Description
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 (5 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 (Page 1 of 4)

Notation/Convention	Meaning
\leftarrow	Assignment
\neg	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
$+_{ui}$	Unsigned integer add
$-_{ui}$	Unsigned integer subtract
$*_{ui}$	Unsigned integer multiply
$+_{si}$	Signed integer add
$-_{si}$	Signed integer subtract
$*_{si}$	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



Table 6-2. Notation and Conventions (Page 2 of 4)

Notation/Convention	Meaning
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, $(a \equiv b) = (a \oplus \neg b)$)
0bnnnn	A number expressed in binary format.
0xnnnn	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:Z}	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.
ROTL(x,y)	Result of rotating x left by y bits
UltToUlmod(X,Y)	Chop unsigned integer X to Y-bit unsigned integer
UltToUlsat(X,Y)	Result of converting the unsigned-integer x to a y-bit unsigned-integer with unsigned-integer saturation
SltToUlsat(X,Y)	Result of converting the signed-integer x to a y-bit unsigned-integer with unsigned-integer saturation
SltToSlmod(X,Y)	Chop integer X- to Y-bit integer
SltToSlsat(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
Cnvtp32ToUI32Sat(x)	Result of converting the single-precision floating-point value x to a 32-bit unsigned-integer with unsigned-integer saturation
Cnvtp32ToSI32Sat(x)	Result of converting the single-precision floating-point value x to a 32-bit signed-integer with signed-integer saturation



Vector/SIMD Multimedia Extension Technology

Table 6-2. Notation and Conventions (Page 3 of 4)

Notation/Convention	Meaning
CnvUI32ToFP32(x)	Result of converting the 32-bit unsigned-integer x to floating-point single format
CnvSI32ToFP32(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
Ulmaximum(X,Y)	Maximum of 2 unsigned integer values, X and Y
Slmaximum(X,Y)	Maximum of 2 signed integer values, X and Y
FPmaximum(X,Y)	Maximum of 2 floating-point values, X and Y
Ulminimum(X,Y)	Minimum of 2 unsigned integer values, X and Y
Slminimum(X,Y)	Minimum of 2 signed integer values, X and Y
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 log ₂ (X)
FPPower2Estimate3(X)	3-bit-accurate floating-point estimate of 2**X
CarryOut(X + Y)	Carry out of the sum of X and Y
0bnnnn	A number expressed in binary format.
0xnnnn	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: <ul style="list-style-type: none">• (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.
(rA 0)	The contents of rA if the rA field has the value 1–31, or the value 0 if the rA field is 0.
(rX)	The contents of rX
x[n]	n is a bit or field within x, where x is a register
x ⁿ	x is raised to the nth power
ABS(x)	Absolute value of x
CEIL(x)	Least integer \geq 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.



Table 6-2. Notation and Conventions (Page 4 of 4)

Notation/Convention	Meaning
Clear left and shift left	Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a known nonnegative array index by the width of an element. These operations are used for rotate and shift instructions.
Cleared	Bits = '0'.
Do	Do loop. <ul style="list-style-type: none">• Indenting shows range.• "To" or "by" clauses specify incrementing an iteration variable.• "While" clauses give termination conditions.
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 doublewords; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. Note: 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.

Vector/SIMD Multimedia Extension Technology

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

Table 6-3. Instruction Field Conventions

PowerPC Architecture Specification:	Equivalent in the Vector Processing Technology Specification as:
D	d
DS	ds
FLM	FM
RA, RB, RT, RS	rA, rB, rD, rS
RA, RB, RT, RS	A, B, D, S
SI	SIMM
U	IMM
UI	UIMM
VA, VB, VC, VT, VS	vA, vB, vC, vD, vS
/, //, ///	0...0 (shaded)

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

Table 6-4. Precedence Rules

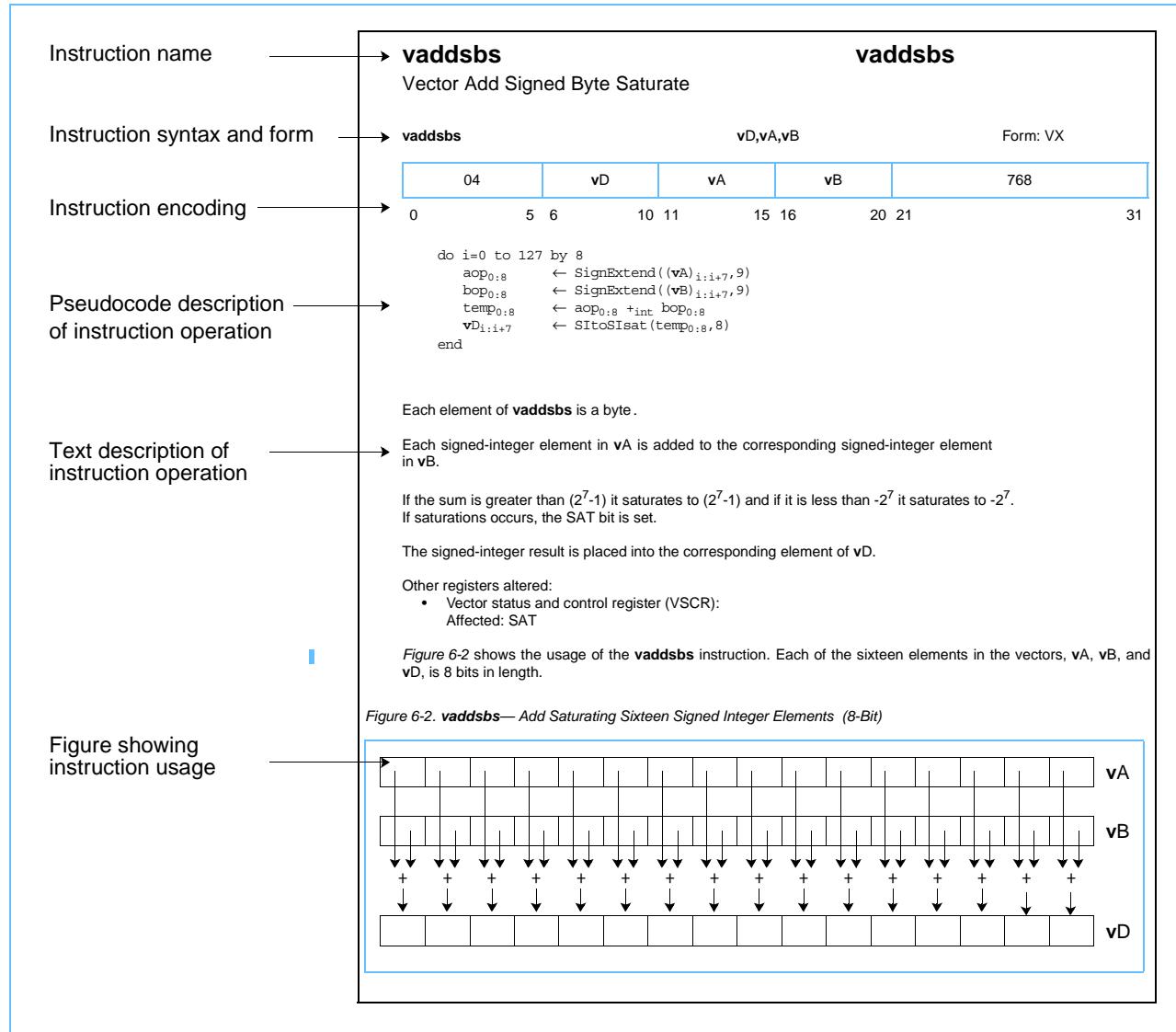
Operators	Associativity
x[n], function evaluation	Left to right
(n)x or replication, x(n) or exponentiation	Right to left
unary -, \neg	Right to left
*, \div	Left to right
+, -	Left to right
	Left to right
=, \neq , $<$, \leq , $>$, \geq , $<U$, $>U$, ?	Left to right
&, \oplus , \equiv	Left to right
	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, '-' associates from left to right, so that $a - b - c = (a - b) - c$. 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 Vector Instruction Set

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

Figure 6-1. Format for Instruction Description Page



dss

Data Stream Stop (0x7C00 066C)

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

31	A	0 0	STRM	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: A does not represent rA 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, see *Section 5.1.1 PowerPC Memory Access Ordering* for more information.

See *Section 5.2.1 Software-Directed Prefetch* for more information about 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 about the **dss** instruction, see *Section 5 Cache, Exceptions, and Memory Management* on page 95."

Note: The Cell Broadband Engine processor treats **dss** and **dsall** as no-ops.

dst**dst**

Data Stream Touch (0x7C00 02AC)

dst
dstt**rA,rB,STRM**
rA,rB,STRM(T='0')
(T='1')

Form: X

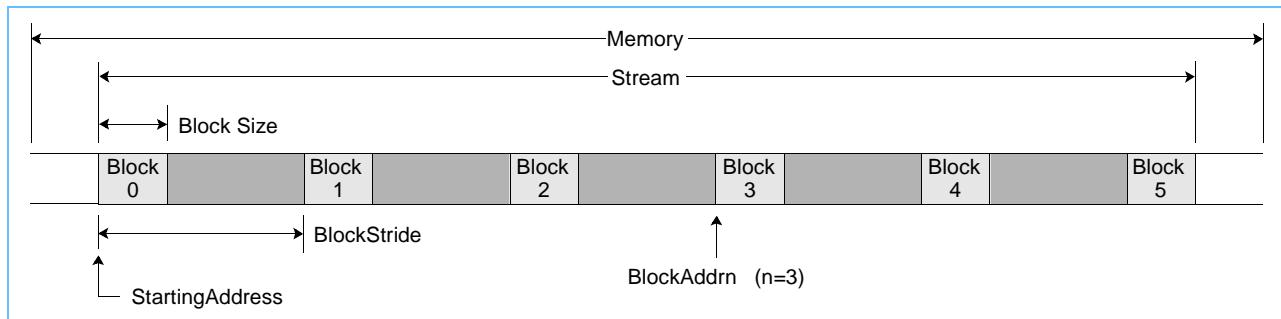


```
addr0:63 ← (rA)
DataStreamPrefetchControl ← "start" || STRM || T || (rB) || addr
```

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 **rA** and **rB** with the stream ID specified by **STRM**. The instruction defines a data stream **STRM** as starting at an “Effective Address” (**rA**) and having “Count” units of “Size” bytes separated by “Stride” bytes (as specified in **rB**). 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’).

Figure 6-3. **dst**—Data Stream Touch



The **dst** instruction performs the following functions:

- Defines the characteristics of a data stream **STRM** by the contents of **rA** and **rB**
- 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 **rA** 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**

Vector/SIMD Multimedia Extension Technology

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

- Effective Address: **rA**, where **rA** ≠ '0'
- Block Size: **rB[35-39]** if **rB[35-39]** ≠ '0'; otherwise 32
- Block Count: **rB[40-47]** if **rB[40-47]** ≠ '0'; otherwise 256
- Block Stride: **rB[48-63]** (signed integer) if **rB[48-63]** ≠ '0'; otherwise 32768

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

Other registers altered:

- None

Simplified mnemonics:

dst	rA,rB,STRM	equivalent to	dst	rA,rB,STRM,0
dstt	rA,rB,STRM	equivalent to	dst	rA,rB,STRM,1

For more information about the **dst** instruction, see *Section 5 Cache, Exceptions, and Memory Management* on page 95."

Note: The Cell Broadband Engine processor treats **dst** and **dstt** as no-ops.

dstst

Data Stream Touch for Store (0x7C00 02EC)

dstst
dsttst**rA,rB,STRM**
rA,rB,STRM(T='0')
(T='1')

Form: X

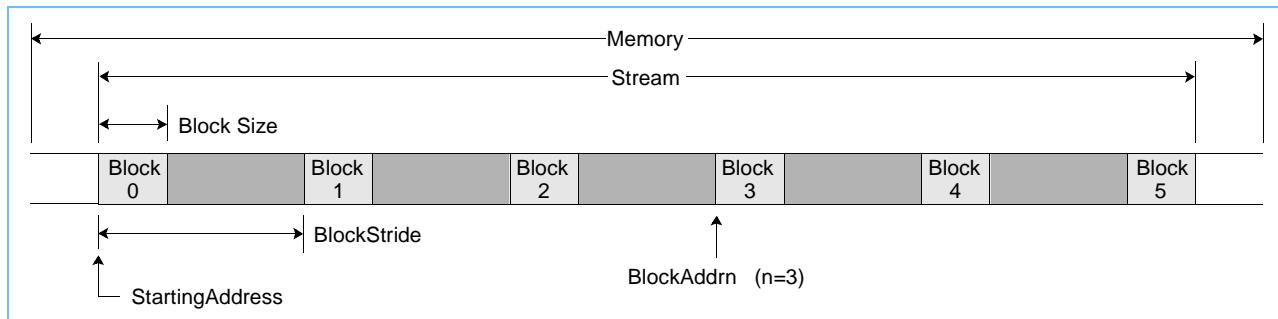
31	T	0 0	STRM	rA	rB	374	0
0	5	6 7 8 9 10 11		15 16	20 21		30 31

$\text{addr}_{0:63} \leftarrow (\text{rA})$
 $\text{DataStreamPrefetchControl} \leftarrow \text{"start"} \parallel T \parallel \text{static} \parallel (\text{rB}) \parallel \text{addr}$

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 registers **rA** and **rB** with the stream ID specified by **STRM**. The instruction defines a data stream **STRM** as starting at an “Effective Address” (**rA**) and having “Count” units of “Size” bytes separated by “Stride” bytes (as specified in **rB**). 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'**).

Figure 6-4. **dstst**—Data Stream Touch for Store



The **dstst** instruction performs the following functions:

- Defines the characteristics of a data stream **STRM** by the contents of **rA** and **rB**
- 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 **rA** 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**.

Vector/SIMD Multimedia Extension Technology

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

- Effective Address: **rA**, where **rA** ≠ '0'
- Block Size: **rB[35-39]** if **rB[35-39]** ≠ '0'; otherwise 32
- Block Count: **rB[40-47]** if **rB[40-47]** ≠ '0'; otherwise 256
- Block Stride: **rB[48-63]** (signed integer) if **rB[48-63]** ≠ '0'; otherwise 32768

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

63

Other registers altered:

- None

Simplified mnemonics:

dstst	rA,rB,STRM	equivalent to	dstst	rA,rB,STRM,0
dststt	rA,rB,STRM	equivalent to	dstst	rA,rB,STRM,1

For more information about the **dstst** instruction, see *Section 5 Cache, Exceptions, and Memory Management* on page 95.”

Note: The Cell Broadband Engine processor treats **dstst** and **dststt** as no-ops.

lvebx

Load Vector Element Byte Indexed (0x7C00 000E)

lvebx**vD,rA,rB**

Form: X

31	vD	rA	rB	7	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else           b ← (rA)
EA ← b + (rB)
eb ← EA60:63
vD ← undefined
(vD)ebx8 : (ebx8)+7 ← MEM(EA, 1)

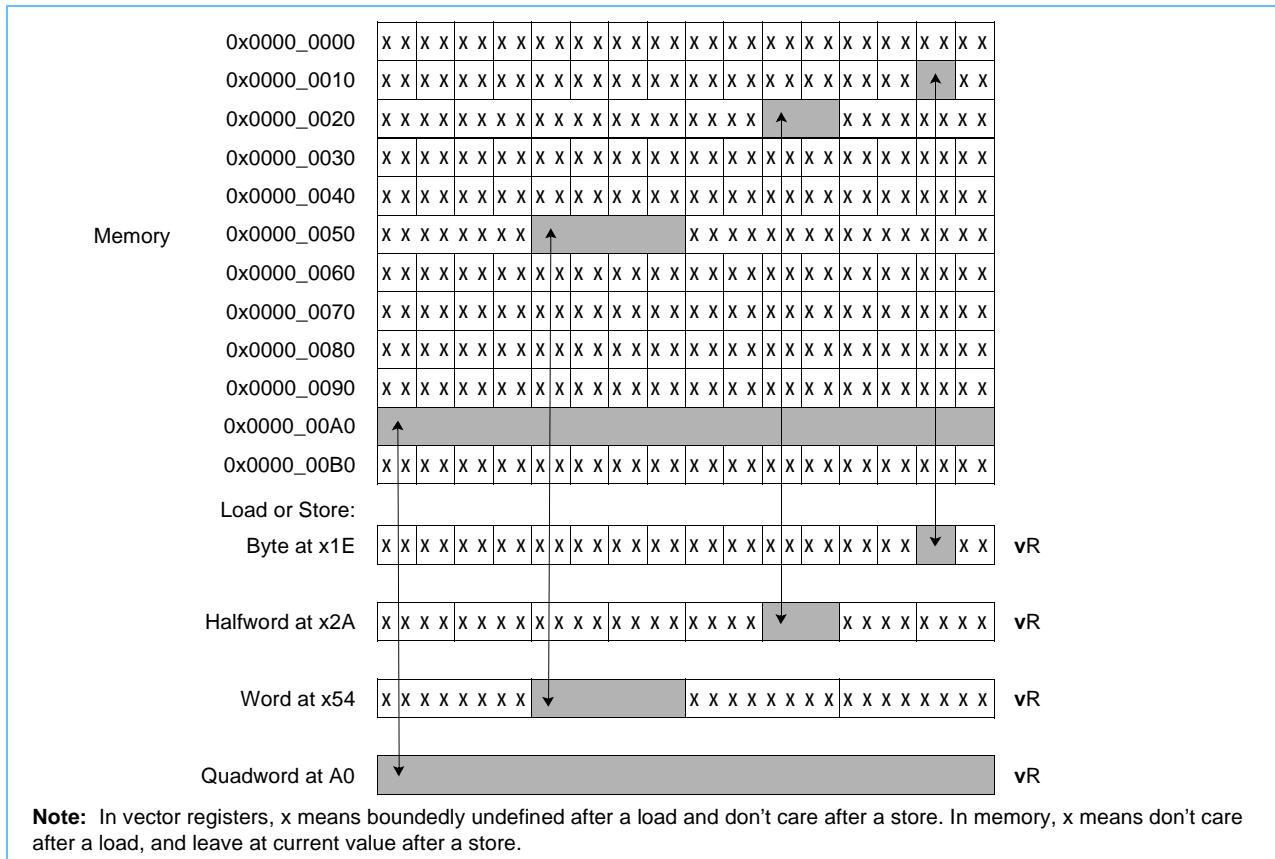
```

Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB. Let m be the value of bits [60–63] of EA, where m is the byte offset of the byte in its aligned quadword in memory.

The byte addressed by EA is loaded into byte m of register vD. The remaining bytes in vD are set to undefined values.

Other registers altered:

- None

Vector/SIMD Multimedia Extension Technology*Figure 6-5. Effects of Example Load/Store Instructions*

Ivehx

Load Vector Element Halfword Indexed (0x7C00 004E)

Ivehx**vD,rA,rB**

Form: X

31	vD	rA	rB	39	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else
    b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA60:63
vD ← undefined
(vD) (ebx8):(ebx8) + 15 ← MEM(EA, 2)

```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value ‘0’ if rA is equal to ‘0’, and the contents of register rB with the value 0xFFFF_FFFF_FFFF_FFFE. Let m be the value of bits [60-62] of EA, where m is the halfword offset of the halfword in its aligned quadword in memory.

The halfword addressed by EA is loaded into halfword m of register vD. The remaining halfwords in register vD are set to undefined values. *Figure 6-5* shows this instruction.

Other registers altered:

- None

lviewx

Load Vector Element Word Indexed (0x7C00 008E)

lviewx**vD,rA,rB****lviewx**

Form: X

31			vD			rA			rB			71			0
0			5 6			10 11			15 16			20 21			30 31

```

if rA=0 then b ← 0
else      b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFFC
eb ← EA60:63
vD ← undefined
(vD)ebx8:(ebx8)+31] ← MEM (EA, 4)

```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB with the value 0xFFFF_FFFF_FFFF_FFFC. Let m be the value of bits 60:61 of EA, where m is the word offset of the word in its aligned quadword in memory.

The word addressed by EA is loaded into word m of register vD. The remaining words in register vD are set to undefined values. *Figure 6-5* shows this instruction.

Other registers altered:

- None

lvIx**lvIx**

Load Vector Left Indexed (0x7C00 040E)

lvIx**vD,rA,rB**

Form X

31		vD	rA	rB	519	0
0	5 6	10 11	15 16	20 21	30 31	

```

if rA=0 then base ← 0
else   base ← (rA)
EA ← (base + (rB))
eb ← EA60:63
(vD) ← MEM(EA,16-eb) || (ebx8) (0)

```

Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.

Let eb be the value of bits [60-63] of EA.

The 16-eb bytes in memory addressed by EA are loaded into the left-most 16-eb byte elements of register vD. The right-most eb byte elements of register vD are set to 0.

Other registers altered:

- None

Note: **lvIx** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

Vector/SIMD Multimedia Extension Technology

|v|x|

Load Vector Left Indexed Last (0x7C00 060E)

|v|x|**vD,rA,rB****|v|x|**

Form: X

31	vD	rA	rB	775	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then base ← 0
else    base ← (rA)
EA ← (base + (rB))
eb ← EA60:63
(vD) ← MEM(EA,16-eb) || (ebx8) (0)
mark_cache_block_as_not_likely_to_be_needed_again_anytime_soon(EA)

```

Let the effective address EA be the sum of the contents of register **rA**, or the value '0' if **rA** is equal to '0', and the contents of register **rB**.

Let eb be the value of bits [60-63] of EA.

The (16-eb) bytes in memory addressed by EA are loaded into the left-most (16-eb) byte elements of register **vD**. The right-most eb byte elements of register **vD** are set to '0'.

This instruction provides a hint that the quadword in memory accessed by EA will probably not be needed again by the program in the near future.

Other registers altered:

- None

Note: **|v|x|** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

lvrx**lvrx**

Load Vector Right Indexed (0x7C00 044E)

lvrx**vD,rA,rB**

Form: X

31	vD	rA	rB	551	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then base ← 0
else    base ← (rA)
EA ← (base + (rB))
eb ← EA[60:63]
(vD) ← (16-eb×8) (0) || MEM(EA-eb,eb)

```

Let the effective address EA be the sum of the contents of register **rA**, or the value ‘0’ if **rA** is equal to ‘0’, and the contents of register **rB**.

Let eb be the value of bits [60-63] of EA.

If eb is not equal to ‘0’ (for example, EA is not quadword-aligned), then eb bytes in memory addressed by (EA-eb) are loaded into the right-most eb byte elements of **vD** and the left-most (16-eb) byte elements of register **vD** are set to ‘0’.

If eb is equal to ‘0’ (for example, EA is quadword-aligned), then the contents of register **vD** are set to ‘0’.

Other registers altered:

- None

Note: **lvrx** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

Vector/SIMD Multimedia Extension Technology**Ivrxi**

Load Vector Right Indexed Last (0x7C00 064E)

Ivrxi**vD,rA,rB**

Form: X

31	vD	rA	rB	807	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then base ← 0
else    base ← (rA)
EA ← (base + (rB))
eb ← EA60:63
(vD) ← (16-eb×8)(0) || MEM(EA-eb,eb)
mark_cache_block_as_not_likely_to_be_needed_again_anytime_soon(EA)

```

Let the effective address EA be the sum of the contents of register **rA**, or the value '0' if **rA** is equal to '0', and the contents of register **rB**.

Let eb be the value of bits [60-63] of EA.

If eb is not equal to '0' (for example, EA is not quadword-aligned), then eb bytes in memory addressed by (EA-eb) are loaded into the right-most eb byte elements of **vD** and the left-most (16-eb) byte elements of register **vD** are set to '0'.

If eb is equal to '0' (for example, EA is quadword-aligned), then the contents of register **vD** are set to '0'.

This instruction provides a hint that the quadword in memory accessed by EA will probably not be needed again by the program in the near future.

Other registers altered:

- None

Note: **Ivrxi** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

lvsl

Load Vector for Shift Left (0x7C00 000C)

lvsl**vD,rA,rB****lvsl**

Form: X

31	vD	rA	rB	6	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA = 0 then b ← 0
else b ← (rA)
addr[0:63] ← b + (rB)
sh ← addr[60:63]

if sh = 0x0 then (vD) ← 0x000102030405060708090A0B0C0D0E0F
if sh = 0x1 then (vD) ← 0x0102030405060708090A0B0C0D0E0F10
if sh = 0x2 then (vD) ← 0x02030405060708090A0B0C0D0E0F1011
if sh = 0x3 then (vD) ← 0x030405060708090A0B0C0D0E0F101112
if sh = 0x4 then (vD) ← 0x0405060708090A0B0C0D0E0F10111213
if sh = 0x5 then (vD) ← 0x05060708090A0B0C0D0E0F1011121314
if sh = 0x6 then (vD) ← 0x060708090A0B0C0D0E0F101112131415
if sh = 0x7 then (vD) ← 0x0708090A0B0C0D0E0F10111213141516
if sh = 0x8 then (vD) ← 0x08090A0B0C0D0E0F1011121314151617
if sh = 0x9 then (vD) ← 0x090A0B0C0D0E0F101112131415161718
if sh = 0xA then (vD) ← 0x0A0B0C0D0E0F10111213141516171819
if sh = 0xB then (vD) ← 0x0B0C0D0E0F101112131415161718191A
if sh = 0xC then (vD) ← 0x0C0D0E0F101112131415161718191A1B
if sh = 0xD then (vD) ← 0x0D0E0F101112131415161718191A1B1C
if sh = 0xE then (vD) ← 0x0E0F101112131415161718191A1B1C1D
if sh = 0xF then (vD) ← 0x0F101112131415161718191A1B1C1D1E

```

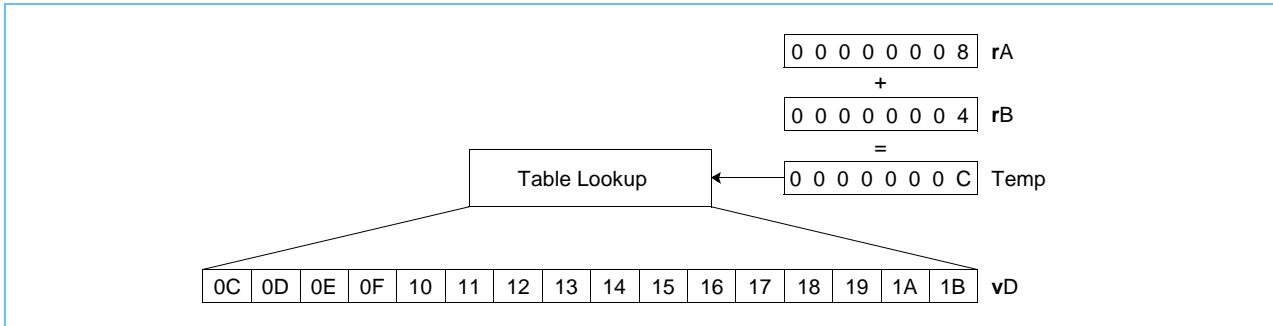
Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB. Let sh be the value of bits [60-63] of the effective address (EA).

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

Other registers altered:

- None

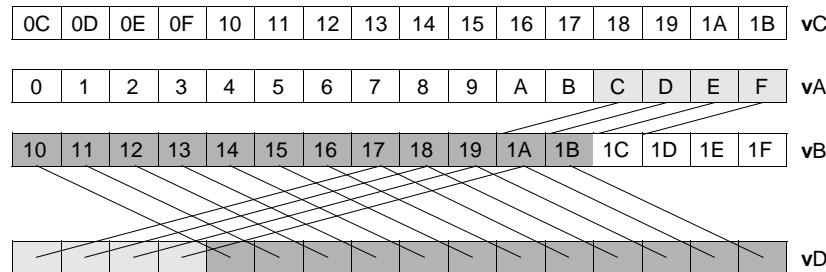
Figure 6-6. Load Vector for Shift Left



Vector/SIMD Multimedia Extension Technology

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

*Figure 6-7. Instruction **vperm** Used in Aligning Data*



See the description of the **lvsR** instruction for suggested uses of the **lvsI** instruction.

Ivsr

Load Vector for Shift Right (0x7C00 004C)

Ivsr**Ivsr****vD,rA,rB**

Form: X

31	5	6	vD	10	11	rA	15	16	rB	20	21	38	30	31	0
0															

```

if rA = 0 then b ← 0
else           b ← (rA)
EA ← b + (rB)
sh ← EA[60:63]
if sh=0x0 then (vD) ← 0x101112131415161718191A1B1C1D1E1F
if sh=0x1 then (vD) ← 0x0F101112131415161718191A1B1C1D1E
if sh=0x2 then (vD) ← 0x0EOF101112131415161718191A1B1C1D
if sh=0x3 then (vD) ← 0x0D0EOF101112131415161718191A1B1C
if sh=0x4 then (vD) ← 0x0COD0EOF101112131415161718191A1B
if sh=0x5 then (vD) ← 0x0BC0D0EOF101112131415161718191A
if sh=0x6 then (vD) ← 0x0AB0C0D0EOF10111213141516171819
if sh=0x7 then (vD) ← 0x090A0B0C0D0EOF101112131415161718
if sh=0x8 then (vD) ← 0x08090A0B0C0D0EOF1011121314151617
if sh=0x9 then (vD) ← 0x0708090A0B0C0D0EOF10111213141516
if sh=0xA then (vD) ← 0x060708090A0B0C0D0EOF101112131415
if sh=0xB then (vD) ← 0x05060708090A0B0C0D0EOF1011121314
if sh=0xC then (vD) ← 0x0405060708090A0B0C0D0EOF10111213
if sh=0xD then (vD) ← 0x030405060708090A0B0C0D0EOF101112
if sh=0xE then (vD) ← 0x02030405060708090A0B0C0D0EOF10111
if sh=0xF then (vD) ← 0x0102030405060708090A0B0C0D0EOF10

```

Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB. Let sh be the value of bits [60-63] of EA.

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

Note: **IvsI** and **Ivsr** 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 **vA** and **vB** specified by the **vperm**. The control vector created by **IvsI** 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 **vA** and **vB** for **vperm**. For shifting left, the **vB** register for **vperm** should contain all zeros and **vA** should contain the value to be shifted, and vice versa for shifting right. *Figure 6-6* on page 127 shows a similar instruction only in that figure the shift is to the left.

Other registers altered:

- None

Vector/SIMD Multimedia Extension Technology**lvx**

Load Vector Indexed (0x7C00 00CE)

lvx**lvx****vD,rA,rB**

Form: X

31	vD	rA	rB	103	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else      b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFF0
vD ← MEM(EA,16)

```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB with the value 0xFFFF_FFFF_FFFF_FFF0.

The quadword in memory addressed by EA is loaded into register vD.

Other registers altered:

- None

lvxl**lvxl**

Load Vector Indexed Last (0x7C00 02CE)

lvxl**vD,rA,rB**

Form: X

31	vD	rA	rB	359	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else          b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFF0
vD ← MEM(EA,16)
mark_cache_block_as_not_likely_to_be_needed_again_anytime_soon(EA)

```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value 0 if rA is equal to 0, and the contents of register rB with the value 0xFFFF_FFFF_FFFF_FFF0.

The quadword addressed by EA is loaded into register vD.

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 Last (**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.

Other registers altered:

- None

mfvscr

Move from Vector Status and Control Register (0x1000 0604)

mfvscr**vD****mfvscr**

Form: VX

04	vD	0 0 0 0	0 0 0 0	1540
0	5 6	10 11	15 16	20 21

$$\text{vD} \leftarrow 96(0) \parallel (\text{VSCR})$$

The contents of the Vector Status and Control Register (VSCR) are placed into register vD.

Note: 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 (0x1000 0C44)

mtvscr**vB**

Form: VX

04	0 0 0 0 0	0 0 0 0 0	vB	1604
0	5 6	10 11	15 16	20 21

VSCR \leftarrow (vB)_{96:127}

The contents of register vB are placed into the VSCR.

Other registers altered:

- None

stvebx

Store Vector Element Byte Indexed (0x7C00 010E)

stvebx**vS,rA,rB****stvebx**

Form: X

31	vS	rA	rB	135	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
eb ← EA60:63
then MEM(EA,1) ← (vS)eb × 8:(eb × 8)+7

```

Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB. Let m be the value of bits [60:63] of EA, where m is the byte offset of the byte in its aligned quadword in memory.

Byte m of register vS is stored into the byte in memory addressed by EA. *Figure 6-5* shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvehx

Store Vector Element Halfword Indexed (0x7C00 014E)

stvehx**vS,rA,rB**

Form: X

31	vS	rA	rB	167	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else
    b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA0:63
MEM(EA, 2) ← vS[eb × 8 : (eb × 8) + 15]

```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB with 0xFFFF_FFFF_FFFF_FFFE. Let m be the value of bits [60:62] of EA, where m is the halfword offset of the halfword in its aligned quadword in memory.

Halfword m of register vS is stored into the halfword addressed by EA. *Figure 6-5* shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvewx

Store Vector Element Word Indexed (0x7C00 018E)

stvewx**vS,rA,rB****stvewx**

Form: X

31	5	6	vS	10	11	rA	15	16	rB	20	21	199	0	30	31
0															

```

if rA=0 then b ← 0
else      b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFFC
eb ← EA60:63
MEM(EA, 4) ← (vS)eb × 8 : (eb × 8) + 31

```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB with 0xFFFF_FFFF_FFFF_FFFC. Let m be the value of bits [60:61] of EA, where m is the word offset of the word in its aligned quadword in memory.

Word m of register vS 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

stvlx

Store Vector Left Indexed (0x7C00 050E)

stvlx**vS,rA,rB****stvlx**

Form X

31	vS	rA	rB	647	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
eb ← EA60:63
MEM(EA, 16-eb) ← (vS)0:(127-eb×8)

```

Let the effective address EA be the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB.

Let eb be the value of bits [60-63] of EA.

The contents of the left-most 16-eb byte elements of register vS are stored into the 16-eb bytes of memory addressed by EA.

Other registers altered:

- None

Note: **stvlx** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

stvlxl**stvlxl**

Store Vector Left Indexed Last (0x7C00 070E)

stvlxl**vS,rA,rB**

Form X

31	vS	rA	rB	903	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
eb ← EA60:63
MEM(EA,16-eb) ← (vS)0:(127-eb×8)
mark_cache_block_as_not_likely_to_be_needed_again_anytime_soon(EA)

```

Let the effective address EA be the sum of the contents of register rA, or the value 0 if rA is equal to 0, and the contents of register rB.

Let eb be the value of bits [60-63] of EA.

The contents of the left-most 16-eb byte elements of register vS are stored into the 16-eb bytes of memory addressed by EA.

This instruction provides a hint that the quadword in memory accessed by EA will probably not be needed again by the program in the near future.

Other registers altered:

- None

Note: **stvlxl** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

stvrx

Store Vector Right Indexed (0x7C00 054E)

stvrx**vS,rA,rB**

Form X

31	vS	rA	rB	679	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else          b ← (rA)
EA ← b + (rB)
eb ← EA60:63
if eb!=0 then do
    MEM(EA-eb,eb) ← (vS)(128-eb×8):127

```

Let the effective address EA be the sum of the contents of register **rA**, or the value 0 if **rA** is equal to 0, and the contents of register **rB**.

Let eb be the value of bits [60-63] of EA.

If eb is not equal to '0' (for example, EA is not quadword-aligned), then the contents of the right-most eb byte elements of register **vS** are stored into the eb bytes of memory addressed by (EA-eb).

If eb is equal to '0' (for example, EA is quadword-aligned), then memory is not altered by this instruction.

Other registers altered:

- None

Note: **stvrx** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

stvrxl**stvrxl**

Store Vector Right Indexed Last (0x7C00 074E)

stvrxl**vS,rA,rB**

Form X

31	vS	rA	rB	935	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else b ← (rA)
EA ← b + (rB)
eb ← EA60:63
if eb!=0 then do
    MEM(EA-eb, eb) ← (vS)(128-eb×8):127
    mark_cache_block_as_not_likely_to_be_needed_again_anytime_soon(EA)

```

Let the effective address EA be the sum of the contents of register rA, or the value ‘0’ if rA is equal to ‘0’, and the contents of register rB.

Let eb be the value of bits [60-63] of EA.

If eb is not equal to ‘0’ (for example, EA is not quadword-aligned), then the contents of the right-most eb byte elements of register vS are stored into the eb bytes of memory addressed by (EA-eb).

If eb is equal to ‘0’ (for example, EA is quadword-aligned), then memory is not altered by this instruction.

This instruction provides a hint that the quadword in memory accessed by EA will probably not be needed again by the program in the near future.

Other registers altered:

- None

Note: **stvrxl** is supported only on the Cell Broadband Engine processor and is not portable to other implementations of the vector/SIMD multimedia extension technology.

stvx

Store Vector Indexed (0x7C00 01CE)

stvx

vS,rA,rB

stvx

Form: X

31	vS	rA	rB	231	0
0	5 6	10 11	15 16	20 21	30 31

```
if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFF0
MEM(EA,16) ← (vS)
```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value '0' if rA is equal to '0', and the contents of register rB with 0xFFFF_FFFF_FFFF_FFF0.

The contents of register vS are stored into the quadword addressed by EA. *Figure 6-5* shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvxl

Store Vector Indexed Last (0x7C00 03CE)

stvxl**vS,rA,rB****stvxl**

Form: X

31	vS	rA	rB	487	0
0	5 6	10 11	15 16	20 21	30 31

```

if rA=0 then b ← 0
else b ← (rA)
EA ← (b + (rB)) & 0xFFFF_FFFF_FFFF_FFF0
MEM(EA,16) ← (vS)
mark_cache_block_as_not_likely_to_be_needed_again_anytime_soon(EA)

```

Let the effective address EA be the result of ANDing the sum of the contents of register rA, or the value 0 if rA is equal to '0', and the contents of register rB with 0xFFFF_FFFF_FFFF_FFF0.

The contents of register vS are stored into the quadword addressed by EA. The **stvxl** instruction 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 **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

Vector Add Carryout Unsigned Word (0x1000 0180)

vaddcuw**vD,vA,vB**

Form: VX

04	vD	vA	vB	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
    (vD)i:i+31 ← ZeroExtend(temp0, 32)
end

```

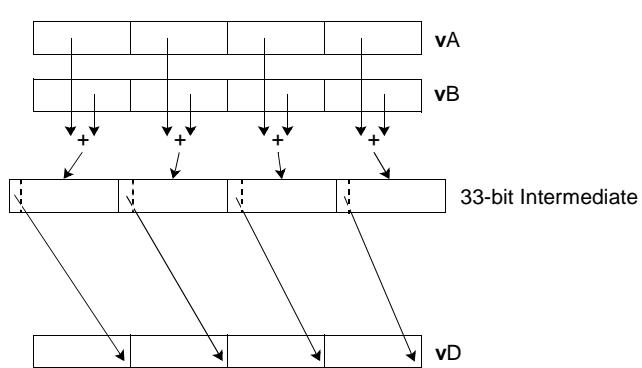
Each unsigned-integer word element i register n **vA** is added to the corresponding unsigned-integer word element in **vB**. The carry out of bit [0] of the 32-bit sum is zero-extended to 32 bits and placed into the corresponding word element of register **vD**.

Other registers altered:

- None

Figure 6-8 shows the usage of the **vaddcuw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

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



vaddfp

Vector Add Floating Point (0x1000 000A)

vaddfp**vD,vA,vB**

Form: VX

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

```
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 element in register **vA** is added to the corresponding single-precision floating-point element in register **vB**. Each intermediate result is rounded and placed in the corresponding single-precision floating-point element in register **vD**.

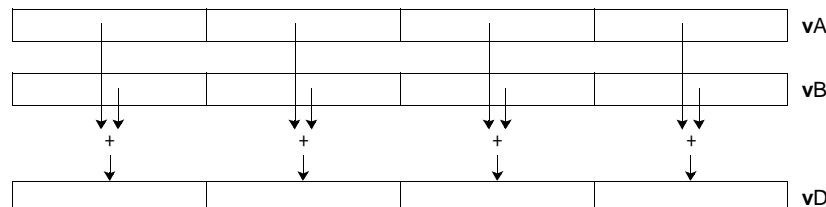
If VSCR[NJ] ='1', every denormalized operand element is truncated to a '0' of the same sign before the operation is carried out, and each denormalized result element truncates to a '0' of the same sign.

Other registers altered:

- None

Figure 6-9 shows the usage of the **vaddfp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

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



vaddssbs

Vector Add Signed Byte Saturate (0x1000 0300)

vaddssbs**vD,vA,vB**

Form: VX

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

```

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
    (vD)i:i+7 ← SItoSISat(temp0:8, 8)
end

```

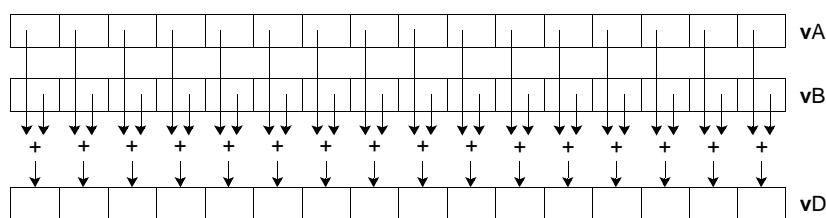
Each signed-integer byte element in register **vA** is added to the corresponding signed-integer byte element in register **vB**. If the intermediate result is greater than 2^7-1 , it saturates to 2^7-1 . If the intermediate result is less than -2^7 , it saturates to -2^7 . If saturation occurs, the saturation (SAT) bit is set. The signed-integer result is placed into the corresponding element of register **vD**.

Other registers altered:

- Vector Status and Control Register (VSCR):
Affected: SAT

Figure 6-10 shows the usage of the **vaddssbs** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

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



vaddshs

Vector Add Signed Halfword Saturate (0x1000 0340)

vaddshs**vD,vA,vB**

Form: VX

04	vD	vA	vB	832
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+15 ← SIToSISat(temp0:16,16)
end

```

Each element of **vaddshs** is a halfword.

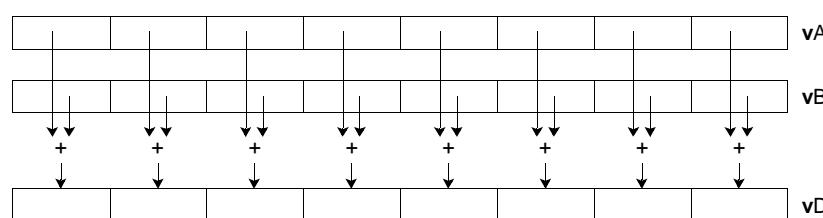
Each signed-integer halfword element in register **vA** is added to the corresponding signed-integer halfword element in register **vB**. If the intermediate result is greater than $2^{15}-1$, it saturates to $2^{15}-1$. If the intermediate result is less than -2^{15} , it saturates to -2^{15} . If saturation occurs, the SAT bit is set. The result is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- Vector Status and Control Register (VSCR):
Affected: SAT

Figure 6-11 shows the usage of the **vaddshs** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

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



vaddsws

Vector Add Signed Word Saturate (0x1000 0380)

vaddsws**vD,vA,vB**

Form: VX

04	vD	vA	vB	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
    (vD)i:i+31 ← SIToSISat(temp0:32, 32)
end

```

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

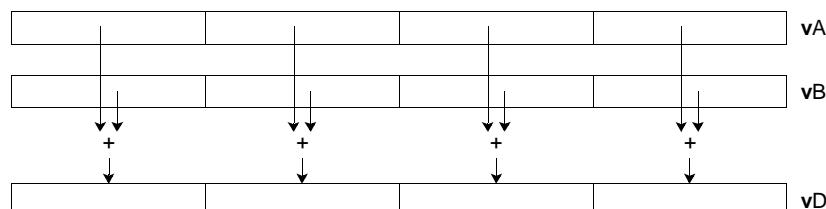
Each signed-integer word element in register **vA** is added to the corresponding signed-integer word element in register **vB**. If the intermediate result is greater than $2^{31}-1$, it saturates to $2^{31}-1$. If the intermediate result is less than -2^{31} , it saturates to -2^{31} . If saturation occurs, the SAT bit is set. The signed-integer result is placed into the corresponding word element of register **vD**.

Other registers altered:

- Vector Status and Control Register (VSCR):
Affected: SAT

Figure 6-12 shows the usage of the **vaddsws** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

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



vaddubm**vaddubm**

Vector Add Unsigned Byte Modulo (0x1000 0000)

vaddubm**vD,vA,vB**

Form: VX

04	vD	vA	vB	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 integer byte element in register **vA** is modulo added to the corresponding integer byte element in register **vB**. The integer result is placed into the corresponding byte element of register **vD**.

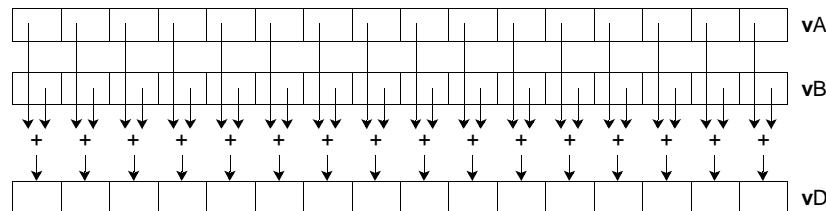
Note: The **vaddubm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-13 shows the **vaddubm** instruction usage. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

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



vaddubs

Vector Add Unsigned Byte Saturate (0x1000 0200)

vaddubs**vD,vA,vB**

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
    (vD)i:i+7 ← UItoUIsat(temp0:8, 8)
end

```

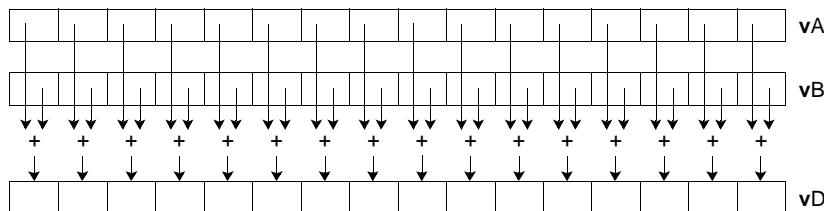
Each unsigned-integer byte element in register **vA** is added to the corresponding unsigned-integer byte element in register **vB**. If the intermediate result is greater than 2^8-1 , it saturates to 2^8-1 . If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding byte element of register **vD**.

Other registers altered:

- Vector Status and Control Register (VSCR):
Affected: SAT

Figure 6-14 shows the usage of the **vaddubs** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

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



vadduhm**vadduhm**

Vector Add Unsigned Halfword Modulo (0x1000 0040)

vadduhm**vD,vA,vB**

Form: VX

04	vD	vA	vB	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 integer halfword element in register **vA** is added to the corresponding integer halfword element in register **vB**. The integer result is placed into the corresponding halfword element of register **vD**.

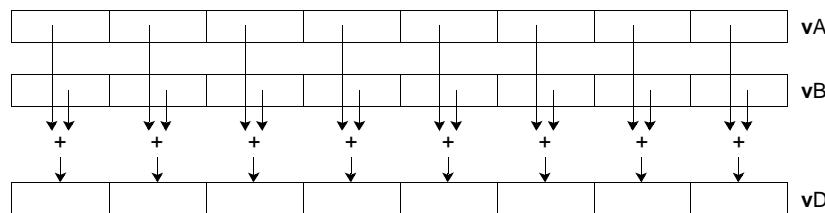
Note: The **vadduhm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-15 shows the usage of the **vadduhm** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

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



vadduhs

Vector Add Unsigned Halfword Saturate (0x1000 0240)

vadduhs

vD,vA,vB

Form: VX

04	vD	vA	vB	576
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+15 ← UItoUIsat(temp0:16,16)
end

```

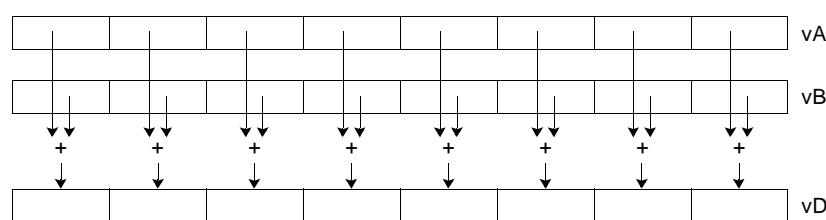
Each unsigned-integer halfword element in register **vA** is added to the corresponding unsigned-integer halfword element in register **vB**. If the intermediate result is greater than $2^{16}-1$, it saturates to $2^{16}-1$. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- Vector Status and Control Register (VSCR):
Affected: SAT

Figure 6-16 shows the usage of the **vadduhs** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** are 16 bits in length.

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



vadduwm

Vector Add Unsigned Word Modulo (0x1000 0080)

vadduwm**vD,vA,vB**

Form: VX

04	vD	vA	vB	128
0	5 6	10 11	15 16	20 21

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

Each integer word element in register **vA** is modulo added to the corresponding integer word element in register **vB**. The integer result is placed into the corresponding word element of register **vD**.

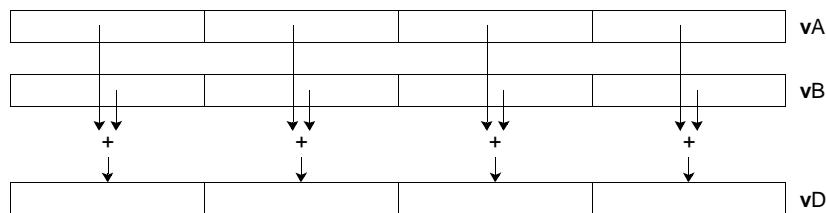
Note: The **vadduwm** instruction can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-17 shows the usage of the **vadduwm** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

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



vadduws

Vector Add Unsigned Word Saturate (0x1000 0280)

vadduws

vD,vA,vB

Form: VX

04	vD	vA	vB	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
    (vD)i:i+31 ← UItoUIsat(temp0:32, 32)
end

```

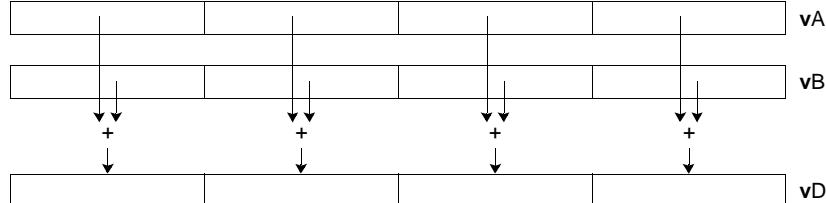
Each unsigned-integer word element in register **vA** is added to the corresponding unsigned-integer word element in register **vB**. If the intermediate result is greater than $2^{32}-1$, it saturates to $2^{32}-1$. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding word element in register **vD**.

Other registers altered:

- Vector Status and Control Register (VSCR):
Affected: SAT

Figure 6-18 shows the usage of the **vadduws** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

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



vand

Vector Logical AND (0x1000 0404)

vand**vD,vA,vB****vand**

Form: VX

04	vD	vA	vB	1028
0	5 6	10 11	15 16	20 21

$$(\text{vD}) \leftarrow (\text{vA}) \& (\text{vB})$$

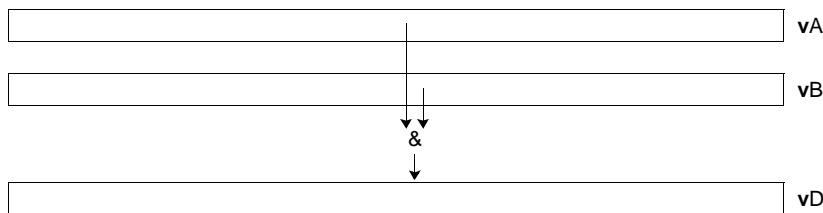
The contents of register **vA** are bitwise ANDed with the contents of register **vB** and the result is placed into register **vD**.

Other registers altered:

- None

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

Figure 6-19. **vand**—Logical Bitwise AND



vandc

Vector Logical AND with Complement (0x1000 0444)

vandc**vD,vA,vB**

Form: VX

04	vD	vA	vB	1092
0	5 6	10 11	15 16	20 21

$$(vD) \leftarrow (vA) \& \neg(vB)$$

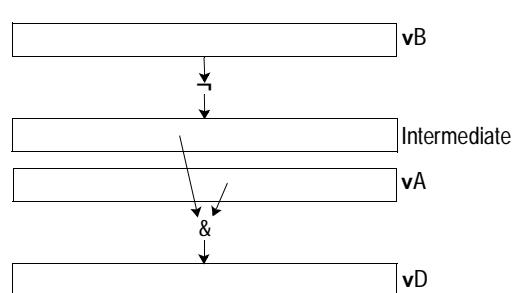
The contents of register vA are ANDed with the one's complement of the contents of register vB and the result is placed into register vD.

Other registers altered:

- None

Figure 6-20 shows usage of the **vandc** instruction.

Figure 6-20. **vandc**—Logical Bitwise AND with Complement



vavgsb

Vector Average Signed Byte (0x1000 0502)

vavgsb**vD,vA,vB****vavgsb**

Form: VX

04	vD	vA	vB	1282
0	5 6	10 11	15 16	20 21

```
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
    (vD)i:i+7 ← temp0:7
end
```

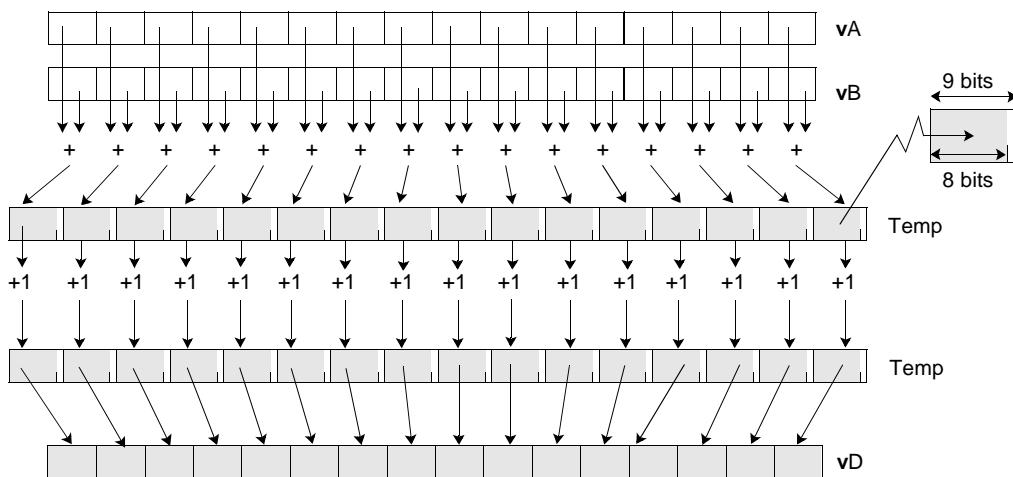
Each signed-integer byte element in register **vA** is added to the corresponding signed-integer byte element in register **vB**, producing a 9-bit signed-integer sum. The sum is incremented by '1'. The high-order 8 bits of the result are placed into the corresponding byte element in register **vD**.

Other registers altered:

- None

Figure 6-21 shows the usage of the **vavgsb** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

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



vavgsh

Vector Average Signed Halfword (0x1000 0542)

vavgsh**vD,vA,vB**

Form: VX

04	vD	vA	vB	1346
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+15 ← temp0:15
end

```

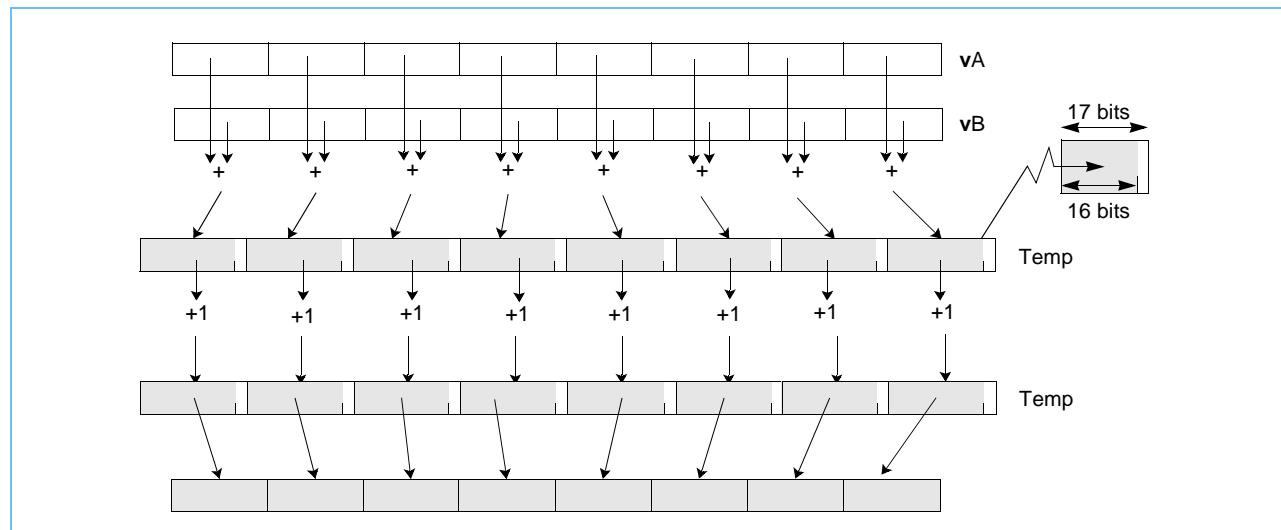
Each signed-integer halfword element in register **vA** is added to the corresponding signed-integer halfword element in register **vB**, producing an 17-bit signed-integer sum. The sum is incremented by '1'. The high-order 16 bits of the result are placed into the corresponding halfword element in register **vD**.

Other registers altered:

- None

Figure 6-22 shows the usage of the **vavgsh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-22. **vavgsh**—Average Eight Signed Integer Elements (16-bits)



vavgsw

Vector Average Signed Word (0x1000 0582)

vavgsw**vD,vA,vB**

Form: VX

04	vD	vA	vB	1410
0	5 6	10 11	15 16	20 21

```
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
    (vD)i:i+31 ← temp0:31
end
```

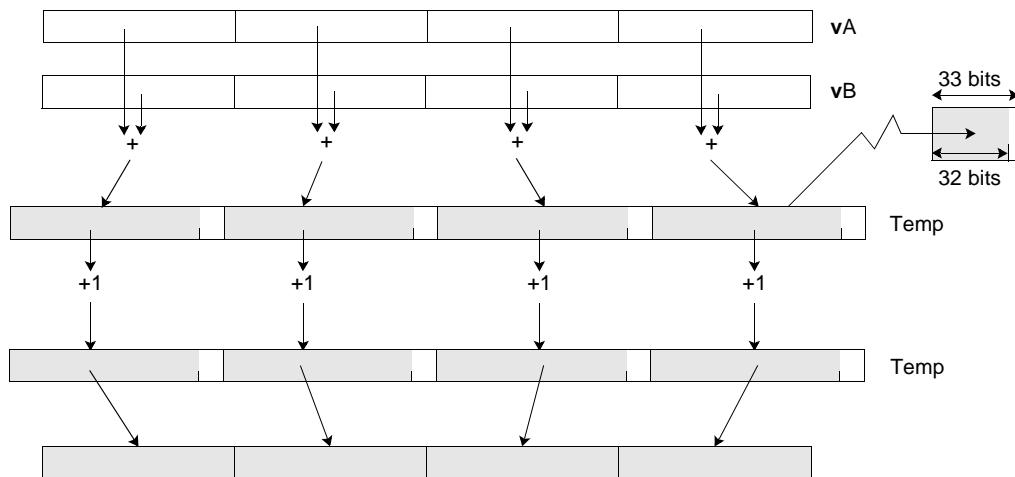
Each signed-integer word element in register **vA** is added to the corresponding signed-integer word element in register **vB**, producing a 33-bit signed-integer sum. The sum is incremented by '1'. The high-order 32 bits of the result are placed into the corresponding word element of register **vD**.

Other registers altered:

- None

Figure 6-23 shows the usage of the **vavgsw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-23. **vavgsw** — Average Four Signed Integer Elements (32-Bit)



vavgub

Vector Average Unsigned Byte (0x1000 0402)

vavgub**vD,vA,vB**

Form: VX

04	vD	vA	vB	1026
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 8
  aop0:8  ← ZeroExtend( (vA)i:i+7, 9)
  bop0:n  ← ZeroExtend( (vB)i:i+71, 9)
  temp0:n  ← aop0:8 +int bop0:8 +int 1
  (vD)i:i+7 ← temp0:7
end

```

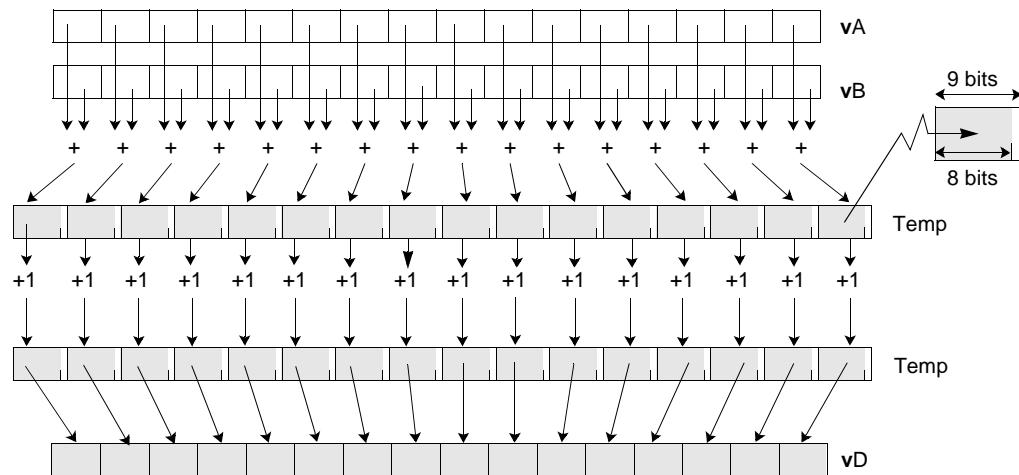
Each unsigned-integer byte element in register **vA** is added to the corresponding unsigned-integer byte element in register **vB**, producing a 9-bit unsigned-integer sum. The sum is incremented by '1'. The high-order 8 bits of the result are placed into the corresponding element of register **vD**.

Other registers altered:

- None

Figure 6-24 shows the usage of the **vavgub** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-24. **vavgub**—Average Sixteen Unsigned Integer Elements (8-bits)



vavguh

Vector Average Unsigned Halfword (0x1000 0442)

vavguh**vD,vA,vB****vavguh**

Form: VX

04	vD	vA	vB	1090
0	5 6	10 11	15 16	20 21

```

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
  (vD)i:i+15 ← temp0:15
end

```

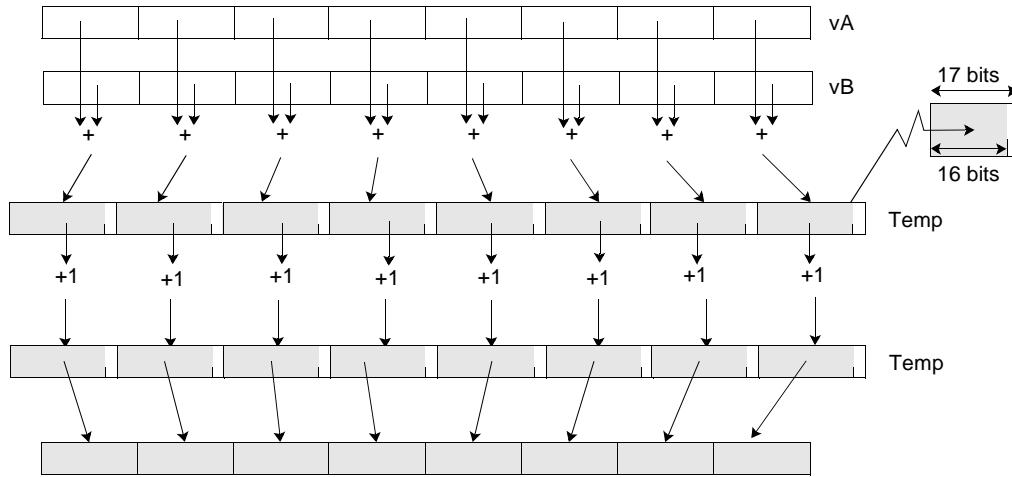
Each unsigned-integer halfword element in register **vA** is added to the corresponding unsigned-integer halfword element in register **vB**, producing a 17-bit unsigned-integer. The sum is incremented by '1'. The high-order 16 bits of the result are placed into the corresponding halfword element of register **vD**.

Other registers altered:

- None

Figure 6-25 shows the usage of the **vavgsh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-25. **vavgsh** — Average Eight Signed Integer Elements (16-Bit)



vavguw

Vector Average Unsigned Word (0x1000 0482)

vavguw**vD,vA,vB**

Form: VX

04	vD	vA	vB	1154
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+31 ← temp0:31
end

```

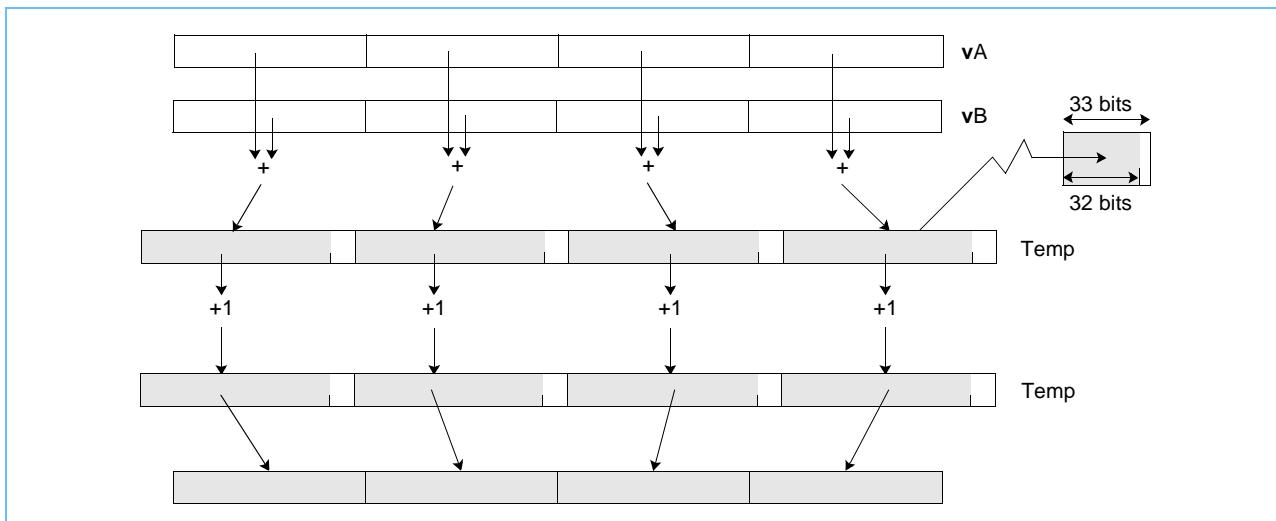
Each unsigned-integer word element in register **vA** is added to the corresponding unsigned-integer word element in register **vB**, producing an 33-bit unsigned-integer sum. The sum is incremented by '1'. The high-order 32 bits of the result are placed into the corresponding word element of register **vD**.

Other registers altered:

- None

Figure 6-26 shows the usage of the **vavguw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** 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 (0x1000 034A)

vcfsx**vD,vB,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
  (vD)i:i+31 ← CnvtSI32ToFP32 ( (vB)i:i+31 ) ÷fp 2UIMM
end
```

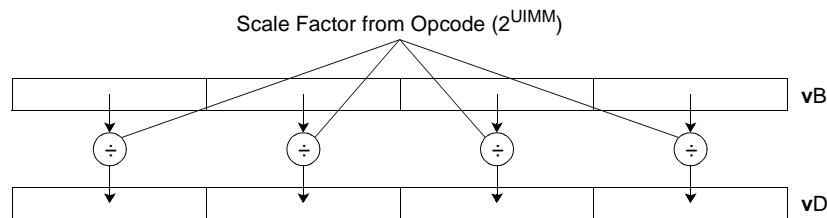
Each signed fixed-point integer word element in register **vB** is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIMM} (**UIMM** = unsigned immediate value) and placed into the corresponding word element in register **vD**.

Other registers altered:

- None

Figure 6-27 shows the usage of the **vcfsx** instruction. Each of the four elements in the vectors **vB** and **vD** is 32 bits in length.

Figure 6-27. vcfsx—Convert Four Signed Integer Elements to Four Floating-Point Elements (32-Bit)



vcfux

Vector Convert from Unsigned Fixed-Point Word (0x1000 030A)

vcfux**vD,vB,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

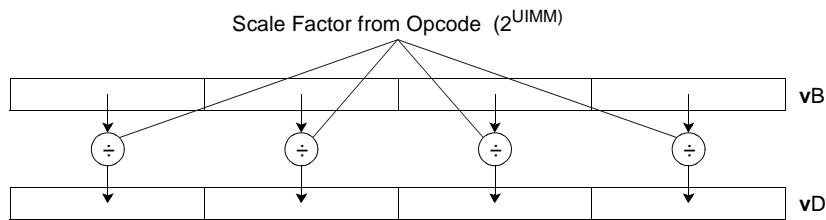
$$(\text{vD})_{i:i+31} \leftarrow \text{CnvtUI32ToFP32}((\text{vB})_{i:i+31}) \div_{fp} 2^{\text{UIMM}}$$
end

Each unsigned fixed-point integer word element in register **vB** is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIMM} and placed into the corresponding word element in register **vD**.

Other registers altered:

- None

Figure 6-28 shows the usage of the **vcfux** instruction. Each of the four elements in the registers **vB** and **vD** is 32 bits in length.

Figure 6-28. **vcfux**—Convert Four Unsigned Integer Elements to Four Floating-Point Elements (32-Bit)

vcmpbfpx

Vector Compare Bounds Floating Point (0x1000 03C6)

vcmpbfpx
vcmpbfpx.**vD,vA,vB**
vD,vA,vB(Rc='0')
(Rc='1')

Form: VXR

04	vD	vA	vB	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)
  (vD)i:i+31 <- ¬le || ¬ge || ³⁰
end
if Rc=1 then do
  ib <- (vD = ¹²⁸₀)
  CR24:27 <- 0b00 || ib || 0b0
end

```

Each single-precision floating-point word element in register **vA** is compared to the corresponding single-precision floating-point element in register **vB**. A 2-bit value is formed that indicates whether the element in register **vA** is within the bounds specified by the element in register **vB**, as follows.

Bit [0] of the 2-bit value is '0' if the element in register **vA** is less than or equal to the element in register **vB**, and is '1' otherwise. Bit [1] of the 2-bit value is '0' if the element in register **vA** is greater than or equal to the negative of the element in register **vB**, and is '1' 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 register **vD** and the remaining bits of the element are set to '0'.

If **Rc=1**, CR Field 6 is set to indicate whether all four elements in register **vA** are within the bounds specified by the corresponding element in register **vB**, as follows:

- CR6 = 0b00 || all_within_bounds || 0

Note: If any single-precision floating-point word element in register **vB** is negative; the corresponding element in register **vA** is out of bounds. Note that if a **vA** or a **vB** 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.

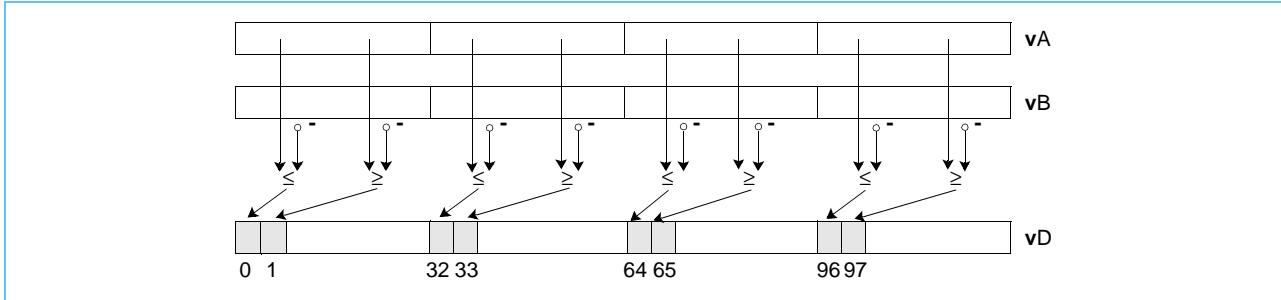
Other registers altered:

- Condition Register (CR6):

Affected: Bit [2]	(if Rc =1)
-------------------	--------------------

Figure 6-29 shows the usage of the **vcmpbfpx** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-29. **vcmpbfp**—Compare Bounds of Four Floating-Point Elements (32-Bit)



vcmpeqfp

Vector Compare Equal-to-Floating Point (0x1000 00C6)

vcmpeqfp**vD,vA,vB**(Rc = '0')
(Rc = '1')

Form: VXR

04	vD	vA	vB	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 (vD)i:i+31 ← 0xFFFF_FFFF
    else (vD)i: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 register **vA** is compared to the corresponding single-precision floating-point word element in register **vB**. The corresponding word element in **vD** is set to all '1's if the element in register **vA** is equal to the element in register **vB**, and is cleared to all '0's otherwise.

If Rc = '1', CR6 field is set according to all, some, or none of the elements pairs compare equal:

- CR6 = all_equal || 0b0 || none_equal || 0b0

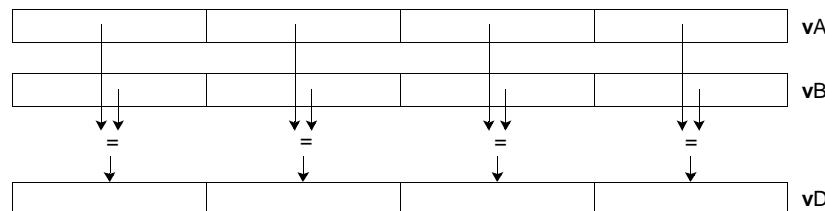
Note: If a **vA** or **vB** element is a NaN, the corresponding result will be 0x0000_0000.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc = '1')

Figure 6-30 shows the usage of the **vcmpeqfp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-30. **vcmpeqfp**—Compare Equal of Four Floating-Point Elements (32-Bit)



vcmpequbx

Vector Compare Equal-to Unsigned Byte (0x1000 0006)

vcmpequb
vcmpequb.**vD,vA,vB**
vD,vA,vB(Rc = '0')
(Rc = '1')

Form: VXR

vcmpequbx

04	vD	vA	vB	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 (vD)i:i+7 ← 81
    else (vD)i:i+7 ← 80
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR[24:27] ← t || 0b0 || f || 0b0
  end

```

Each integer byte element in register **vA** is compared to the corresponding integer byte element in register **vB**. The corresponding byte element in register **vD** is set to all '1's if the element in register **vA** is equal to the element in register **vB**, and is cleared to all '0's otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal:

- CR6 = all_equal || 0b0 || none_equal || 0b0

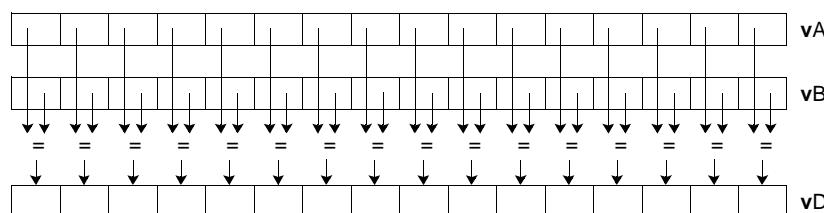
Note: **vcmpequb[.]** can be used for unsigned or signed integers.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0–3] (if Rc = '1')

Figure 6-31 shows the usage of the **vcmpequb** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-31. **vcmpequb**—Compare Equal of Sixteen Integer Elements (8-bits)



vcmpequh

Vector Compare Equal-to Unsigned Halfword (0x1000 0046)

vcmpequh**vD,vA,vB**(Rc = '0')
(Rc = '1')

Form: VXR

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 integer halfword element in **vA** is compared to the corresponding integer halfword element in **vB**. The corresponding halfword element in **vD** is set to all '1's if the element in **vA** is equal to the element in **vB**, and is cleared to all '0's otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal:

- CR6 = all_equal || 0b0 || none_equal || 0b0.

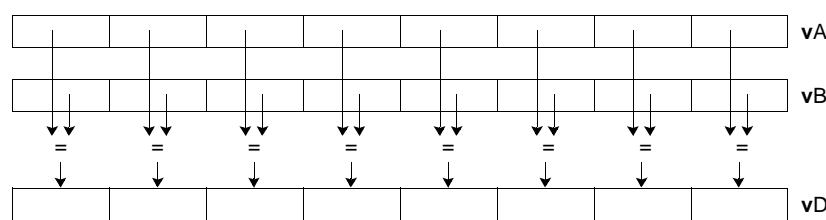
Note: **vcmpequh[.]** can be used for unsigned or signed integers.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0–3] (if Rc = '1')

Figure 6-32 shows the usage of the **vcmpequh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-32. **vcmpequh**—Compare Equal of Eight Integer Elements (16-Bit)



vcmpequwx

Vector Compare Equal-to Unsigned Word (0x1000 0086)

vcmpequw
vcmpequw.**vD,vA,vB**
vD,vA,vB(Rc = '0')
(Rc = '1')

Form: VXR

04	vD	vA	vB	Rc	134
0	5 6	10 11	15 16	20 21 22	31

```

do i=0 to 127 by 32
  if (vA)i:i+31 =int (vB)i:i+31
    then (vD)i:i+31 ← n1
    else (vD)i:i+31 ← n0
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR[24:27] ← t || 0b0 || f || 0b0
  end

```

Each integer word element in register **vA** is compared to the corresponding integer word element in register **vB**. The corresponding element in register **vD** is set to all '1's if the element in register **vA** is equal to the element in register **vB**, and is cleared to all '0's otherwise.

The CR6 is set according to whether all, some, or none of the elements compare equal:

- CR6 = all_equal || 0b0 || none_equal || 0b0

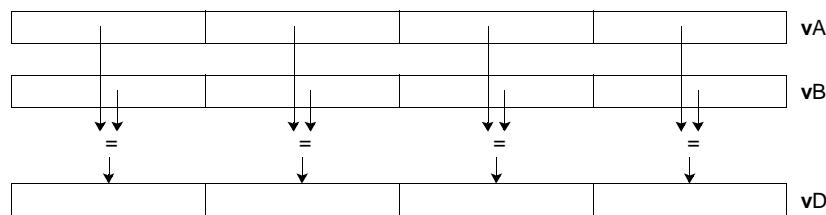
Note: **vcmpequw[.]** can be used for unsigned or signed integers.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc = '1')

Figure 6-33 shows the usage of the **vcmpequw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-33. **vcmpequw**—Compare Equal of Four Integer Elements (32-Bit)



vcmpgefp**vcmpgefp**

Vector Compare Greater-Than-or-Equal-to Floating Point (0x1000 01C6)

vcmpgefp**vD,vA,vB**(Rc='0')
(Rc='1')

Form: VXR

04	vD	vA	vB	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 (vD)i:i+31 ← 0xFFFF_FFFF
    else (vD)i:i+31 ← 0x0000_0000
  end
  if Rc=1 then do
    t ← ((vD) = 11281)
    f ← ((vD) = 11280)
    CR24:27 ← t || 0b0 || f || 0b0
  end

```

Each single-precision floating-point word element in register **vA** is compared to the corresponding single-precision floating-point word element in register **vB**. The corresponding word element in register **vD** is set to all '1's if the element in register **vA** is greater than or equal to the element in register **vB**, and is cleared to all '0's otherwise.

If Rc ='1', CR6 is set as follows:

- CR6 = all_greater_or_equal || 0b0 || none_greater_or_equal || 0b0.

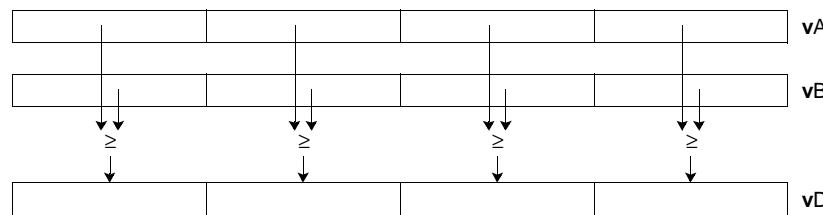
Note: If a **vA** or **vB** element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc ='1')

Figure 6-34 shows the usage of the **vcmpgefp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-34. **vcmpgefp**—Compare Greater-Than-or-Equal of Four Floating-Point Elements (32-Bit)



vcmpgtfp

Vector Compare Greater-Than Floating-Point (0x1000 02C6)

vcmpgtfp**vcmpgtfp.**(Rc = '0')
(Rc = '1')

Form: VXR

04	vD	vA	vB	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 (vD)i:i+31 ← 0xFFFF_FFFF
    else (vD)i: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 register **vA** is compared to the corresponding single-precision floating-point word element in register **vB**. The corresponding word element in register **vD** is set to all '1's if the element in register **vA** is greater than the element in register **vB**, and is cleared to all '0's otherwise.

If Rc ='1', CR6 is set as follows:

- CR6 = all_greater_than || 0b0 || none_greater_than || 0b0.

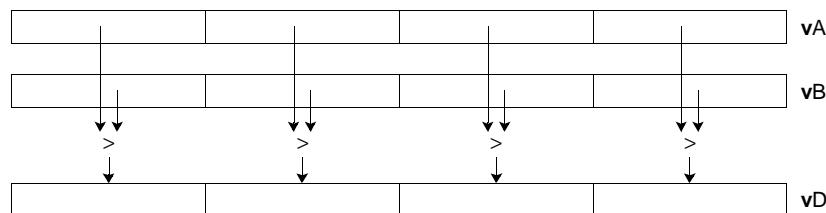
Note: If a **vA** or **vB** element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc ='1')

Figure 6-17 shows the usage of the **vcmpgtfp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-35. **vcmpgtfp**—Compare Greater-Than of Four Floating-Point Elements (32-Bit)



vcmpgtsbx

Vector Compare Greater-Than Signed Byte (0x1000 0306)

vcmpgtsb
vcmpgtsb.**vD,vA,vB**
vD,vA,vB(Rc = '0')
(Rc = '1')

Form: VXR

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 (vD)i:i+7 ← 81
    else (vD)i:i+7 ← 80
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR24:27 ← t || 0b0 || f || 0b0
  end

```

Each signed-integer byte element in register **vA** is compared to the corresponding signed-integer byte element in register **vB**. The corresponding element in **vD** is set to all '1's if the element in **vA** is greater than the element in **vB**, and is cleared to all '0's otherwise.

If Rc ='1', CR6 is set as follows:

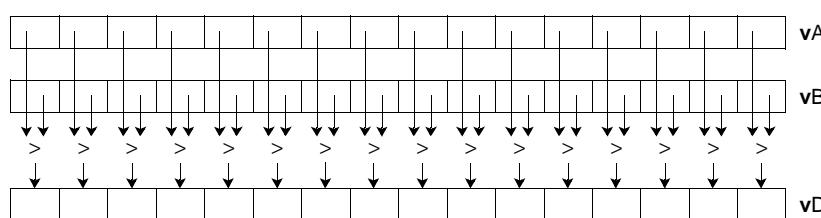
- CR6 = all_greater_than || 0b0 || none_greater_than || 0b0.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc ='1')

Figure 6-36 shows the usage of the **vcmpgtsb** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-36. **vcmpgtsb**—Compare Greater-Than of Sixteen Signed Integer Elements (8-Bit)



vcmpgtshx

Vector Compare Greater-Than Signed Halfword (0x1000 0346)

vcmpgtsh**vD,vA,vB**
vcmpgtsh.(Rc = '0')
(Rc = '1')

Form: VXR

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 (vD)i:i+15 " 161
    else (vD)i:i+15 " 160
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR24:27 ← t || 0b0 || f || 0b0
  end

```

Each signed-integer halfword element in register **vA** is compared to the corresponding signed-integer halfword element in register **vB**. The corresponding halfword element in register **vD** is set to all '1's if the element in **vA** is greater than the element in **vB**, and is cleared to all '0's otherwise.

If Rc = '1', CR6 is set as follows:

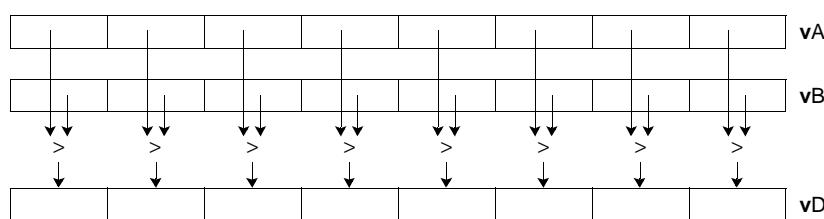
- CR6 = all_greater_than || 0b0 || none_greater_than || 0b0.

Other registers altered:

- Condition Register (CR6):
 - Affected: Bits [0-3] (if Rc = '1')

Figure 6-37 shows the usage of the **vcmpgtsh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-37. **vcmpgtsh**—Compare Greater-Than of Eight Signed Integer Elements (16-Bit)



vcmpgtswx

Vector Compare Greater-Than Signed Word (0x1000 0386)

vcmpgtsw
vcmpgtsw.**vD,vA,vB**
vD,vA,vB(Rc = '0')
(Rc = '1')

Form: VXR

04	vD	vA	vB	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 (vD)i:i+31 ← 321
    else (vD)i:i+31 ← 320
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR24:27 ← t || 0b0 || f || 0b0
  end

```

Each signed-integer word element in register **vA** is compared to the corresponding signed-integer word element in register **vB**. The corresponding word element in register **vD** is set to all '1's if the element in **vA** is greater than the element in **vB**, and is cleared to all '0's otherwise.

If Rc ='1', CR6 is set as follows:

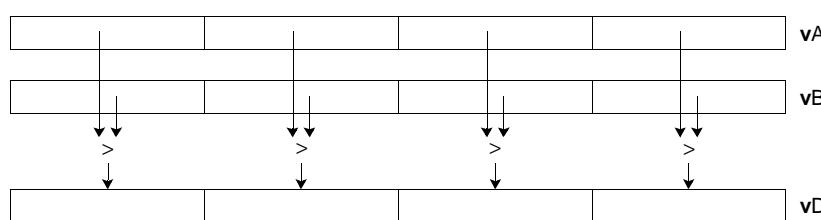
- CR6 = all_greater_than || 0b0 || none_greater_than || 0b0.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc ='1')

Figure 6-38 shows the usage of the **vcmpgtsw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-38. **vcmpgtsw**—Compare Greater-Than of Four Signed Integer Elements (32-Bit)



vcmpgtubx

Vector Compare Greater-Than Unsigned Byte (0x1000 0206)

vcmpgtub
vcmpgtub.**vD,vA,vB**
vD,vA,vB(Rc = '0')
(Rc = '1')

Form: VXR

04	vD	vA	vB	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 ← 81
    else (vD)i:i+7 ← 80
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR[24-27] ← t || 0b0 || f || 0b0
  end

```

Each unsigned-integer byte element in register **vA** is compared to the corresponding unsigned-integer byte element in register **vB**. The corresponding byte element in register **vD** is set to all '1's if the element in **vA** is greater than the element in **vB**, and is cleared to all '0's otherwise.

If Rc ='1', CR6 is set as follows:

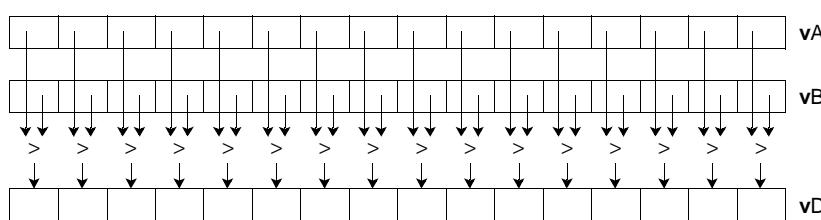
- CR6 = all_greater_than || 0b0 || none_greater_than || 0b0.

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc ='1')

Figure 6-39 shows the usage of the **vcmpgtub** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-39. **vcmpgtub**—Compare Greater-Than of Sixteen Unsigned Integer Elements (8-Bit)



vcmpgtuhx**vcmpgtuhx**

Vector Compare Greater-Than Unsigned Halfword (0x1000 0246)

vcmpgtuh**vD,vA,vB**(Rc = '0')
(Rc = '1')

Form: VXR

vcmpgtuh.

04	vD	vA	vB	Rc	582
0	5 6	10 11	15 16	20 21 22	31

```

do i=0 to 127 by 16
  if (vA)i:i+15 > vBi:i+15
    then (vD)i:i+15 ← 161
    else (vD)i:i+15 ← 160
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR[24-27] ← t || 0b0 || f || 0b0
  end

```

Each unsigned-integer halfword element in register **vA** is compared to the corresponding unsigned-integer halfword element in register **vB**. The corresponding halfword element in register **vD** is set to all '1's if the element in **vA** is greater than the element in **vB**, and is cleared to all '0's otherwise.

If Rc ='1', CR6 is set as follows:

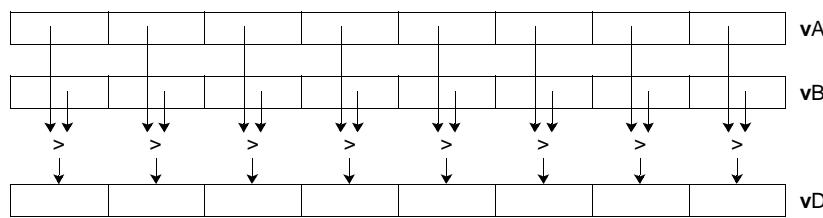
- CR6 = all_greater_than || 0b0 || none_greater_than || 0b0

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc ='1')

Figure 6-40 shows the usage of the **vcmpgtuh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-40. **vcmpgtuh**—Compare Greater-Than of Eight Unsigned Integer Elements (16-Bit)



vcmpgtuw

Vector Compare Greater-Than Unsigned Word (0x1000 0286)

vcmpgtuw**vD,vA,vB**
vcmpgtuw.(Rc = '0')
(Rc = '1')

Form: VXR

04	vD	vA	vB	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 (vD)i:i+31 ← 321
    else (vD)i:i+31 ← 320
  end
  if Rc=1 then do
    t ← ((vD) = 1281)
    f ← ((vD) = 1280)
    CR[24-27] ← t || 0b0 || f || 0b0
  end

```

Each unsigned-integer word element in register **vA** is compared to the corresponding unsigned-integer word element in register **vB**. The corresponding word element in register **vD** is set to all '1's if the element in **vA** is greater than the element in **vB**, and is cleared to all '0's otherwise.

If Rc ='1', CR6 is set as follows:

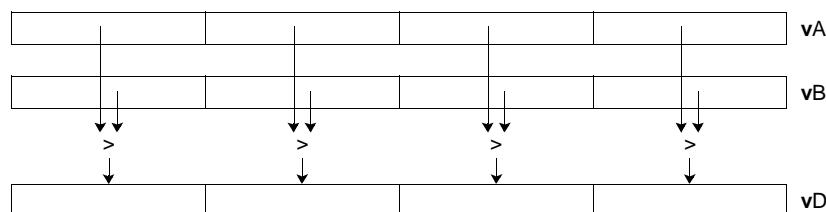
- CR6 = all_greater_than || 0b0 || none_greater_than || 0b0

Other registers altered:

- Condition Register (CR6):
Affected: Bits [0-3] (if Rc ='1')

Figure 6-41 shows the usage of the **vcmpgtuw** instruction. Each of the four elements in the vectors, **vA**, **vB**, and **vD**, 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 (0x1000 03CA)

vctsxs**vD,vB,UIMM**

Form: VX

04	vD	UIMM	vB	970
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  if (vB)i+1:i+8=255 | (vB)i+1:i+8 + UIMM ≤ 254 then
    (vD)i:i+31 ← CnvtFP32ToSI32Sat((vB)i:i+31 * fp 2UIMM)
  else
    do
      if (vB)i=0 then (vD)i:i+31 ← 0x7FFF_FFFF
      else (vD)i:i+31 ← 0x8000_0000
      VSCRSAT ← 1
    end
  end
end

```

Each single-precision word element in register **vB** 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 in register **vD**.

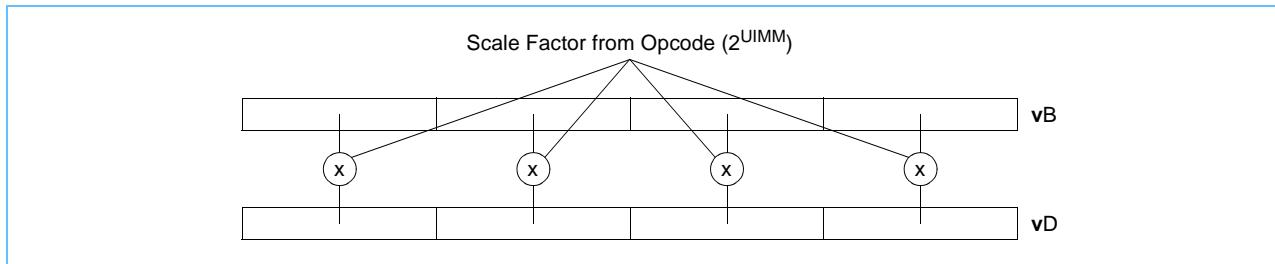
Fixed-point integers used by the vector convert instructions can be interpreted as consisting of 32-UIMM integer bits followed by UIMM fraction bits. The vector convert to fixed-point word instructions support only the rounding mode, Round toward Zero. A single-precision number can be converted to a 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** instruction. Each of the four elements in the vectors **vB** and **vD** 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 (0x1000 038A)

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
    (vD)i:i+31 ← CnvtFP32ToUI32Sat((vB)i:i+31 * fp 2UIM)
  else
    do
      if (vB)i=0 then vDi:i+31 ← 0xFFFF_FFFF
      else (vD)i:i+31 ← 0x0000_0000
      VSCRSAT ← 1
    end
  end
end

```

Each single-precision floating-point word element in **vB** 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 in register **vD**.

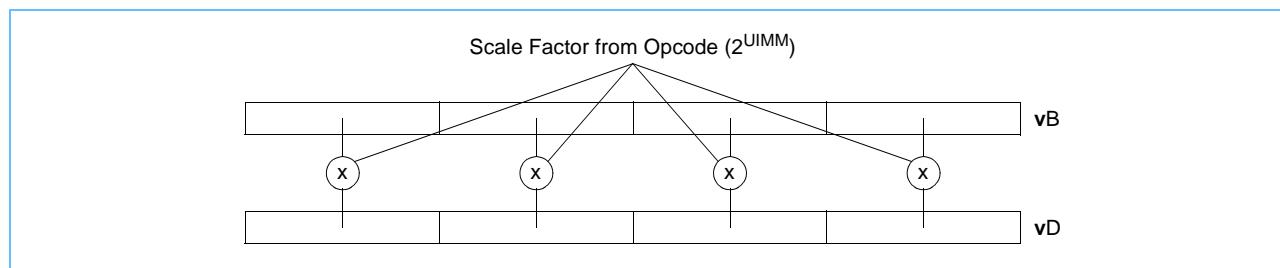
Other registers altered:

- Vector Status and Control Register (VSCR):

Affected: SAT

Figure 6-43 shows the usage of the **vctuxs** instruction. Each of the four elements in the vectors **vB** and **vD** 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 (0x1000 018A)

vexptefp**vD,vB**

Form: VX

04	vD	0 0 0 0 0	vB	394
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  x ← (vB)i:i+31
  (vD)i:i+31 ← 2x
end

```

The single-precision floating-point estimate of 2 raised to the power of each single-precision floating-point element in register vB is placed into the corresponding element in register vD.

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 vB. The most significant 12 bits of the estimate's significant are monotonic. Note that the value placed into the element of vD 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 $+\infty$, the result is exact.

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

Table 6-5. vexptefp Operation with Various Values of Element vB

Value of Element in vB	Result
$-\infty$	+0
-0	+1
+0	+1
$+\infty$	$+\infty$
NaN	QNaN

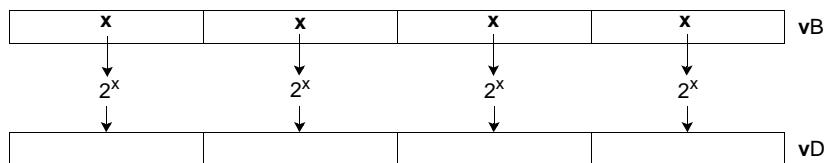
If VSCR[NJ] = '1', every denormalized operand element is truncated to a '0' of the same sign before the operation is carried out, and each denormalized result element truncates to a '0' of the same sign.

Other registers altered:

- None

Figure 6-44 shows the usage of the **vexptefp** instruction. Each of the four elements in the registers vB and vD 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

Vector Log₂ Estimate Floating Point (0x1000 01CA)**vlogefp****vD,vB**

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
  (vD)i:i+31 ← log2(x)
end

```

The single-precision floating-point estimate of the base 2 logarithm of each single-precision floating-point element in register vB is placed into the corresponding element in register vD.

Let x be any single-precision floating-point input value. Unless |x-1| is less than or equal to 0.125 or the single-precision floating-point result of computing the base 2 logarithm of x would be an infinity or a QNaN, the estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than 2^{-5} . Under the same conditions, the estimate has a relative error in precision no greater than one part in 8.

The most significant 12 bits of the estimate's significand are monotonic. The estimate is exact if x=2y, where y is an integer between -149 and +127 inclusive. Otherwise the value placed into the element of register vD may vary between implementations and between different executions on the same implementation.

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

Table 6-6. **vlogefp** with Special Values

Value	Result
-∞	QNaN
less than 0	QNaN
±0	-∞
+∞	+∞
NaN	QNaN

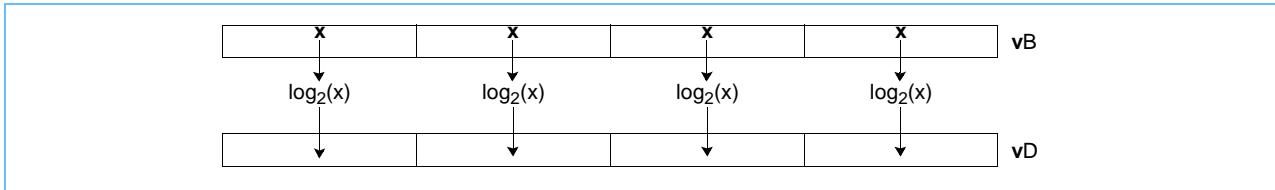
If VSCR[NJ] = '1', every denormalized operand element is truncated to a '0' of the same sign before the operation is carried out, and each denormalized result element truncates to a '0' of the same sign.

Other registers altered:

- None

Figure 6-45 shows the usage of the **vlogefp** instruction. Each of the four elements in the registers vB and vD is 32 bits in length.

Figure 6-45. **vlogefp**— \log_2 Estimate Floating-Point for Four Floating-Point Elements (32-Bit)



vmaddfp

Vector Multiply Add Floating Point (0x1000 002E)

vmaddfp**vD,vA,vC,vB**

Form: VA

04	vD	vA	vB	vC	46
0	5 6	10 11	15 16	20 21	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 register **vA** is multiplied by the corresponding single-precision floating-point word element in register **vC**. The corresponding single-precision floating-point word element in register **vB** is added to the product. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element in register **vD**.

Note that a vector multiply floating-point instruction is not provided. The effect of such an instruction can be obtained by using **vmaddfp** with **vB** 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.)

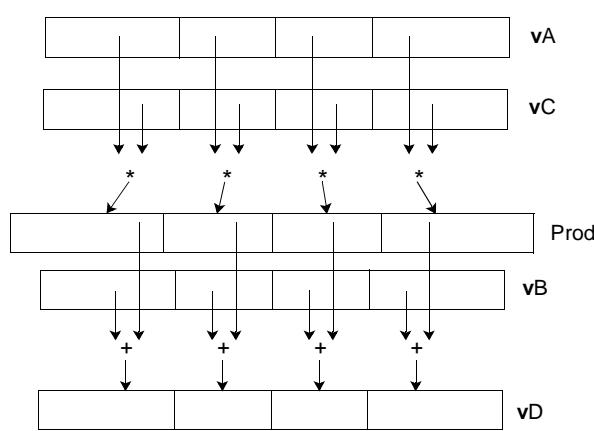
Other registers altered:

- 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** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-46. **vmaddfp**—Multiply-Add Four Floating-Point Elements (32-Bit)



vmaxfp

Vector Maximum Floating Point (0x1000 040A)

vmaxfp**vD,vA,vB**

Form: VX

04	vD	vA	vB	1034
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  if (vA)i:i+31 ≥fp (vB)i:i+31
    then (vD)i:i+31 ← (vA)i:i+31
    else (vD)i:i+31 ← (vB)i:i+31
  end

```

Each single-precision floating-point word element in register **vA** is compared to the corresponding single-precision floating-point word element in register **vB**. The larger of the two single-precision floating-point values is placed into the corresponding word element in register **vD**.

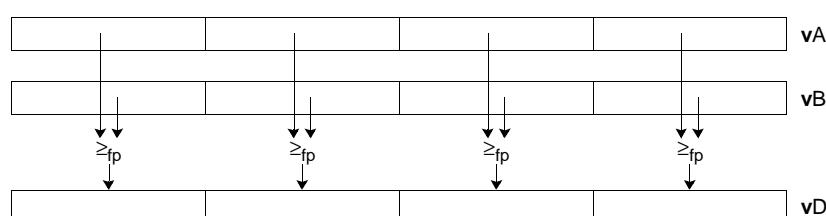
The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

Other registers altered:

- None

Figure 6-47 shows the usage of the **vmaxfp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-47. **vmaxfp**—Maximum of Four Floating-Point Elements (32-Bit)



vmaxsb

Vector Maximum Signed Byte (0x1000 0102)

vmaxsb**vD,vA,vB**

Form: VX

04	vD	vA	vB	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 (vD)i:i+7 ← (vA)i:i+7
    else (vD)i:i+7 ← (vB)i:i+7
  end

```

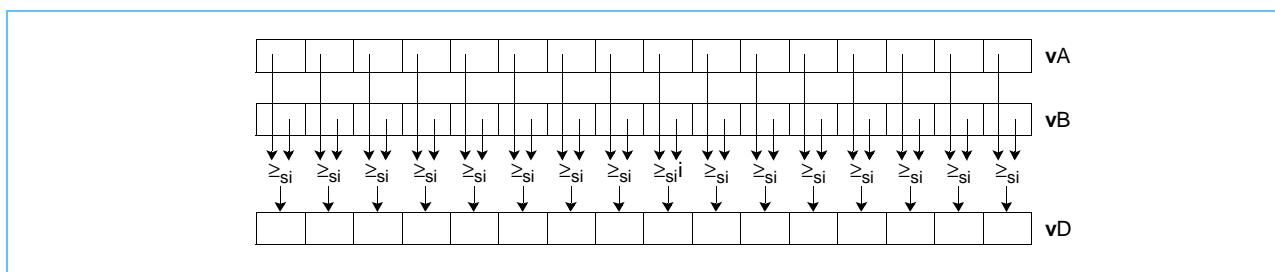
Each signed-integer byte element in register **vA** is compared to the corresponding signed-integer byte element in register **vB**. The larger of the two signed-integer values is placed into the corresponding byte element in register **vD**.

Other registers altered:

- None

Figure 6-48 shows the usage of the **vmaxsb** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-48. **vmaxsb**—Maximum of Sixteen Signed Integer Elements (8-Bit)



vmaxsh

Vector Maximum Signed Halfword (0x1000 0142)

vmaxsh**vD,vA,vB**

Form: VX

04	vD	vA	vB	322
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 16
  if (vA)i:i+7 ≥si (vB)i:i+15
    then (vD)i:i+15 ← (vA)i:i+15
    else (vD)i:i+15 ← (vB)i:i+15
end

```

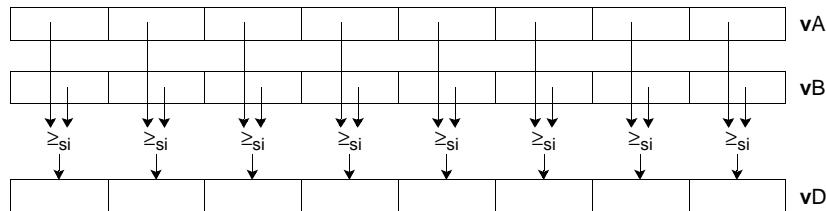
Each signed-integer halfword element in register **vA** is compared to the corresponding signed-integer halfword element in register **vB**. The larger of the two signed-integer values is placed into the corresponding halfword element in register **vD**.

Other registers altered:

- None

Figure 6-49 shows the usage of the **vmaxsh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-49. **vmaxsh**—Maximum of Eight Signed Integer Elements (16-Bit)



vmaxsw

Vector Maximum Signed Word (0x1000 0182)

vmaxsw**vD,vA,vB**

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 (vD)i:i+31 ← (vA)i:i+31
    else (vD)i:i+31 ← (vB)i:i+31
  end

```

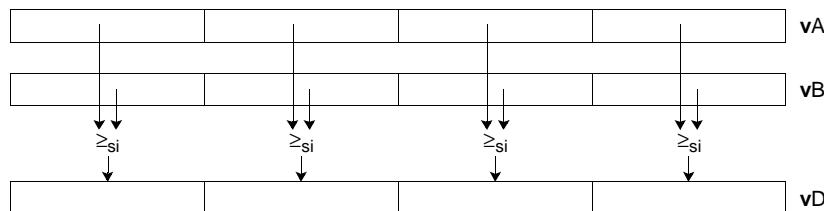
Each signed-integer word element in register **vA** is compared to the corresponding signed-integer word element in register **vB**. The larger of the two signed-integer values is placed into the corresponding word element in register **vD**.

Other registers altered:

- None

Figure 6-50 shows the usage of the **vmaxsw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-50. **vmaxsw**—Maximum of Four Signed Integer Elements (32-Bit)



vmaxub

Vector Maximum Signed Byte (0x1000 0002)

vmaxub**vD,vA,vB**

Form: VX

04	vD	vA	vB	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 (vD)i:i+7 ← (vA)i:i+7
    else (vD)i:i+7 ← (vB)i:i+7
end

```

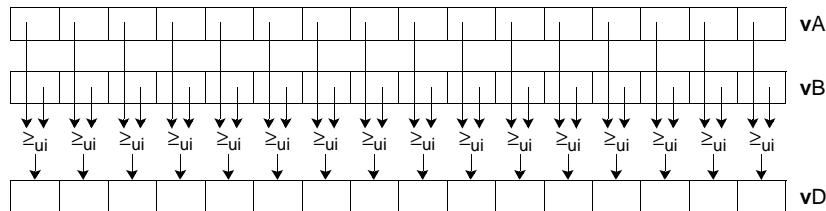
Each unsigned-integer byte element in register **vA** is compared to the corresponding unsigned-integer byte element in register **vB**. The larger of the two unsigned-integer values is placed into the corresponding byte element in register **vD**.

Other registers altered:

- None

Figure 6-51 shows the usage of the **vmaxub** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-51. **vmaxub**—Maximum of Sixteen Unsigned Integer Elements (8-Bit)



vmaxuh

Vector Maximum Unsigned Halfword (0x1000 0042)

vmaxuh**vD,vA,vB**

Form: VX

04	vD	vA	vB	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 (vD)i:i+15 ← (vA)i:i+15
    else (vD)i:i+15 ← (vB)i:i+15
  end

```

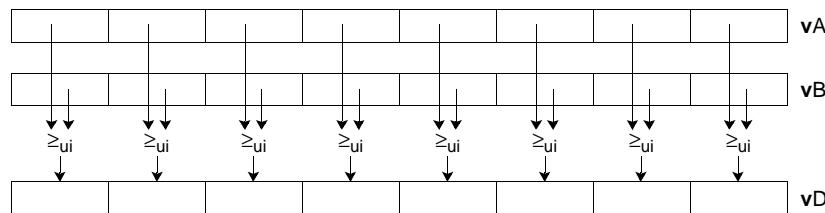
Each unsigned-integer halfword element in register **vA** is compared to the corresponding unsigned-integer halfword element in register **vB**. The larger of the two unsigned-integer values is placed into the corresponding halfword element in register **vD**.

Other registers altered:

- None

Figure 6-52 shows the usage of the **vmaxuh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-52. **vmaxuh**—Maximum of Eight Unsigned Integer Elements (16-Bit)



vmaxuw

Vector Maximum Unsigned Word (0x1000 0082)

vmaxuw**vD,vA,vB**

Form: VX

04	vD		vA		vB		130	
0	5	6	10	11	15	16	20	21

```

do i=0 to 127 by 32
  if (vA)i:i+31 ≥ui (vB)i:i+31
    then (vD)i:i+31 ← (vA)i:i+31
    else (vD)i:i+31 ← (vB)i:i+31
  end

```

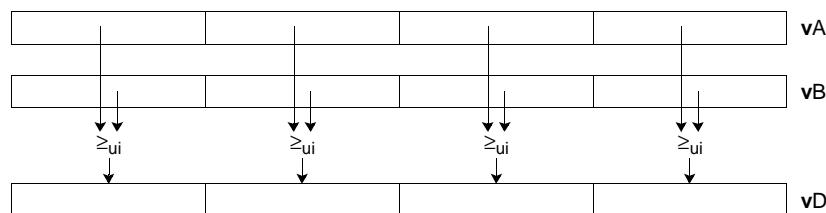
Each unsigned-integer word element in register **vA** is compared to the corresponding unsigned-integer word element in register **vB**. The larger of the two unsigned-integer values is placed into the corresponding word element in register **vD**.

Other registers altered:

- None

Figure 6-53 shows the usage of the **vmaxuw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-53. **vmaxuw**—Maximum of Four Unsigned Integer Elements (32-Bit)



vmhaddshs

Vector Multiply High and Add Signed Halfword Saturate (0x1000 0020)

vmhaddshs**vD,vA,vB,vC**

Form: VA

04	vD	vA	vB	vC	32
0	5 6	10 11	15 16	20 21	25 26

```
do i=0 to 127 by 16
    prod0:31 ← (vA)i:i+15 * si (vB)i:i+15
    temp0:16 ← prod0:16 +int SignExtend((vC)i:i+15,17)
    (vD)i:i+15 ← SItoSISat(temp0:16,16)
end
```

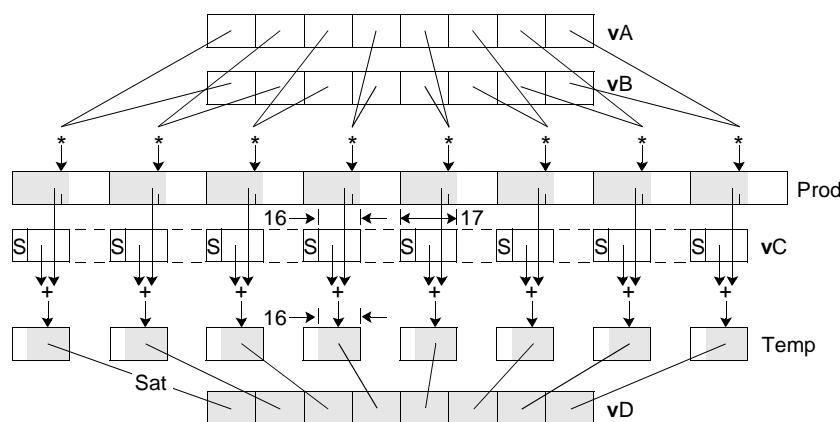
Each signed-integer halfword element in register **vA** is multiplied by the corresponding signed-integer halfword element in register **vB**, producing a 32-bit signed-integer product. The corresponding signed-integer halfword element in register **vC** is sign-extended to 17 bits and added to bits [0:16] of the product. If the intermediate result is greater than $2^{15}-1$, it saturates to $2^{15}-1$. If the intermediate result 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 halfword element in register **vD**.

Other registers altered:

- Vector Status and Control Register (VSCR):
Affected: SAT

Figure 6-54 shows the usage of the **vmhaddshs** instruction. Each of the eight elements in the registers **vA**, **vB**, **vC**, and **vD** is 16 bits in length.

Figure 6-54. **vmhaddshs**—Multiply-High and Add Eight Signed Integer Elements (16-Bit)



vmhraddshs

Vector Multiply High Round and Add Signed Halfword Saturate (0x1000 0021)

vmhraddshs**vD,vA,vB,vC**

Form: VA

04	vD	vA	vB	vC	33
0	5 6	10 11	15 16	20 21	25 26 31

```

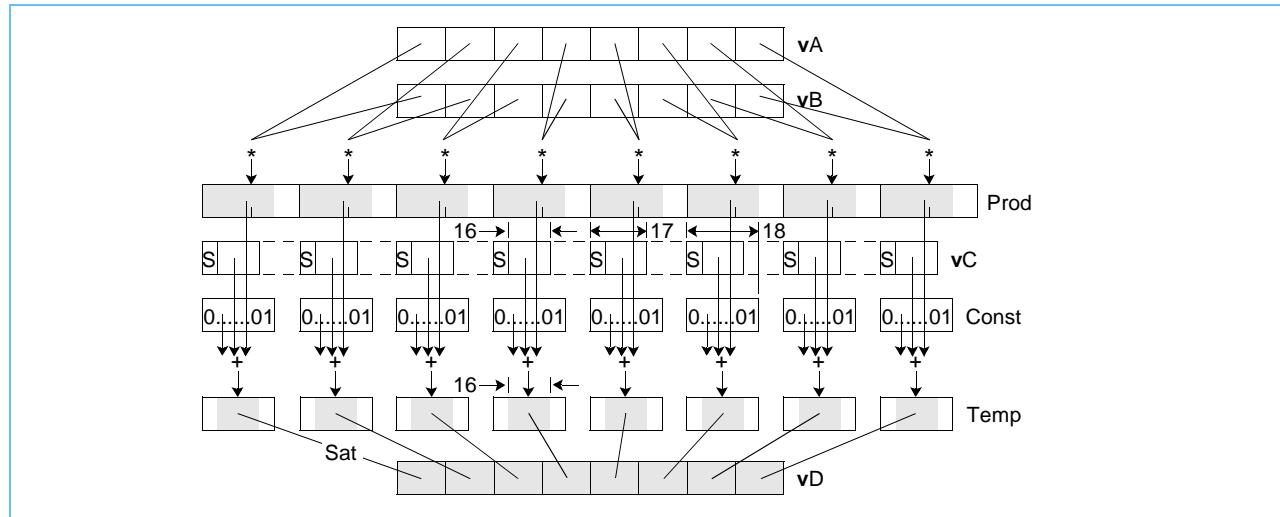
do i=0 to 127 by 16
    prod0:31 ← (vA)i:i+15 * si (vB)i:i+15
    prod0:31 ← prod0:31 + int 0x0000_4000
    temp0:16 ← prod0:16 + int SignExtend((vC)i:i+15,17)
    (vD)i:i+15 ← SIToSISat(temp0:16,16)
end

```

Each signed-integer halfword element in register **vA** is multiplied by the corresponding signed-integer halfword element in register **vB**, producing a 32-bit signed-integer product. The product is rounded by adding the value 0x0000_4000. The corresponding signed-integer halfword element in register **vC** is sign-extended to 17 bits and added to bits [0:16] of the rounded product. If the intermediate result is greater than $(2^{15}-1)$, it saturates to $(2^{15}-1)$. If the intermediate result 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 halfword element of register **vD**.

Figure 6-55 shows the usage of the **vmhraddshs** instruction. Each of the eight elements in the registers **vA**, **vB**, **vC**, and **vD** is 16 bits in length.

Figure 6-55. **vmhraddshs**—Multiply-High Round and Add Eight Signed Integer Elements (16-Bit)



vmminfp

Vector Minimum Floating Point(0x1000 044A)

vmminfp**vD,vA,vB****vmminfp**

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 ((vD)i:i+31 ← ((vA)i:i+31
    else ((vD)i:i+31 ← ((vB)i:i+31
  end

```

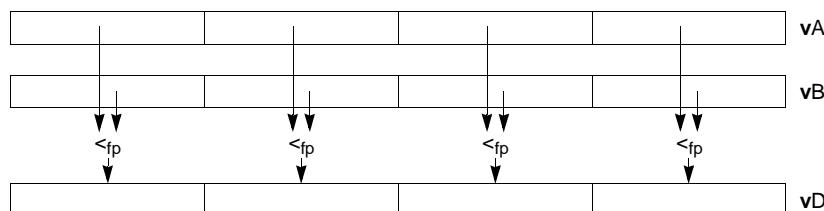
Each single-precision floating-point word element in register **vA** is compared to the corresponding single-precision floating-point word element in register **vB**. The smaller of the two single-precision floating-point values is placed into the corresponding word element of register **vD**.

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 **vmminfp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

*Figure 6-56. **vmminfp**—Minimum of Four Floating-Point Elements (32-Bit)*



vminsb

Vector Minimum Signed Byte (0x1000 0302)

vminsb**vD,vA,vB**

Form: VX

04	vD	vA	vB	770
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 8
  if (vA)i:i+7 <si (vB)i:i+7
    then (vD)i:i+7 ← (vA)i:i+7
    else (vD)i:i+7 ← (vB)i:i+7
end

```

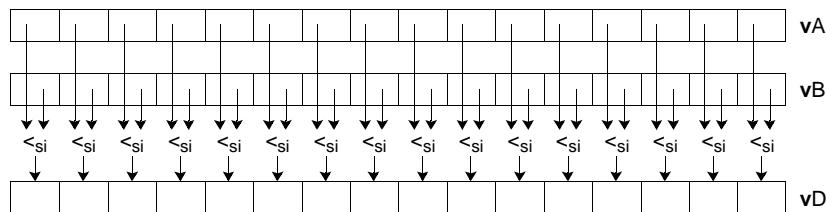
Each signed-integer byte element in register **vA** is compared to the corresponding signed-integer byte element in register **vB**. The larger of the two signed-integer values is placed into the corresponding byte element in register **vD**.

Other registers altered:

- None

Figure 6-57 shows the usage of the **vminsb** instruction. Each of the 16 elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-57. **vminsb**—Minimum of Sixteen Signed Integer Elements (8-Bit)



vminsh

Vector Minimum Signed Halfword (0x1000 0342)

vminsh**vD,vA,vB****vminsh**

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 (vD)i:i+15 ← (vA)i:i+15
    else (vD)i:i+15 ← (vB)i:i+15
  end

```

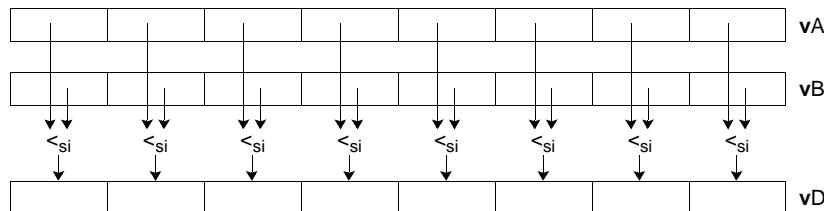
Each signed-integer halfword element in register **vA** is compared to the corresponding signed-integer halfword element in register **vB**. The larger of the two signed-integer values is placed into the corresponding halfword element in register **vD**.

Other registers altered:

- None

Figure 6-58 shows the usage of the **vminsh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-58. **vminsh**—Minimum of Eight Signed Integer Elements (16-Bit)



vminswo

Vector Minimum Signed Word (0x1000 0382)

vminswo**vD,vA,vB**

Form: VX

04	vD		vA		vB		898		
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 (vD)i:i+31 ← (vA)i:i+31
    else (vD)i:i+31 ← (vB)i:i+31
  end

```

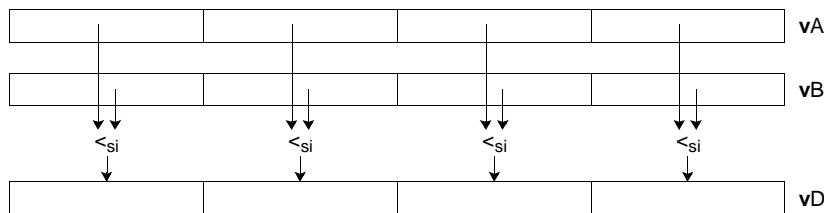
Each signed-integer word element in register **vA** is compared to the corresponding signed-integer word element in register **vB**. The larger of the two signed-integer values is placed into the corresponding word element in register **vD**.

Other registers altered:

- None

Figure 6-59 shows the usage of the **vminswo** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-59. **vminswo**—Minimum of Four Signed Integer Elements (32-Bit)



vminub

Vector Minimum Unsigned Byte (0x1000 0202)

vminub**vD,vA,vB**

Form: VX

04	vD	vA	vB	514
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 8
  if (vA)i:i+7 <ui (vB)i:i+7
    then (vD)i:i+7 ← (vA)i:i+7
    else (vD)i:i+7 ← (vB)i:i+7
  end

```

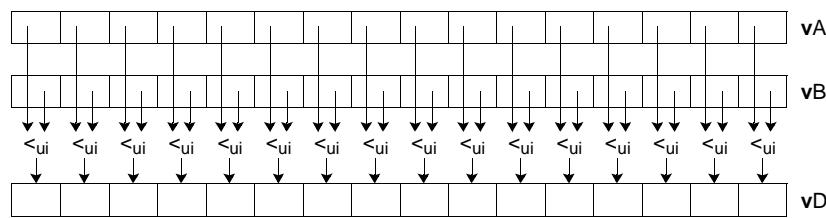
Each unsigned-integer byte element in register **vA** is compared to the corresponding unsigned-integer byte element in register **vB**. The larger of the two unsigned-integer values is placed into the corresponding byte element in register **vD**.

Other registers altered:

- None

Figure 6-60 shows the usage of the **vminub** instruction. Each of the 16 elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-60. **vminub**—Minimum of Sixteen Unsigned Integer Elements (8-Bit)



vminuh

Vector Minimum Unsigned Halfword (0x1000 0242)

vminuh**vD,vA,vB**

Form: VX

04	vD	vA	vB	578
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 16
  if (vA)i:i+15 <ui (vB)i:i+15
    then (vD)i:i+15 ← (vA)i:i+15
    else (vD)i:i+15 ← (vB)i:i+15
end

```

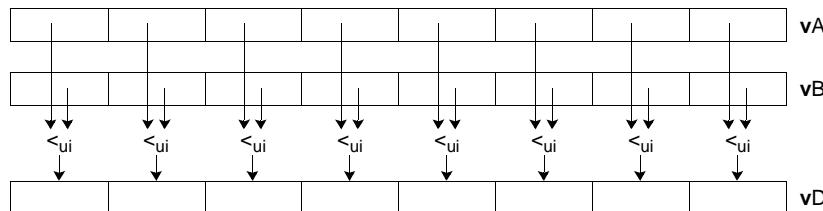
Each unsigned-integer halfword element in register **vA** is compared to the corresponding unsigned-integer halfword element in register **vB**. The larger of the two unsigned-integer values is placed into the corresponding halfword element in register **vD**.

Other registers altered:

- None

Figure 6-61 shows the usage of the **vminuh** instruction. Each of the eight elements in the register **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-61. **vminuh**—Minimum of Eight Unsigned Integer Elements (16-Bit)



vminuw

Vector Minimum Unsigned Word (0x1000 0282)

vminuw**vD,vA,vB**

Form: VX

04	vD	vA	vB	642
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  if (vA)i:i+31 <ui (vB)i:i+31
    then (vD)i:i+31 ← (vA)i:i+31
    else (vD)i:i+31 ← (vB)i:i+31
  end

```

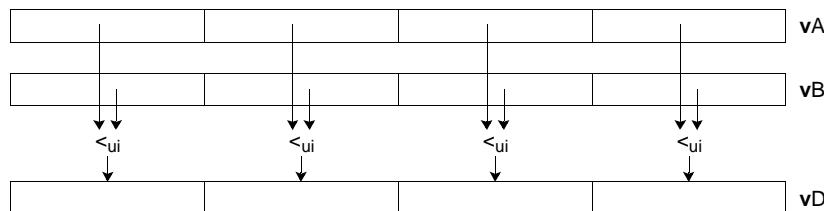
Each unsigned-integer word element in register **vA** is compared to the corresponding unsigned-integer word element in register **vB**. The larger of the two unsigned-integer values is placed into the corresponding word element in register **vD**.

Other registers altered:

- None

Figure 6-62 shows the usage of the **vminuw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** 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 Halfword Modulo (0x1000 0022)

vmladduhm**vD,vA,vB,vC**

Form: VA

04	vD	vA	vB	vC	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
  (vD)i:i+15 ← prod0:31 + int (vC)i:i+15
end

```

Each integer halfword element in register **vA** is multiplied by the corresponding integer halfword element in register **vB**, producing a 32-bit integer product. The product is added to the corresponding integer halfword element in register **vC**. The integer result is placed into the corresponding halfword element in register **vD**.

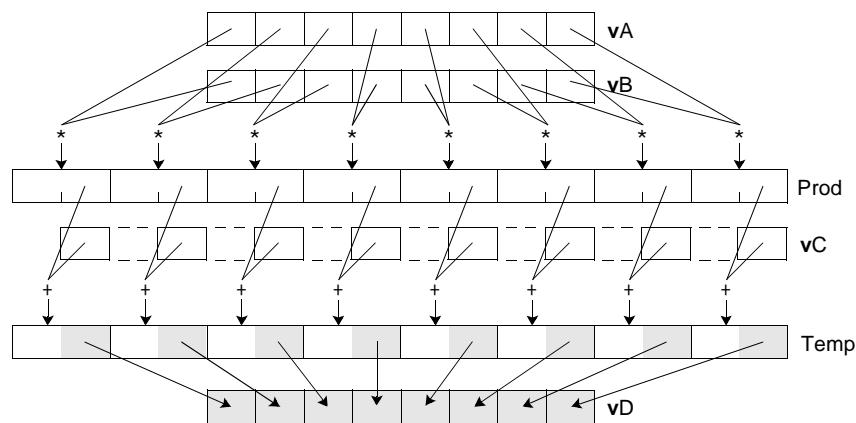
Note: **vmladduhm** can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-63 shows the usage of the **vmladduhm** instruction. Each of the eight elements in the registers **vA**, **vB**, **vC**, and **vD** is 16 bits in length.

Figure 6-63. **vmladduhm**—Multiply-Add of Eight Integer Elements (16-Bit)



vmrghb

Vector Merge High Byte (0x1000 000C)

vmrghb**vD,vA,vB**

Form: VX

04	vD	vA	vB	12
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:i+7 || (vB)i:i+7
end
```

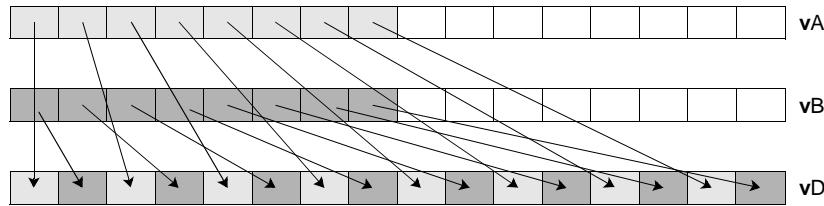
The byte elements in the high-order half of register **vA** are placed, in the same order, into the even-numbered byte elements of register **vD**. The byte elements in the high-order half of register **vB** are placed, in the same order, into the odd-numbered byte elements of register **vD**.

Other registers altered:

- None

Figure 6-64 shows the usage of the **vmrghb** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-64. **vmrghb**—Merge Eight High-Order Elements (8-Bit)



vmrghh

Vector Merge High Halfword (0x1000 004C)

vmrghh**vD,vA,vB**

Form: VX

04	vD	vA	vB	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
```

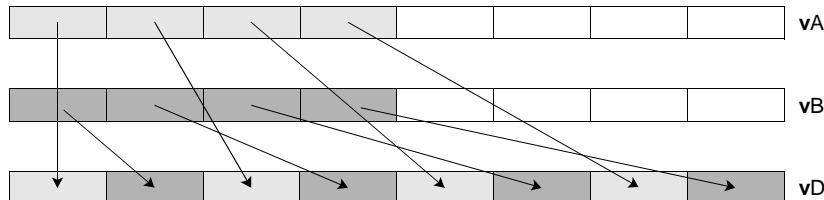
The halfword elements in the high-order half of register vA are placed, in the same order, into the even-numbered halfword elements of register vD. The halfword elements in the high-order half of register vB are placed, in the same order, into the odd-numbered halfword elements of register vD.

Other registers altered:

- None

Figure 6-65 shows the usage of the **vmrghh** instruction. Each of the eight elements in the registers vA, vB, and vD is 16 bits in length.

Figure 6-65. **vmrghh**—Merge Four High-Order Elements (16-Bit)



vmrghw

Vector Merge High Word (0x1000 008C)

vmrghw**vD,vA,vB**

Form: VX

04	vD	vA	vB	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
```

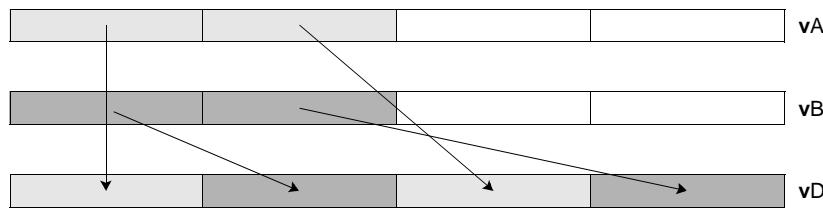
The word elements in the high-order half of register **vA** are placed, in the same order, into the even-numbered word elements of register **vD**. The word elements in the high-order half of register **vB** are placed, in the same order, into the odd-numbered word elements of register **vD**.

Other registers altered:

- None

Figure 6-66 shows the usage of the **vmrghw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

*Figure 6-66. **vmrghw**—Merge Two High-Order Elements (32-Bit)*



vmrglb

Vector Merge Low Byte (0x1000 010C)

vmrglb**vD,vA,vB**

Form: VX

04	vD	vA	vB	268
0	5 6	10 11	15 16	20 21

```
do i=0 to 63 by 8
  (vD)i*2:(i*2)+15 ← (vA)i+64:i+71 || (vB)i+64:i+71
end
```

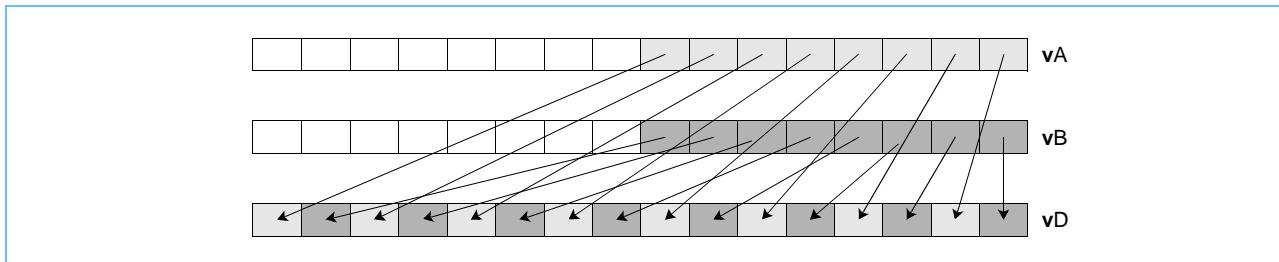
The byte elements in the low-order half of register vA are placed, in the same order, into the even-numbered byte elements of register vD. The byte elements in the low-order half of register vB are placed, in the same order, into the odd-numbered elements of register vD.

Other registers altered:

- None

Figure 6-67 shows the usage of the **vmrglb** instruction. Each of the sixteen elements in the registers vA, vB, and vD is 8 bits in length.

Figure 6-67. **vmrglb**—Merge Eight Low-Order Elements (8-Bit)



vmrglh

Vector Merge Low Halfword (0x1000 014C)

vmrglh**vD,vA,vB****vmrglh**

Form: VX

04	vD	vA	vB	332
0	5 6	10 11	15 16	20 21

```
do i=0 to 63 by 16
  (vD)i*2:(i*2)+31 ← (vA)i+64:i+79 || (vB)i+64:i+79
end
```

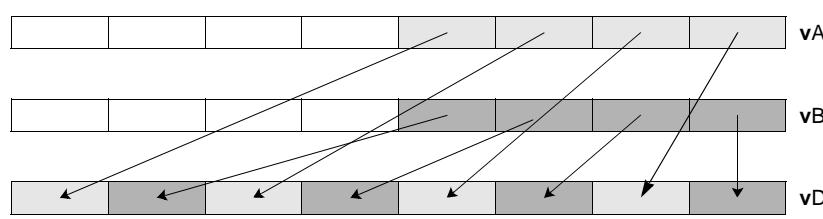
The halfword elements in the low-order half of register **vA** are placed, in the same order, into the even-numbered halfword elements of register **vD**. The halfword elements in the low-order half of register **vB** are placed, in the same order, into the odd-numbered halfword elements of register **vD**.

Other registers altered:

- None

Figure 6-68 shows the usage of the **vmrglh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-68. **vmrglh**—Merge Four Low-Order Elements (16-Bit)



vmrglw

Vector Merge Low Word (0x1000 018C)

vmrglw**vD,vA,vB**

Form: VX

04	vD	vA	vB	396
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+64:i+95 || (vB)i+64:i+95
end
```

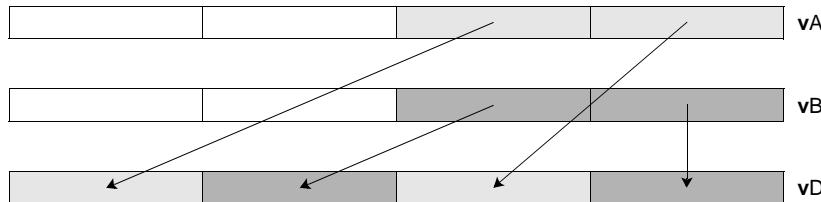
The word elements in the low-order half of register **vA** are placed, in the same order, into the even-numbered word elements of register **vD**. The word elements in the low-order half of register **vB** are placed, in the same order, into the odd-numbered word elements of register **vD**.

Other registers altered:

- None

Figure 6-69 shows the usage of the **vmrglw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-69. **vmrglw**—Merge Two Low-Order Elements (32-Bit)



vmsummbm**vmsummbm**

Vector Multiply Sum Mixed-Sign Byte Modulo (0x1000 0025)

vmsummbm**vD,vA,vB,vC**

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
  (vD)i:i+31 ← temp0:31
end

```

For each word element in register vC, the following operations are performed in the order shown.

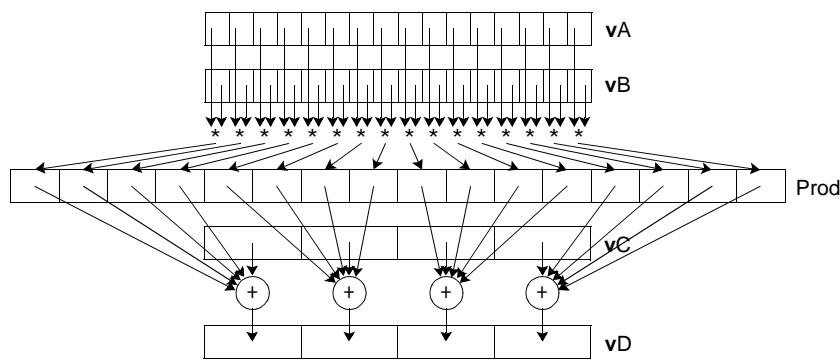
- Each of the four signed-integer byte elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer byte element in register vB, producing a signed-integer 16-bit product.
- The signed-integer modulo sum of these four products is added to the signed-integer word element in register vC.
- The signed-integer result is placed into the corresponding word element of register vD.

Other registers altered:

- None

Figure 6-70 shows the usage of the **vmsummbm** instruction. Each of the sixteen elements in the registers vA, and vB is 8 bits in length. Each of the four elements in the registers vC and vD is 32 bits in length.

Figure 6-70. **vmsummbm**—Multiply-Sum of Integer Elements (8-Bit to 32-Bit)



vmsumshm

Vector Multiply Sum Signed Halfword Modulo (0x1000 0028)

vmsumshm**vD,vA,vB,vC**

Form: VA

04	vD	vA	vB	vC	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
  end
  (vD)i:i+31 ← temp0:31
end

```

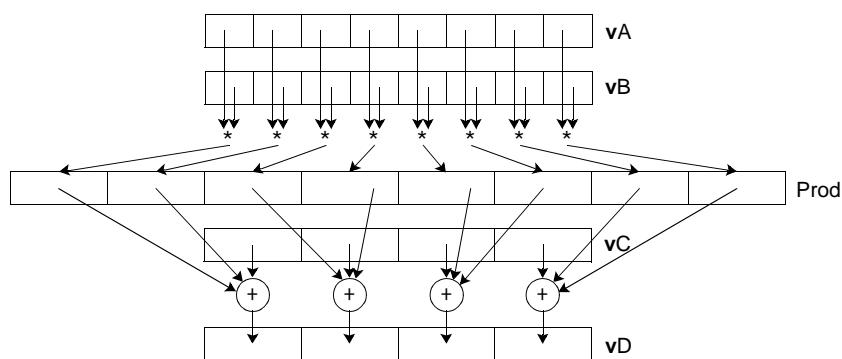
For each word element in register vC, the following operations are performed in the order shown:

- Each of the two signed-integer halfword elements contained in the corresponding word element of register vA is multiplied by the corresponding signed-integer halfword element in register vB, 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 register vC.
- The signed-integer result is placed into the corresponding word element of register vD.

Other registers altered:

- None

Figure 6-71 shows the usage of the **vmsumshm** instruction. Each of the eight elements in the registers vA, and vB is 16 bits in length. Each of the four elements in the registers vC and vD is 32 bits in length.

Figure 6-71. **vmsumshm**—Multiply-Sum of Signed Integer Elements (16-Bit to 32-Bit)

vmsumshs

Vector Multiply Sum Signed Halfword Saturate (0x1000 0029)

vmsumshs**vD,vA,vB,vC**

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)
  end
  (vD)i:i+31 ← SItoSIsat(temp0:33,32)
end

```

For each word element in register vC, the following operations are performed in the order shown:

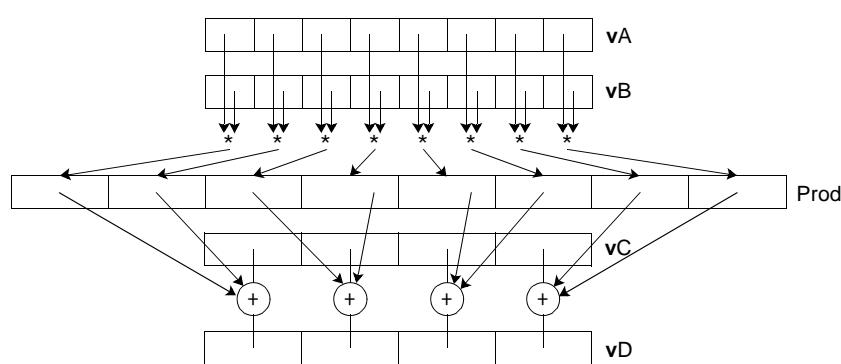
- Each of the two signed-integer halfword elements in the corresponding word element of register vA is multiplied by the corresponding signed-integer halfword element in register vB, producing a signed-integer 32-bit product.
- The signed-integer sum of these two products is added to the signed-integer word element in register vC.
- 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 register vD.

Other registers altered:

- SAT

Figure 6-72 shows the usage of the **vmsumshs** instruction. Each of the eight elements in the registers vA, and vB is 16 bits in length. Each of the four elements in the registers vC and vD is 32 bits in length.

Figure 6-72. **vmsumshs**—Multiply-Sum of Signed Integer Elements (16-Bit to 32-Bit)



vmsumubm

Vector Multiply Sum Unsigned Byte Modulo (0x1000 0024)

vmsumubm**vD,vA,vB,vC**

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)
  end
  (vD)i:i+31 ← temp0:31
end

```

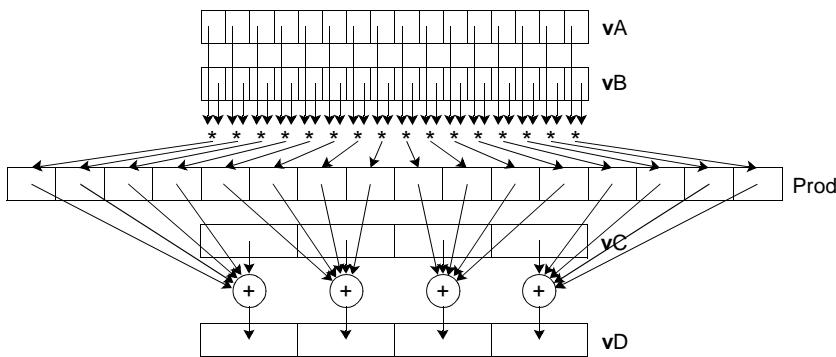
For each word element in register vC, the following operations are performed in the order shown:

- Each of the four unsigned-integer byte elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer byte element in register vB, 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 register vC.
- The unsigned-integer result is placed into the corresponding word element of register vD.

Other registers altered:

- None

Figure 6-73 shows the usage of the **vmsumubm** instruction. Each of the sixteen elements in the registers vA, and vB is 8 bits in length. Each of the four elements in the register vC and vD is 32 bits in length.

Figure 6-73. **vmsumubm**—Multiply-Sum of Unsigned Integer Elements (8-Bit to 32-Bit)

vmsumuhm

Vector Multiply Sum Unsigned Halfword Modulo (0x1000 0026)

vmsumuhm**vD,vA,vB,vC**

Form: VA

04	vD	A	vB	vC	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
  end
  (vD)i:i+31 ← temp2:33
end

```

For each word element in register vC, the following operations are performed in the order shown:

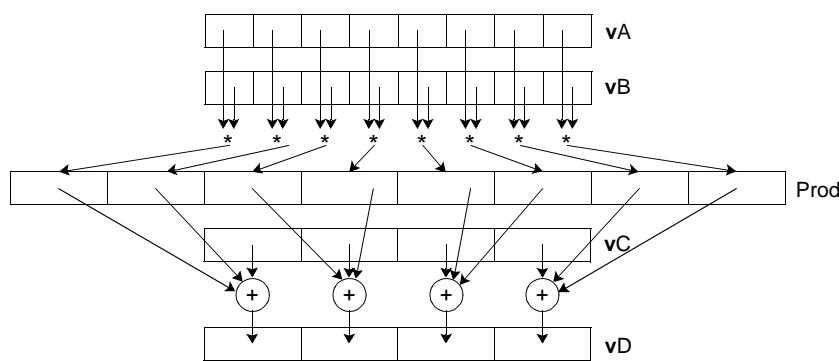
- Each of the two unsigned-integer halfword elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer halfword element in register vB, producing an unsigned-integer 32-bit product.
- The unsigned-integer sum of these two products is added to the unsigned-integer word element in register vC.
- The unsigned-integer result is placed into the corresponding word element of register vD.

Other registers altered:

- None

Figure 6-74 shows the usage of the **vmsumuhm** instruction. Each of the eight elements in the registers vA, and vB is 16 bits in length. Each of the four elements in the registers vC and vD is 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 Halfword Saturate (0x1000 0027)

vmsumuhs**vD,vA,vB,vC**

Form: VA

04	vD	vA	vB	vC	39
0	5 6	10 11	15 16	20 21	25 26 31

```

do i=0 to 127 by 32
  temp0:33 ← ZeroExtend( (vC)i:i+31, 34)
  do j=0 to 31 by 16
    prod0:31 ← (vA)i+j:i+j+15 * ui (vB)i+j:i+j+15
    temp0:33 ← temp0:33 + int ZeroExtend(prod0:31, 34)
  end
  (vD)i:i+31 ← UItoUIsat (temp0:33, 32)
end

```

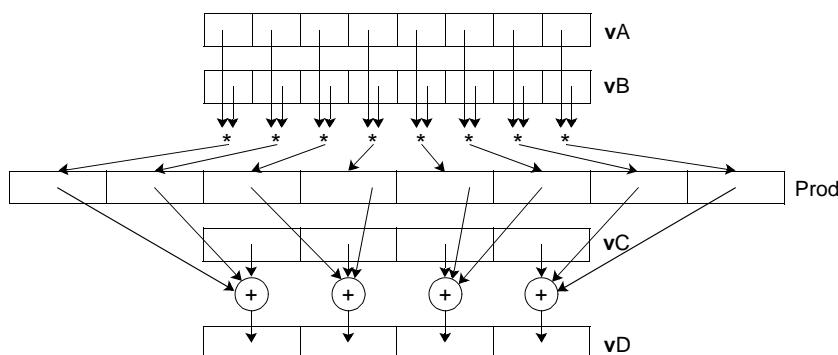
For each word element in register vC, the following operations are performed in the order shown:

- Each of the two unsigned-integer halfword elements contained in the corresponding word element of register vA is multiplied by the corresponding unsigned-integer halfword element in register vB, producing an unsigned-integer 32-bit product.
- The sum of the two 32-bit unsigned-integer products is added to the unsigned-integer word element in register vC.
- If the intermediate result is greater than $2^{32} - 1$, it saturates to $2^{32} - 1$. If saturation occurs, the SAT bit is set.
- The unsigned-integer result is placed into the corresponding word element of register vD.

Other registers altered:

- SAT

Figure 6-75 shows the usage of the **vmsumuhs** instruction. Each of the eight elements in the registers vA, and vB is 16 bits in length. Each of the four elements in the registers vC and vD is 32 bits in length.

Figure 6-75. **vmsumuhs**—Multiply-Sum of Unsigned Integer Elements (16-Bit to 32-Bit)

vmulesb

Vector Multiply Even Signed Byte (0x1000 0308)

vmulesb**vD,vA,vB**

Form: VX

04	vD	vA	vB	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
  (vD)i:i+15 ← prod0:15
end

```

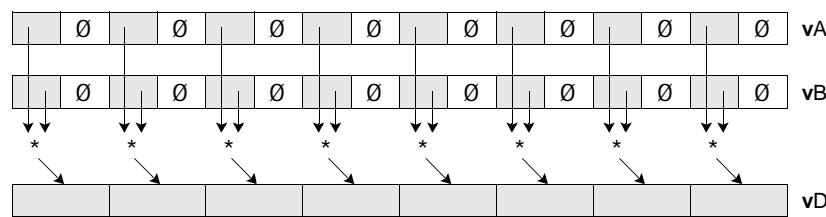
Each even-numbered signed-integer byte element in register **vA** is multiplied by the corresponding signed-integer byte element in register **vB**. The eight 16-bit signed-integer products are placed, in the same order, into the eight halfwords of register **vD**.

Other registers altered:

- None

Figure 6-76 shows the usage of the **vmulesb** instruction. Each of the sixteen elements in the registers **vA**, and **vB** is 8 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-76. **vmulesb**—Even Multiply of Eight Signed Integer Elements (8-Bit)



vmulesh

Vector Multiply Even Signed Halfword (0x1000 0348)

vmulesh**vD,vA,vB**

Form: VX

04	vD	vA	vB	840
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
    prod0:31 ← (vA)i:i+15 *si (vB)i:i+15
    (vD)i:i+31 ← prod0:31
end

```

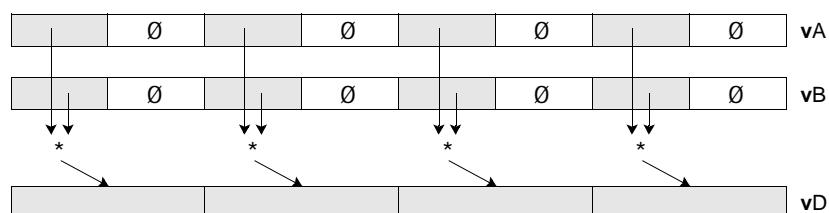
Each even-numbered signed-integer halfword element in register **vA** is multiplied by the corresponding signed-integer halfword element in register **vB**. The four 32-bit signed-integer products are placed, in the same order, into the four word elements of register **vD**.

Other registers altered:

- None

Figure 6-77 shows the usage of the **vmulesh** instruction. Each of the eight elements in the registers **vA** and **vB** is 16 bits in length. Each of the four elements in the register **vD** is 32 bits in length.

Figure 6-77. **vmulesb**—Even Multiply of Four Signed Integer Elements (16-Bit)



vmuleub

Vector Multiply Even Unsigned Byte (0x1000 0208)

vmuleub**vD,vA,vB**

Form: VX

04	vD	vA	vB	520
0	5 6	10 11	15 16	20 21

```

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

```

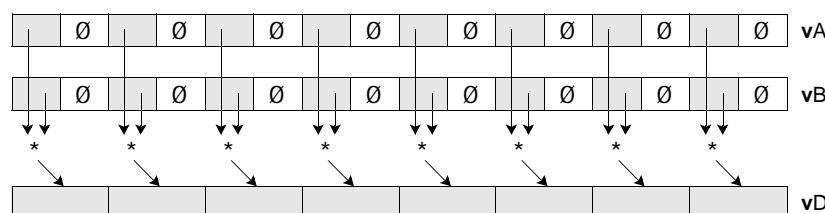
Each even-numbered unsigned-integer byte element in register **vA** is multiplied by the corresponding unsigned-integer byte element in register **vB**. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight halfword elements of register **vD**.

Other registers altered:

- None

Figure 6-78 shows the usage of the **vmuleub** instruction. Each of the sixteen elements in the registers **vA** and **vB** is 8 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-78. **vmuleub**—Even Multiply of Eight Unsigned Integer Elements (8-Bit)



vmuleuh

Vector Multiply Even Unsigned Halfword (0x1000 0248)

vmuleuh**vD,vA,vB**

Form: VX

04	vD	vA	vB	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

```

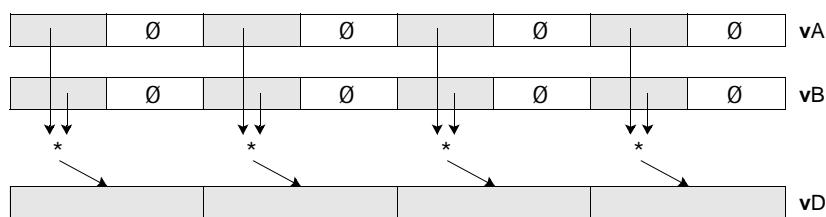
Each even-numbered unsigned-integer halfword element in register **vA** is multiplied by the corresponding unsigned-integer halfword element in register **vB**. The four 32-bit unsigned-integer products are placed, in the same order, into the four word elements of register **vD**.

Other registers altered:

- None

Figure 6-79 shows the usage of the **vmuleuh** instruction. Each of the eight elements in the registers **vA** and **vB** is 16 bits in length. Each of the four elements in the register **vD** is 32 bits in length.

Figure 6-79. **vmuleuh**—Even Multiply of Four Unsigned Integer Elements (16-Bit)



vmulosb

Vector Multiply Odd Signed Byte (0x1000 0108)

vmulosb**vD,vA,vB**

Form: VX

04	vD	vA	vB	264
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 16
  prod0:15 ← (vA)i+8:i+15 * si (vB)i+8:i+15
  (vD)i:i+15 ← prod0:15
end

```

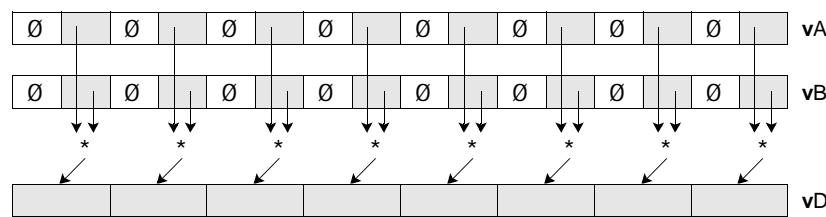
Each odd-numbered signed-integer byte element in register **vA** is multiplied by the corresponding signed-integer byte element in register **vB**. The eight 16-bit signed-integer products are placed, in the same order, into the eight halfword elements in register **vD**.

Other registers altered:

- None

Figure 6-80 shows the usage of the **vmulosb** instruction. Each of the sixteen elements in the registers **vA** and **vB** is 8 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-80. **vmulosb**—Odd Multiply of Eight Signed Integer Elements (8-Bit)



vmulosh

Vector Multiply Odd Signed Halfword (0x1000 0148)

vmulosh**vD,vA,vB**

Form: VX

04	vD	vA	vB	328
0	5 6	10 11	15 16	20 21

```
do i=0 to 127 by 32
    prod0:31 ← (vA)i+16:i+31 *si (vB)i+16:i+31
    (vD)i:i+31 ← prod0:31
end
```

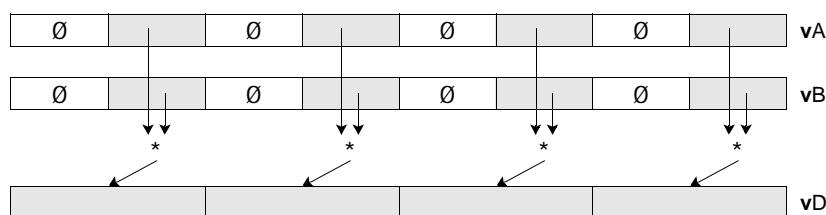
Each odd-numbered signed-integer halfword element in register **vA** is multiplied by the corresponding signed-integer halfword element in register **vB**. The four 32-bit signed-integer products are placed, in the same order, into the four word elements in register **vD**.

Other registers altered:

- None

Figure 6-81 shows the usage of the **vmuleuh** instruction. Each of the eight elements in the registers **vA** and **vB** is 16 bits in length. Each of the four elements in the register **vD** is 32 bits in length.

Figure 6-81. **vmuleuh**—Odd Multiply of Four Unsigned Integer Elements (16-Bit)



vmuloub

Vector Multiply Odd Unsigned Byte (0x1000 0008)

vmuloub**vD,vA,vB**

Form: VX

04	vD	vA	vB	8
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 8
  prod0:15 ← (vA)i+8:i+15 * ui (vB)i+n:i+15
  (vD)i:i+15 ← prod0:15
end

```

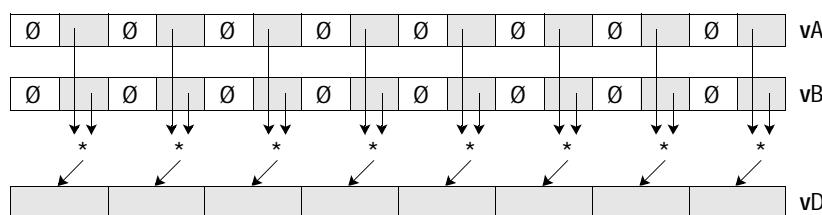
Each odd-numbered unsigned-integer byte element in register **vA** is multiplied by the corresponding unsigned-integer byte element in register **vB**. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight halfword elements in register **vD**.

Other registers altered:

- None

Figure 6-82 shows the usage of the **vmuloub** instruction. Each of the sixteen elements in the registers **vA** and **vB** is 8 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-82. **vmuloub**—Odd Multiply of Eight Unsigned Integer Elements (8-Bit)



vmulouh**vmulouh**

Vector Multiply Odd Unsigned Halfword (0x1000 0048)

vmulouh**vD,vA,vB**

Form: VX

04	vD	vA	vB	72
0	5 6	10 11	15 16	20 21

```
do i=0 to 127 by 16
    prod0:31 ← (vA)i+16:i+31 * ui (vB)i+n:i+31
    (vD)i:i+31 ← prod0:31
end
```

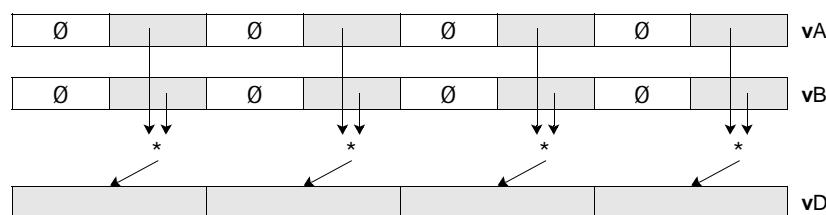
Each odd-numbered unsigned-integer halfword element in register **vA** is multiplied by the corresponding unsigned-integer halfword element in register **vB**. The four 32-bit unsigned-integer products are placed, in the same order, into the four word elements in register **vD**.

Other registers altered:

- None

Figure 6-83 shows the usage of the **vmulouh** instruction. Each of the eight elements in the registers **vA** and **vB** is 16 bits in length. Each of the four elements in the register **vD** 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 (0x1000 002F)

vnmsubfp**vD,vA,vC,vB**

Form: VA

04	vD	vA	vB	vC	47
0	5 6	10 11	15 16	20 21	25 26

```
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 register **vA** is multiplied by the corresponding single-precision floating-point word element in register **vC**. The corresponding single-precision floating-point word element in register **vB** 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 in register **vD**.

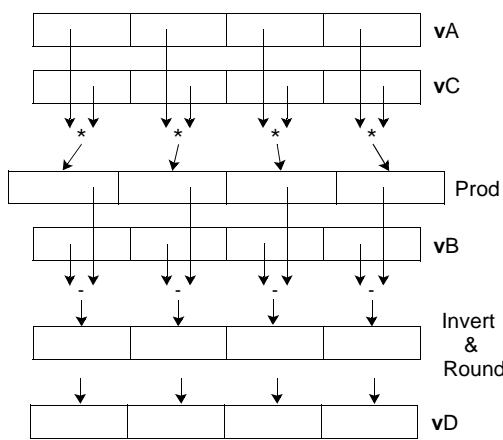
Note: Only one rounding occurs in this operation. Also note that a QNaN result is not negated.

Other registers altered:

- None

Figure 6-84 shows the usage of the **vnmsubfp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-84. **vnmsubfp**—Negative Multiply-Subtract of Four Floating-Point Elements (32-Bit)



vnor

Vector Logical NOR (0x1000 0504)

vnor**vD,vA,vB****vnor**

Form: VX

04	vD	vA	vB	1284
0	5 6	10 11	15 16	20 21

$$(vD) \leftarrow \neg((vA) \mid (vB))$$

The contents of **vA** are bitwise ORed with the contents of register **vB** and the complemented result is placed into register **vD**.

Other registers altered:

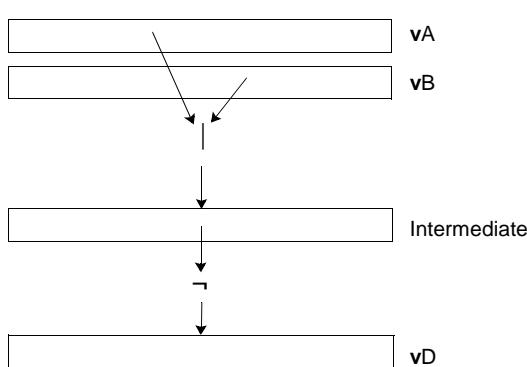
- None

Simplified mnemonics:

vnot **vD, vS** equivalent to **vnor** **vD, vS, vS**

*Figure 6-85 shows the usage of the **vnor** instruction.*

*Figure 6-85. **vnor**—Bitwise NOR of 128-Bit Vector*



vor

Vector Logical OR (0x1000 0484)

vor**vD,vA,vB****vor**

Form: VX

04	vD	vA	vB	1156
0	5 6	10 11	15 16	20 21

(vD) \leftarrow (vA) | (vB)

The contents of register vA are ORed with the contents of register vB and the result is placed into register vD.

Other registers altered:

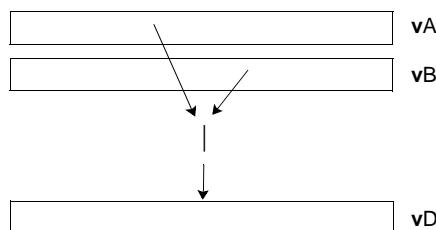
- None

Simplified mnemonics:

vmr **vD, vS** equivalent to **vor** **vD, vS, vS**

Figure 6-86 shows the usage of the vor instruction.

Figure 6-86. vor—Bitwise OR of 128-Bit Vector



vperm

Vector Permute (0x1000 002B)

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
    (vD)i:i+7 ← tempb:b+7
end

```

Let the source vector be the concatenation of the contents of register vA followed by the contents of register vB. 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 vC are placed into byte element i of register vD.

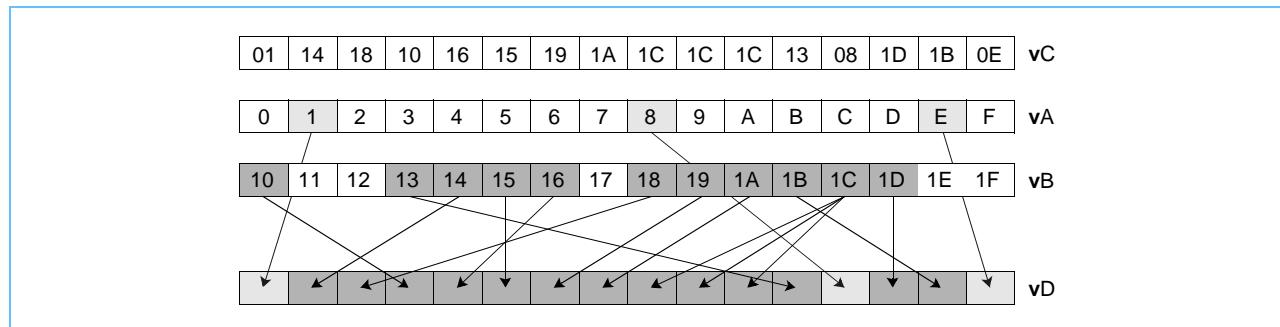
Other registers altered:

- None

Programming note: See the programming notes with the Load Vector for Shift Left (page 127) and Load Vector for Shift Right (page 129) instructions for examples of usage on the **vperm** instruction.

Figure 6-87 shows the usage of the **vperm** instruction. Each of the sixteen elements in the registers vA, vB, and vD is 8 bits in length.

Figure 6-87. **vperm**—Concatenate Sixteen Integer Elements (8-Bit)



Vector/SIMD Multimedia Extension Technology**vpkpx**

Vector Pack Pixel32 (0x1000 030E)

vpkpx**vD,vA,vB**

Form: VX

04	vD	vA	vB	782
0	5 6	10 11	15 16	20 21

```

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 register **vA** followed by the contents of register **vB**. Each word element in the source vector is packed to produce a 16-bit value as described below and placed into the corresponding halfword element of **vD**. 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)

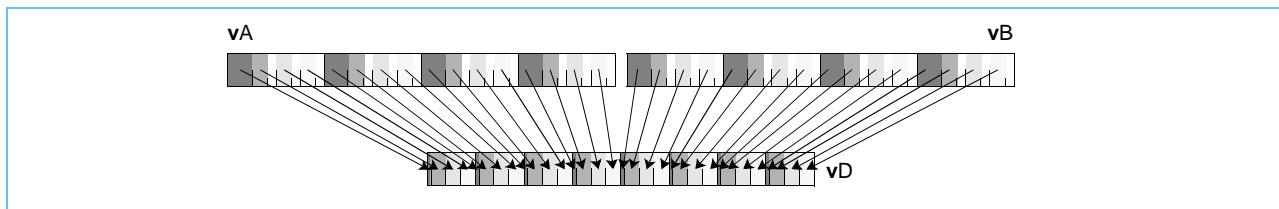
Other registers altered:

- None

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

Figure 6-88 shows the usage of the **vpkpx** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-88. **vpkpx**—Pack Eight Elements (32-Bit) to Eight Elements (16-Bit)



vpkshss**vpkshss**

Vector Pack Signed Halfword Signed Saturate (0x1000 018E)

vpkshss**vD,vA,vB**

Form: VX

04	vD	vA	vB	398
0	5 6	10 11	15 16	20 21

```

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 register **vA** followed by the contents of register **vB**.

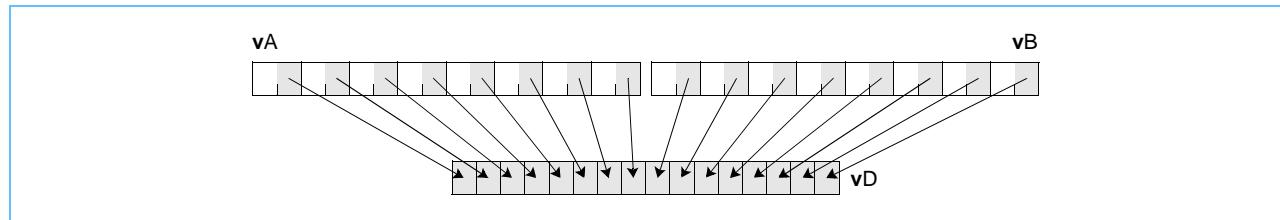
Each signed integer halfword element in the source vector is converted to an 8-bit signed integer. If the value of the element is greater than $(2^7 - 1)$ it saturates to $(2^7 - 1)$ and if the value is less than (-2^7) it saturates to (-2^7) . If saturation occurs, the SAT bit is set. The result is placed into the corresponding byte element of register **vD**.

Other registers altered:

- SAT

Figure shows the usage of the **vpkshss** instruction. Each of the eight elements in the registers **vA** and **vB** is 16 bits in length. Each of the sixteen elements in the register **vD** is 8 bits in length.

vpkshss—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Signed Integer Elements (8-Bit)



vpkshus

Vector Pack Signed Halfword Unsigned Saturate (0x1000 010E)

vpkshus**vD,vA,vB**

Form: VX

04	vD	vA	vB	270
0	5 6	10 11	15 16	20 21

```

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 register **vA** followed by the contents of register **vB**.

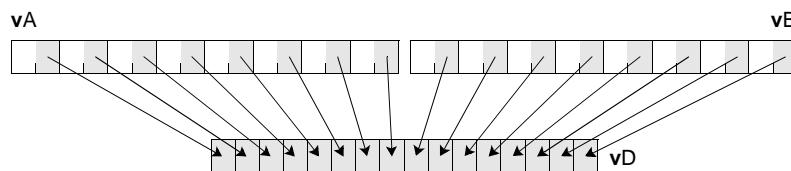
Each signed integer halfword element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than $(2^8 - 1)$ it saturates to $(2^8 - 1)$ and if the value is less than '0' the result saturates to '0'. If saturation occurs, the SAT bit is set. The result is placed into the corresponding byte element of register **vD**.

Other registers altered:

- SAT

Figure 6-89 shows the usage of the **vpkshus** instruction. Each of the eight elements in the registers **vA** and **vB** is 16 bits in length. Each of the sixteen elements in the register **vD** is 8 bits in length.

Figure 6-89. **vpkshus**—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)



vpkswss

Vector Pack Signed Word Signed Saturate (0x1000 01CE)

vpkswss**vD,vA,vB**

Form: VX

04	vD	vA	vB	462
0	5 6	10 11	15 16	20 21

```

do i=0 to 63 by 16
  (vD)i:i+15 ← SItoSIsat((vA)i*2:(i*2)+31,16)
  (vD)i+64:i+79← SItoSIsat((vB)i*2:(i*2)+31,16)
end

```

Let the source vector be the concatenation of the contents of register **vA** followed by the contents of register **vB**.

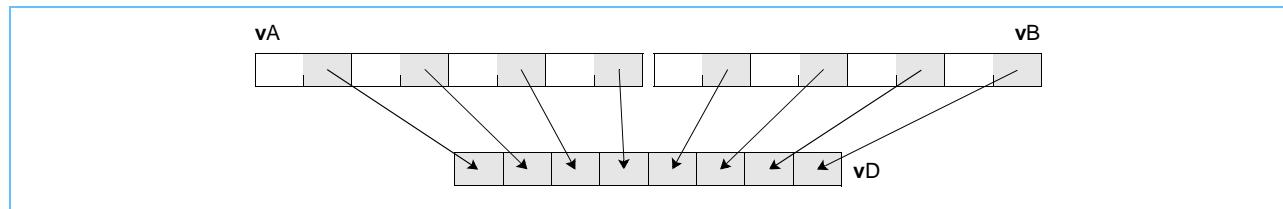
Each signed integer word element in the source vector is converted to a 16-bit signed integer halfword. If the value of the element is greater than ($2^{15} - 1$) the result saturates to ($2^{15} - 1$) and if the value is less than (-2^{15}) the result saturates to (-2^{15}). If saturation occurs, the SAT bit is set. The result is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- SAT

Figure 6-90 shows the usage of the **vpkswss** instruction. Each of the four elements in the registers **vA** and **vB** is 32 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-90. **vpkswss**—Pack Eight Signed Integer Elements (32-Bit) to Eight Signed Integer Elements (16-Bit)



vpkswus

Vector Pack Signed Word Unsigned Saturate (0x1000 014E)

vpkswus**vD,vA,vB**

Form: VX

04	vD	vA	vB	334
0	5 6	10 11	15 16	20 21

```

do i=0 to 63 by 16
  (vD)i:i+15 ← SItouiSat((vA)i*2:(i*2)+31,16)
  (vD)i+64:i+79← SItouiSat((vB)i*2:(i*2)+31,16)
end

```

Let the source vector be the concatenation of the contents of register **vA** followed by the contents of register **vB**.

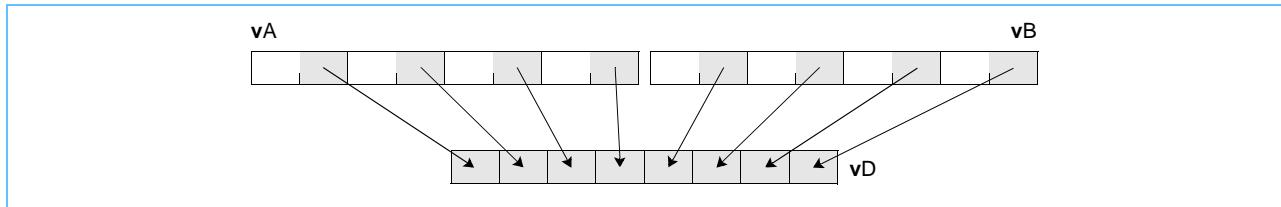
Each signed integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than $(2^{16} - 1)$ the result saturates to $(2^{16} - 1)$ and if the value is less than '0' the result saturates to '0'. If saturation occurs, the SAT bit is set. The result is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- SAT

Figure 6-91 shows the usage of the **vpkswus** instruction. Each of the four elements in the registers **vA** and **vB** is 32 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-91. **vpkswus**—Pack Eight Signed Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)



vpkuhum

Vector Pack Unsigned Halfword Unsigned Modulo (0x1000 000E)

vpkuhum**vD,vA,vB**

Form: VX

04	vD	vA	vB	14
0	5 6	10 11	15 16	20 21

```

do i=0 to 63 by 8
  (vD)i:i+7 ← (vA)(i*2)+8:(i*2)+15
  (vD)i+64:i+71 ← (vB)(i*2)+8:(i*2)+15
end

```

Let the source vector be the concatenation of the contents of register **vA** followed by the contents of register **vB**.

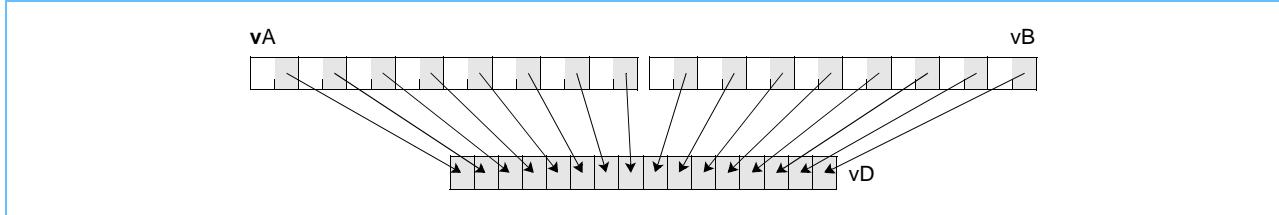
The low-order byte of each halfword element in the source vector is placed into the corresponding byte element of register **vD**.

Other registers altered:

- None

Figure 6-92 shows the usage of the **vpkuhum** instruction. Each of the eight elements in the registers **vA** and **vB** is 16 bits in length. Each of the sixteen elements in the register **vD** is 8 bits in length.

Figure 6-92. vpkuhum—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)



vpkuhus**vpkuhus**

Vector Pack Unsigned Halfword Unsigned Saturate (0x1000 008E)

vpkuhus**vD,vA,vB**

Form: VX

04	vD	vA	vB	142
0	5 6	10 11	15 16	20 21

```

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 register **vA** followed by the contents of register **vB**.

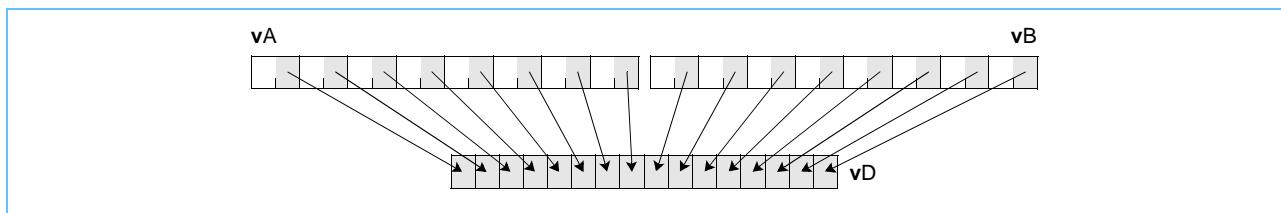
Each unsigned integer halfword element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than $(2^8 - 1)$ the result saturates to $(2^8 - 1)$. If saturation occurs, the SAT bit is set. The result is placed into the corresponding byte element of register **vD**.

Other registers altered:

- SAT

Figure 6-93 shows the usage of the **vpkuhus** instruction. Each of the eight elements in the vectors, **vA**, and **vB**, is 16 bits in length. Each of the sixteen elements in the vector **vD**, is 8 bits in length.

Figure 6-93. vpkuhus—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)



vpkuwum

Vector Pack Unsigned Word Unsigned Modulo (0x1000 004E)

vpkuwum**vD,vA,vB**

Form: VX

04	vD	vA	vB	78
0	5 6	10 11	15 16	20 21

```
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 register **vA** followed by the contents of register **vB**.

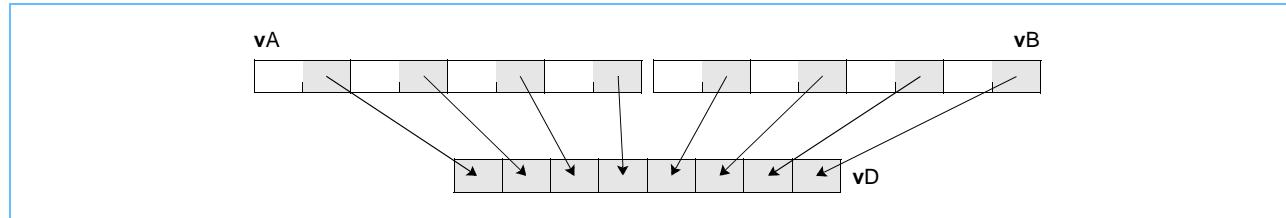
The low-order halfword of each word element in the source vector is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- None

Figure 6-94 shows the usage of the **vpkuwum** instruction. Each of the four elements in the registers **vA** and **vB** is 32 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-94. **vpkuwum**—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)



vpkuwus

Vector Pack Unsigned Word Unsigned Saturate (0x1000 00CE)

vpkuwus**vD,vA,vB**

Form: VX

04	vD	vA	vB	206
0	5 6	10 11	15 16	20 21

```

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 register **vA** followed by the contents of register **vB**.

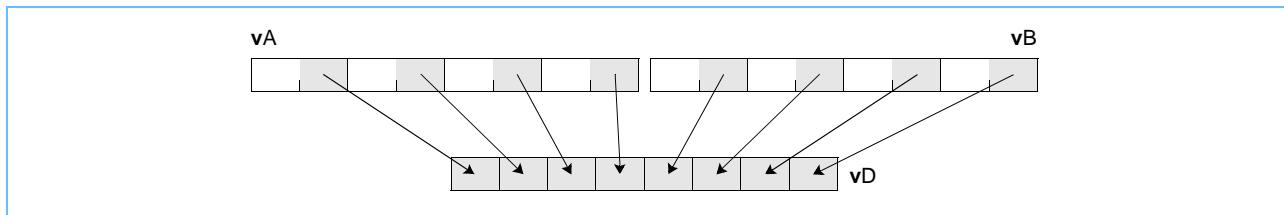
Each unsigned integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than $(2^{16} - 1)$ the result saturates to $(2^{16} - 1)$. If saturation occurs, the SAT bit is set. The result is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- SAT

Figure 6-95 shows the usage of the **vpkuwus** instruction. Each of the four elements in the registers **vA** and **vB** is 32 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.

Figure 6-95. **vpkuwum**—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)





vrefp

Vector Reciprocal Estimate Floating Point (0x1000 010A)

vrefp

vD,vB

vrefp

Form: VX

04	vD	0 0 0 0	vB	266
0	5 6	10 11	15 16	20 21

```
do i=0 to 127 by 32
  x ← (vB)i:i+31
  (vD)i:i+31 ← 1/x
end
```

The single-precision floating-point estimate of the reciprocal of each single-precision floating-point element in register vB is placed into the corresponding element of register vD.

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:

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

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

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

Table 6-7. vrefp—Special Values of the Element in vB

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.

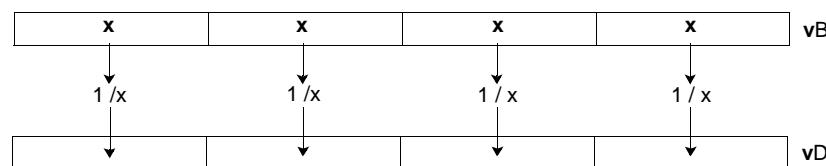
Other registers altered:

- None

Figure 6-96 on page 236 shows the usage of the vrefp instruction. Each of the four elements in the registers vB and vD is 32 bits in length.

Vector/SIMD Multimedia Extension Technology

Figure 6-96. **vrefp**—Reciprocal Estimate of Four Floating-Point Elements (32-Bit)



vrfim**vrfim**

Vector Round to Floating-Point Integer toward Minus Infinity (0x1000 02CA)

vrfim**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	714
0	5 6	10 11	15 16	20 21

```
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 register **vB** is rounded to a single-precision floating-point integer, using the rounding mode Round toward -Infinity, and placed into the corresponding word element of register **vD**.

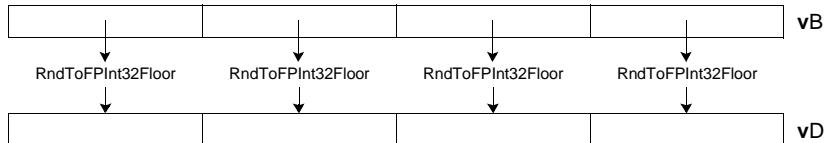
Note: The result is independent of VSCR[NJ].

Other registers altered:

- None

*Figure 6-97 shows the usage of the **vrfim** instruction. Each of the four elements in the registers **vB** and **vD** is 32 bits in length.*

*Figure 6-97. **vrfim**—Round to Minus Infinity of Four Floating-Point Integer Elements (32-Bit)*



vrfin

Vector Round to Floating-Point Integer Nearest (0x1000 020A)

vrfin**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	522
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  (vD)i:i+31 ← RndToFPInt32Near( (vB)i:i+31)
end

```

Each single-precision floating-point word element in register **vB** is rounded to a single-precision floating-point integer, using the rounding mode Round to Nearest, and placed into the corresponding word element of register **vD**.

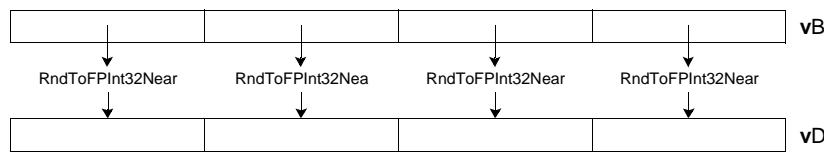
Note: The result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-98 shows the usage of the **vrfin** instruction. Each of the four elements in the registers **vB** and **vD** is 32 bits in length.

Figure 6-98. **vrfin**—Nearest Round to Nearest of Four Floating-Point Integer Elements (32-Bit)



vrfip**vrfip**

Vector Round to Floating-Point Integer toward Plus Infinity (0x1000 028A)

vrfip**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	650
0	5 6	10 11	15 16	20 21

```
do i=0 to 127 by 32
  (vD)i:i+31 ← RndToFPInt32Ceil( (vB)i:i+31)
end
```

Each single-precision floating-point word element in **vB** is rounded to a single-precision floating-point integer, using the rounding mode Round toward +Infinity, and placed into the corresponding word element of **vD**.

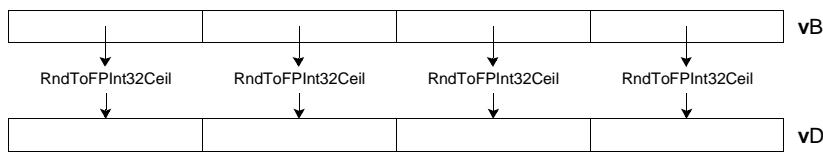
Note: The result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-99 shows the usage of the **vrfip** instruction. Each of the four elements in the vectors **vB** and **vD** is 32 bits in length.

Figure 6-99. **vrfip**—Round to Plus Infinity of Four Floating-Point Integer Elements (32-Bit)



vrfiz**vrfiz**

Vector Round to Floating-Point Integer toward Zero (0x1000 024A)

vrfiz**vD,vB**

Form: VX

04	vD	0 0 0 0 0	vB	586
0	5 6	10 11	15 16	20 21

```
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 register **vB** is rounded to a single-precision floating-point integer, using the rounding mode Round toward Zero, and placed into the corresponding word element of register **vD**.

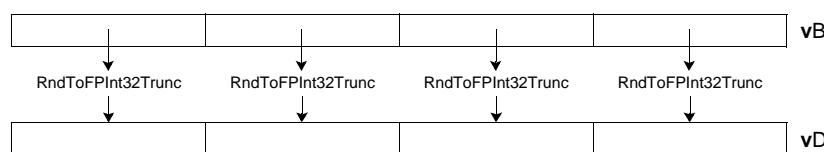
Note: The result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-100 shows the usage of the **vrfiz** instruction. Each of the four elements in the registers **vB** and **vD** is 32 bits in length.

Figure 6-100. vrfiz—Round-to-Zero of Four Floating-Point Integer Elements (32-Bit)



vrlb**vrlb**

Vector Rotate Left Integer Byte (0x1000 0004)

vrlb**vD,vA,vB**

Form: VX

04	vD	vA	vB	4
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 8
  sh ← (vB)i+5:i+7
  (vD)i:i+7 ← ROTL( (vA)i:i+7, sh)
end

```

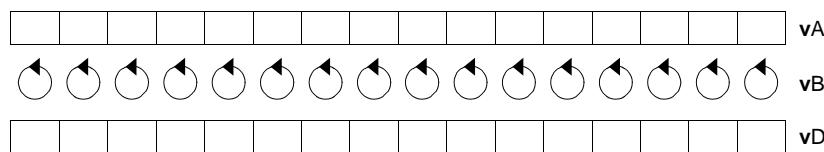
Each element is a byte. Each byte element in register **vA** is rotated left by the number of bits specified in the low-order 3 bits of the corresponding byte element in register **vB**. The result is placed into the corresponding byte element of register **vD**.

Other registers altered:

- None

Figure 6-101 shows the usage of the **vrlb** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-101. **vrlb**—Left Rotate of Sixteen Integer Elements (8-Bit)



Vector/SIMD Multimedia Extension Technology**vrlh**

Vector Rotate Left Integer Halfword (0x1000 0044)

vrlh**vD,vA,vB****vrlh**

Form: VX

04	vD	vA	vB	68
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  (vD)i:i+15 ← ROTL( (vA)i:i+15, sh)
end

```

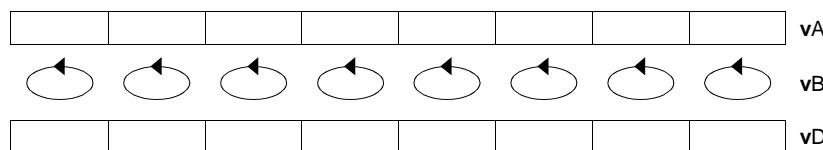
Each element is a halfword. Each halfword element in register **vA** is rotated left by the number of bits specified in the low-order 4 bits of the corresponding halfword element in register **vB**. The result is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- None

Figure 6-102 shows the usage of the **vrlh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-102. **vrlh**—Left Rotate of Eight Integer Elements (16-Bit)



vrlw

Vector Rotate Left Integer Word (0x1000 0084)

vrlw**vD,vA,vB****vrlw**

Form: VX

04	vD	vA	vB	132
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  sh ← (vB)i+27:i+31
  (vD)i:i+31 ← ROTL((vA)i:i+31, sh)
end

```

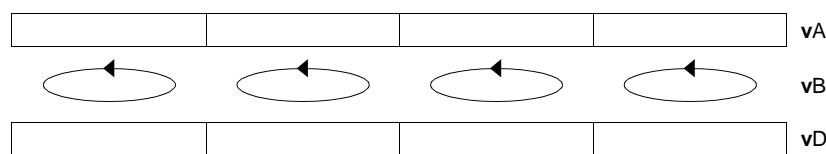
Each element is a word. Each word element in register **vA** is rotated left by the number of bits specified in the low-order 5 bits of the corresponding word element in register **vB**. The result is placed into the corresponding word element of register **vD**.

Other registers altered:

- None

Figure 6-103 shows the usage of the **vrlw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-103. **vrlw**—Left Rotate of Four Integer Elements (32-Bit)



vrsqrtefp**vrsqrtefp**

Vector Reciprocal Square Root Estimate Floating Point (0x1000 014A)

vrsqrtefp**vD,vB**

Form: VX

04	vD	0 0 0 0 0	vB	330
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  x ← (vB)i:i+31
  (vD)i:i+31 ← 1 ÷fp (Dfp(x))
end

```

The single-precision estimate of the reciprocal of the square root of each single-precision element in register **vB** is placed into the corresponding word element of register **vD**. 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 **vB**. Note that the value placed into the element of register **vD** may vary between implementations and between different executions on the same implementation. Operation with various special values of the element in register **vB** is summarized below.

Table 6-8. vrsqrtefp—Special Values

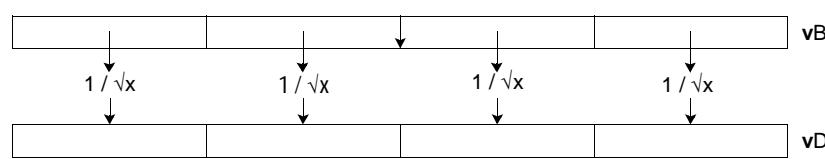
Value in vB	Result
-∞	QNaN
less than 0	QNaN
-0	-∞
+0	+∞
+∞	+0
NaN	QNaN

Other registers altered:

- None

Figure 6-104 shows the usage of the **vrsqrtefp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-104. vrsqrtefp—Reciprocal Square Root Estimate of Four Floating-Point Elements (32-Bit)



vsel

Vector Conditional Select (0x1000 002A)

vsel**vD,vA,vB,vC****vsel**

Form: VA

04	vD	vA	vB	vC	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 (vD)i ← (vB)i
  end

```

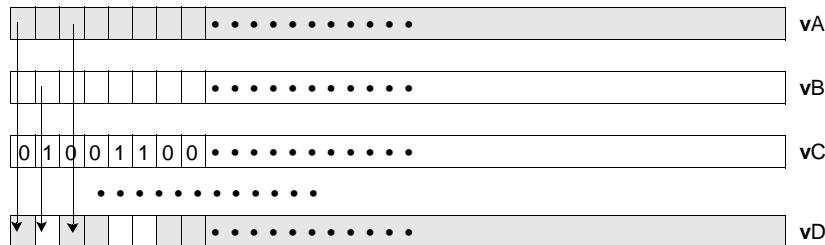
For each bit in register **vC** that contains the value ‘0’, the corresponding bit in register **vA** is placed into the corresponding bit of register **vD**. For each bit in register **vC** that contains the value ‘1’, the corresponding bit in register **vB** is placed into the corresponding bit of register **vD**.

Other registers altered:

- None

Figure 6-105 shows the usage of the **vsel** instruction. Each of the registers **vA**, **vB**, **vC**, and **vD** is 128 bits in length.

Figure 6-105. **vsel**—Bitwise Conditional Select of Vector Contents (128-bit)



vsl

Vector Shift Left (0x1000 01C4)

vsl**vD,vA,vB**

Form: VX

04	vD	vA	vB	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)
end
if t = 1 then (vD) <- (vA) <<ui sh
else (vD) <- undefined

```

Let sh be equal to the contents of bits [125-127] of register vB; sh is the shift count in bits ($0 \leq sh \leq 7$).

The contents of register vA are shifted left by sh bits. Bits shifted out of bit [0] are lost. Zeros are supplied to the vacated bits on the right. The result is placed into register vD.

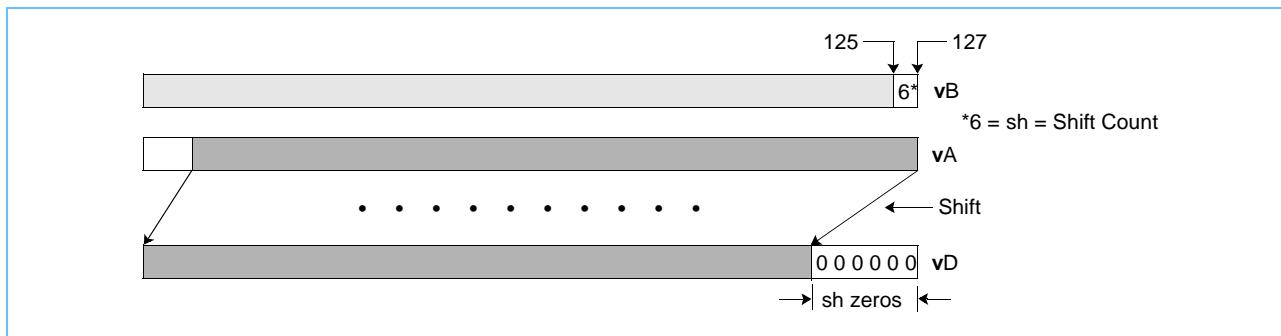
The contents of the low-order three bits of all byte elements in register vB must be identical to vB[125–127]; otherwise the value placed into register vD is undefined.

Other registers altered:

- None

Figure 6-106 shows the usage of the vsl instruction.

Figure 6-106. **vsl**—Shift Bits Left in Vector (128-Bit)



vslb

Vector Shift Left Integer Byte (0x1000 0104)

vslb**vD,vA,vB****vslb**

Form: VX

04	vD	vA	vB	260
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 8
  sh ← (vB)i+5:i+7
  (vD)i:i+7 ← (vA)i:i+7 << ui sh
end

```

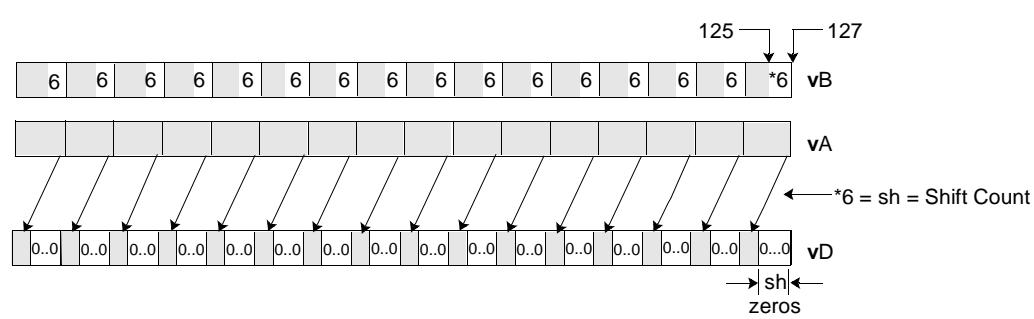
Each element is a byte. Each byte element in register **vA** is shifted left by the number of bits specified in the low-order 3 bits of the corresponding element in register **vB**. Bits shifted out of bit [0] of the byte element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding byte element of register **vD**.

Other registers altered:

- None

Figure 6-107 shows the usage of the **vslb** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-107. **vslb**—Shift Bits Left in Sixteen Integer Elements (8-Bit)



vsldoi

Vector Shift Left Double by Octet Immediate (0x1000 002C)

vsldoi**vD, vA, vB, SHB**

Form: VA

04	vD	vA	vB	0	SHB	44
0	5 6	10 11	15 16	20 21 22	25 26	31

$$(\mathbf{vD}) \leftarrow ((\mathbf{vA}) \parallel (\mathbf{vB})) \ll_{ui} (\text{SHB} \parallel 0b000)$$

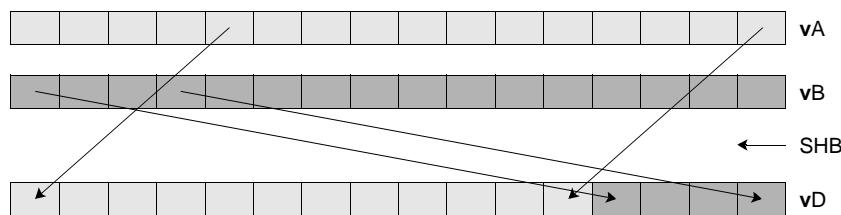
Let the source vector be the concatenation of the contents of register **vA** followed by the contents of register **vB**. Bytes SHB:SHB+15 of the source vector are placed into register **vD**.

Other registers altered:

- None

Figure 6-107 shows the usage of the **vsldoi** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-108. vsldoi—Shift Left by Bytes Specified



vslh

Vector Shift Left Integer Halfword (0x1000 0144)

vslh**vD,vA,vB****vslh**

Form: VX

04	vD	vA	vB	324
0	5 6	10 11	15 16	20 21 31

```

do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  (vD)i:i+15 ← (vA)i:i+15 <<ui sh
end

```

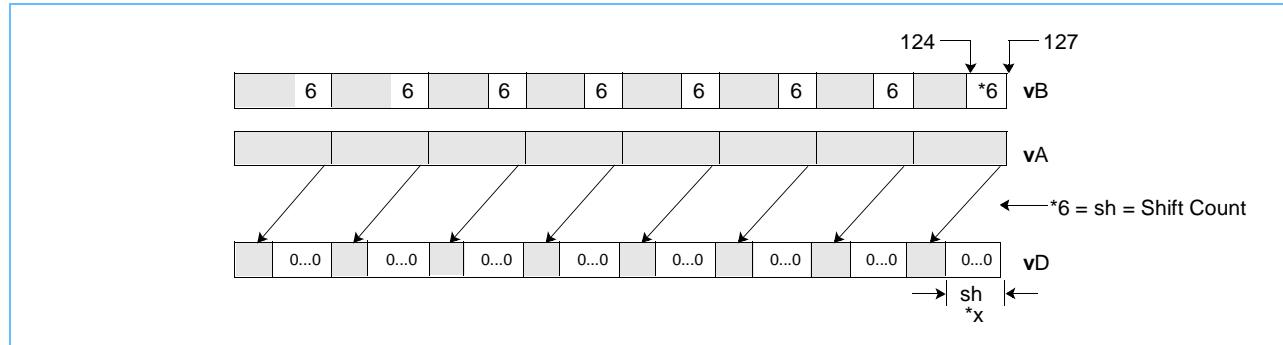
Each element is a halfword. Each halfword element in register vA is shifted left by the number of bits specified in the low-order 4 bits of the corresponding halfword element in register vB. Bits shifted out of bit [0] of the halfword element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding halfword element of register vD.

Other registers altered:

- None

Figure 6-109 shows the usage of the **vslh** instruction. Each of the eight elements in the registers vA, vB, and vD is 16 bits in length.

Figure 6-109. **vslh**—Shift Bits Left in Eight Integer Elements (16-Bit)



vslo

Vector Shift Left by Octet (0x1000 040C)

vslo**vD,vA,vB****vslo**

Form: VX

04	vD	vA	vB	1036
0	5 6	10 11	15 16	20 21

```
shb ← (vB)121:124
(vD) ← (vA) <<ui (shb || 0b000)
```

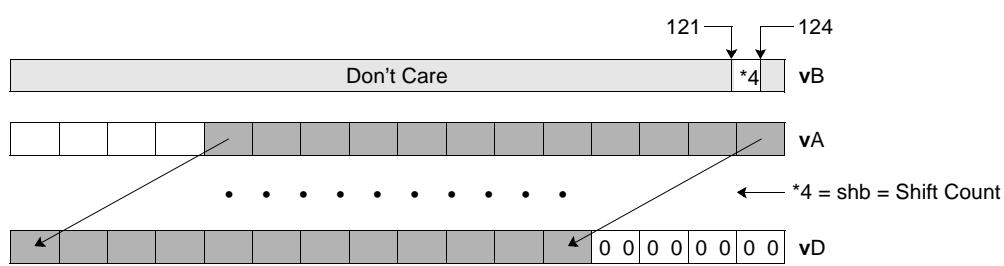
The contents of register **vA** are shifted left by the number of bytes specified in **vB**[121–124]. Bytes shifted out of byte [0] are lost. Zeros are supplied to the vacated bytes on the right. The result is placed into register **vD**.

Other registers altered:

- None

Figure 6-110 shows the usage of the **vslo** instruction.

Figure 6-110. **vslo**—Left Byte Shift of Vector (128-Bit)



vslw**vslw**

Vector Shift Left Integer Word (0x1000 0184)

vslw**vD,vA,vB**

Form: VX

04	vD	vA	vB	388
0	5 6	10 11	15 16	20 21 31

```
do i=0 to 127 by 32
  sh ← (vB)i+27:i+31
  (vD)i:i+31 ← (vA)i:i+31 <<ui sh
end
```

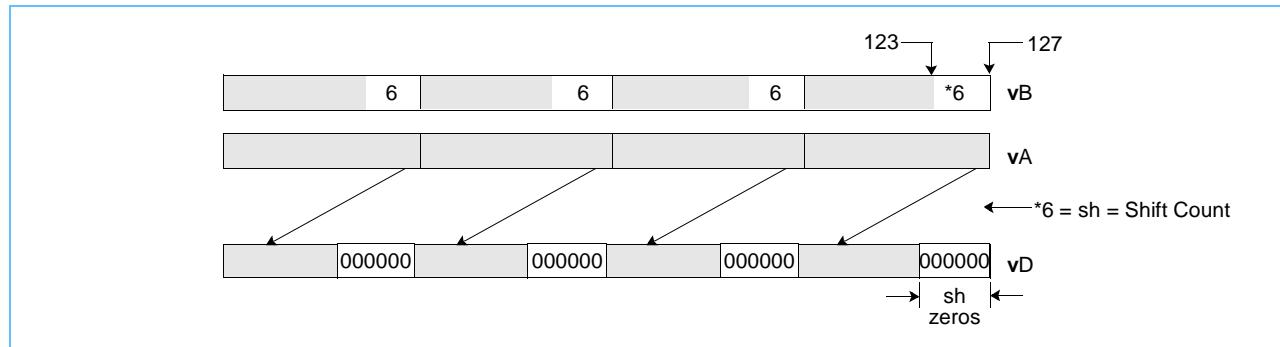
Each element is a word. Each word element in register **vA** is shifted left by the number of bits specified in the low-order 5 bits of the corresponding word element in register **vB**. Bits shifted out of bit [0] of the word element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding word element of register **vD**.

Other registers altered:

- None

Figure 6-111 shows the usage of the **vslw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-111. **vslw**—Shift Bits Left in Four Integer Elements (32-Bit)



vspltb

Vector Splat Byte (0x1000 020C)

vspltb**vD,vB,UIMM**

Form: VX

04	vD	/	UIMM	vB	524
0	5 6	10 11 12	15 16	20 21	31

```
b ← UIMM*8
do i=0 to 127 by 8
    (vD)i:i+7 ← (vB)b:b+7
end
```

The contents of byte element UIMM in register vB are replicated into each byte element of vD.

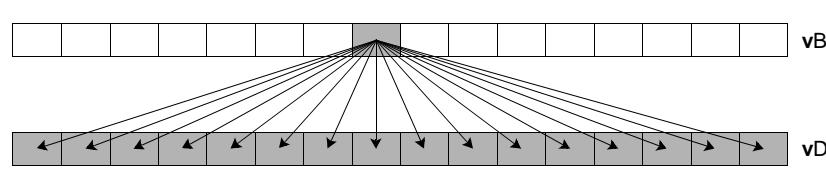
Other registers altered:

- 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 6-112 shows the usage of the **vspltb** instruction. Each of the sixteen elements in the registers vB and vD is 8 bits in length.

Figure 6-112. vspltb—Copy Contents to Sixteen Elements (8-Bit)



vsplth

Vector Splat Halfword (0x1000 024C)

vsplth**vD,vB,UIMM**

Form: VX

04	vD		//	UIMM	vB		588	
0	5	6	10	11 12 13	15	16	20	21

```
b ← UIMM*16
do i=0 to 127 by 16
  (vD)i:i+15 ← (vB)b:b+15
end
```

The contents of halfword element UIMM in register vB are replicated into each halfword element of register vD.

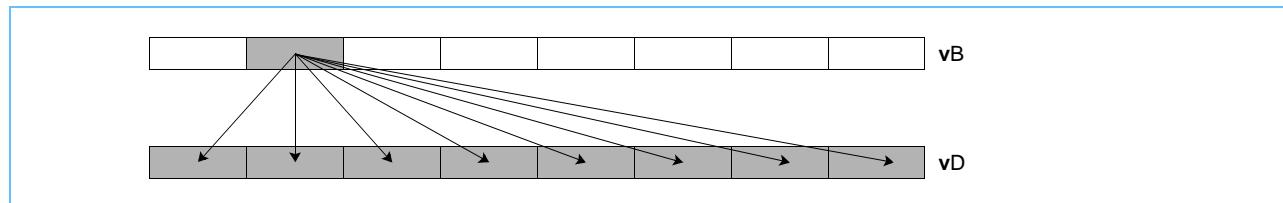
Other registers altered:

- 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 **vsplth** instruction. Each of the eight elements in the registers vB and vD is 16 bits in length.

Figure 6-113. **vsplth**—Copy Contents to Eight Elements (16-Bit)



vspltisb

Vector Splat Immediate Signed Byte (0x1000 030C)

vspltisb**vD,SIMM**

Form: VX

04	vD	SIMM	0 0 0 0	780
0	5 6	10 11	15 16	20 21

```
do i=0 to 127 by 8
  (vD)i:i+7 ← SignExtend (SIMM, 8)
end
```

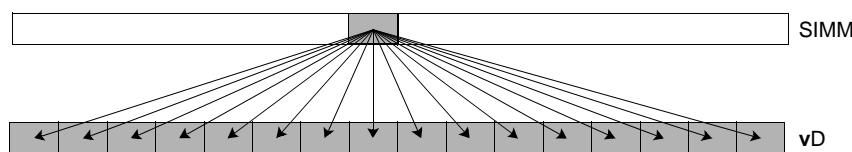
Each element of **vspltisb** is a byte. The value of the SIMM field, sign-extended to 8 bits, is replicated into each byte element of register **vD**.

Other registers altered:

- None

Figure 6-114 shows the usage of the **vspltisb** instruction. Each of the sixteen elements in the register **vD** is 8 bits in length.

Figure 6-114. **vspltisb**—Copy Value into Sixteen Signed Integer Elements (8-Bit)



vspltish**vspltish**

Vector Splat Immediate Signed Halfword (0x1000 034C)

vspltish**vD,SIMM**

Form: VX

04	vD	SIMM	0 0 0 0	844
0	5 6	10 11	15 16	20 21

```
do i=0 to 127 by 16
  (vD)i:i+15 ← SignExtend(SIMM,16)
end
```

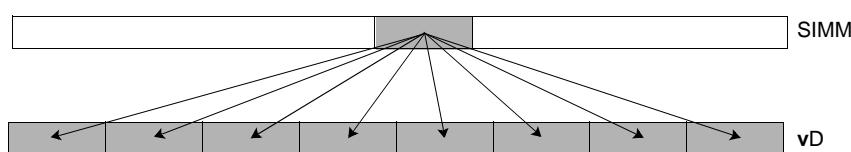
Each element of **vspltish** is a halfword. The value of the SIMM field, sign-extended to 16 bits, is replicated into each halfword element of register vD.

Other registers altered:

- None

Figure 6-115 shows the usage of the **vspltish** instruction. Each of the eight halfword elements in the registers vA, vB, and vD is 16 bits in length.

Figure 6-115. **vspltish**—Copy Value to Eight Signed Integer Elements (16-Bit)



vspltsw

Vector Splat Immediate Signed Word (0x1000 038C)

vspltsw**vD,SIMM**

Form: VX

04	vD	SIMM	0 0 0 0	908
0	5 6	10 11	15 16	20 21

```
do i=0 to 127 by 32
  (vD)i:i+31 ← SignExtend(SIMM,32)
end
```

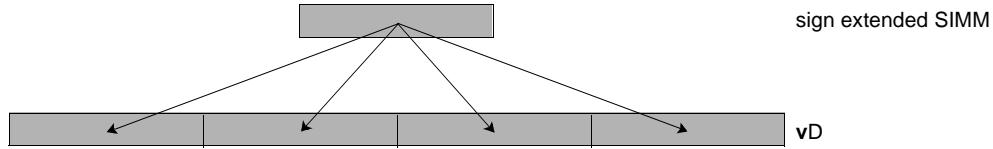
Each element of **vspltsw** is a word. The value of the SIMM field, sign-extended to 32 bits, is replicated into each element of register **vD**.

Other registers altered:

- None

Figure 6-116 shows the usage of the **vspltsw** instruction. Each of the four elements in the register **vD** is 32 bits in length.

Figure 6-116. **vspltsw**—Copy Value to Four Signed Elements (32-Bit)



vspltw

Vector Splat Word (0x1000 028C)

vspltw**vD,vB,UIMM**

Form: VX

04	vD	//	UIMM	vB	652
0	5 6	10 11	13 14 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 register vB are replicated into each word element of register vD.

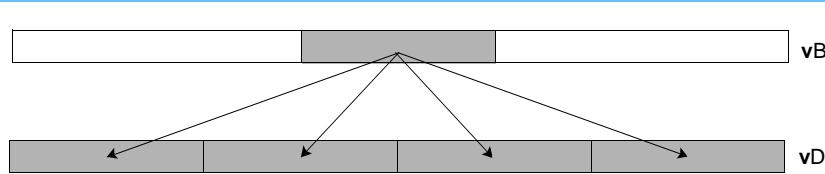
Other registers altered:

- 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 6-117 shows the usage of the **vspltw** instruction. Each of the four elements in the registers vA, vB, and vD is 32 bits in length.

Figure 6-117. **vspltw**—Copy contents to Four Elements (32-Bit)



Vsr

Vector Shift Right (0x1000 02C4)

vsr**vD,vA,vB****Vsr**

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)
end
if t = 1 then (vD) ← (vA) >>ui sh
else(vD) ← undefined

```

Let sh be equal to the contents of bits [125-127] of register vB; sh is the shift count in bits ($0 \leq sh \leq 7$). The contents of register vA 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 register vD.

The contents of the low-order three bits of all byte elements in register vB must be identical to vB[125-127]; otherwise the value placed into vD 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 vX left by the number of bits specified in vY and places the result into vZ.

```

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 (vW) || (vX) left by the number of bits specified in vY and places the high-order 128 bits of the result into vZ.

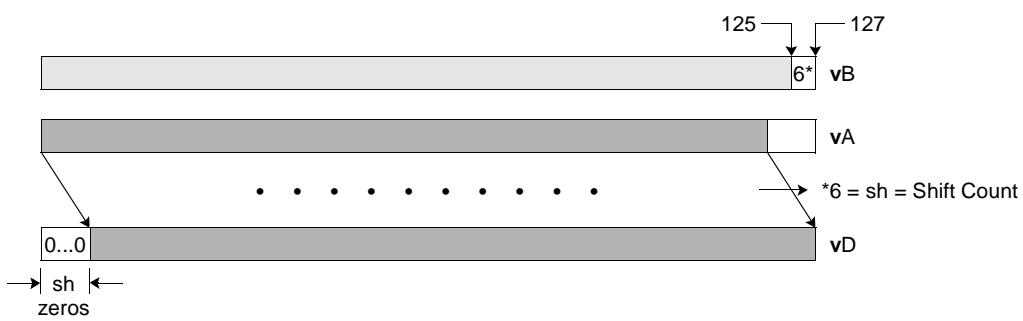
```

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 6-118 shows the usage of the **vsr** instruction. Each of the sixteen elements in the registers vA, vB, and vD is 8 bits in length.

Figure 6-118. **vsr**—Shift Bits Right for Vectors (128-Bit)



vsrab**vsrab**

Vector Shift Right Algebraic Byte (0x1000 0304)

vsrab**vD,vA,vB**

Form: VX

04	vD	vA	vB	772
0	5 6	10 11	15 16	20 21

31

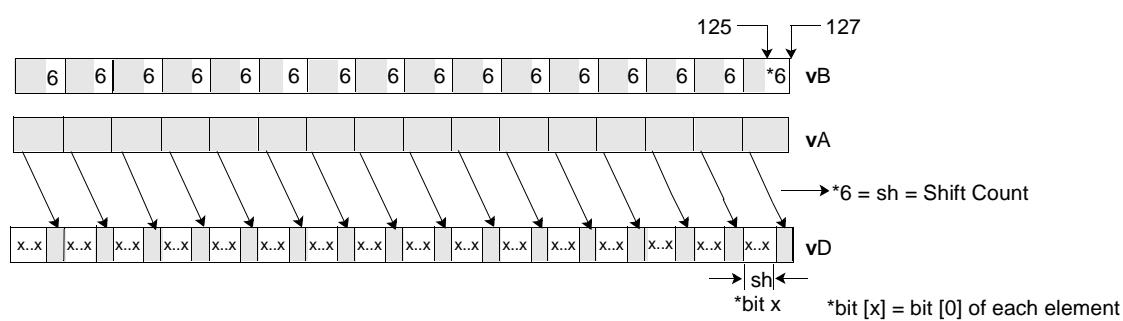
```
do i=0 to 127 by 8
  sh ← (vB)i+2:i+7
  (vD)i:i+7 ← (vA)i:i+7 >>si sh
end
```

Each element is a byte. Each byte element in register **vA** is shifted right by the number of bits specified in the low-order 3 bits of the corresponding byte element in register **vB**. Bits shifted out of bit [7] 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 byte element of register **vD**.

Other registers altered:

- None

Figure 6-119 shows the usage of the **vsrab** instruction. Each of the sixteen elements in the registers **vA** and **vD** is 8 bits in length.

Figure 6-119. **vsrab**—Shift Bits Right in Sixteen Integer Elements (8-Bit)

vsrah**vsrah**

Vector Shift Right Algebraic Halfword (0x1000 0344)

vsrah**vD,vA,vB**

Form: VX

04	vD	vA	vB	836
0	5 6	10 11	15 16	20 21 31

```

do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  (vD)i:i+15 ← (vA)i:i+15 >>si sh
end

```

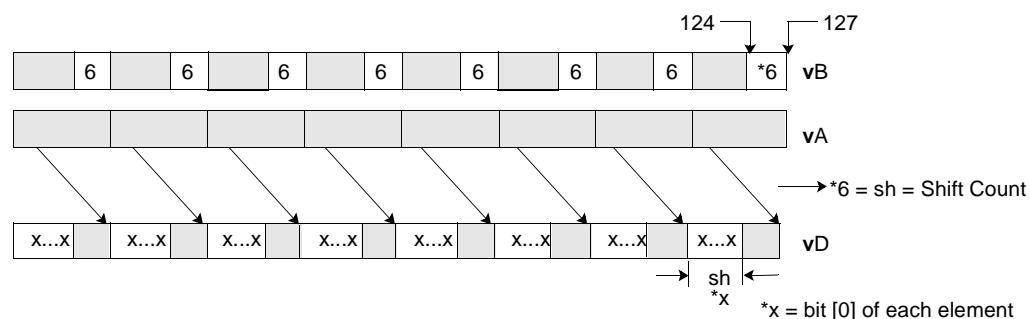
Each halfword element in register **vA** is shifted right by the number of bits specified in the low-order 4 bits of the corresponding halfword element in register **vB**. Bits shifted out of bit [15] of the halfword element are lost. Bit [0] of the halfword element is replicated to fill the vacated bits on the left. The result is placed into the corresponding halfword element of register **vD**.

Other registers altered:

- None

Figure 6-120 shows the usage of the **vsrah** instruction. Each of the eight elements in the registers **vA** and **vD** is 16 bits in length.

Figure 6-120. **vsrah**—Shift Bits Right for Eight Integer Elements (16-Bit)



Vector/SIMD Multimedia Extension Technology**vsraw**

Vector Shift Right Algebraic Word (0x1000 0384)

vsraw**vD,vA,vB**

Form: VX

04	vD	vA	vB	900
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  sh ← (vB)i+27:i+31
  (vD)i:i+31 ← (vA)i:i+31 >>si sh
end

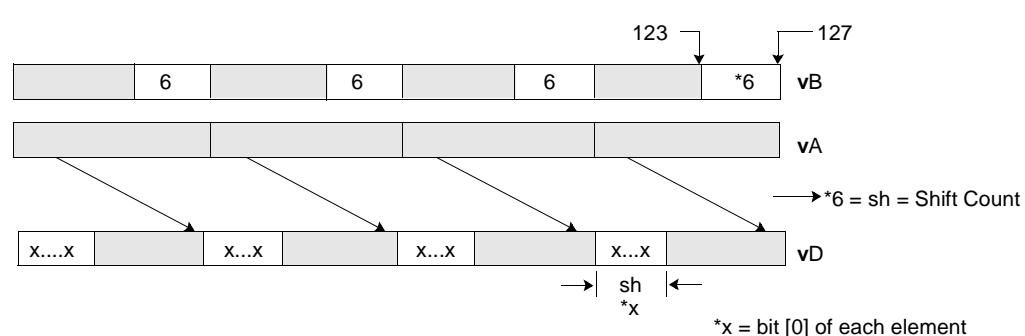
```

Each element is a word. Each element in register **vA** is shifted right by the number of bits specified in the low-order 5 bits of the corresponding element in register **vB**. 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 register **vD**.

Other registers altered:

- None

Figure 6-121 shows the usage of the **vsraw** instruction. Each of the four elements in the register **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-121. **vsraw**—Shift Bits Right in Four Integer Elements (32-Bit)

vSRB

Vector Shift Right Byte (0x1000 0204)

vSRB**vD,vA,vB****vSRB**

Form: VX

04	vD	vA	vB	516
0	5 6	10 11	15 16	20 21 31

```

do i=0 to 127 by 8
  sh ← (vB)i+5:i+7
  (vD)i:i+7 ← (vA)i:i+7 >>ui sh
end

```

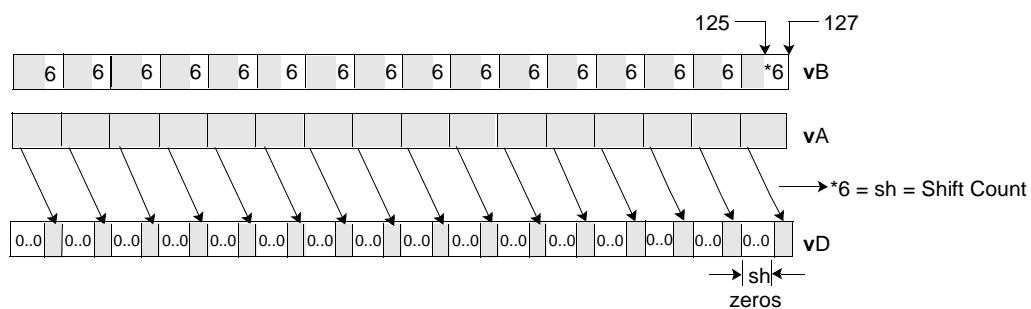
Each element is a byte. Each element in register **vA** is shifted right by the number of bits specified in the low-order 3 bits of the corresponding element in register **vB**. Bits shifted out of bit [7] of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of register **vD**.

Other registers altered:

- None

Figure 6-122 shows the usage of the **vSRB** instruction. Each of the sixteen elements in the registers **vA** and **vD** is 8 bits in length.

Figure 6-122. **vSRB**—Shift Bits Right in Sixteen Integer Elements (8-Bit)



Vector/SIMD Multimedia Extension Technology**vsrh**

Vector Shift Right Halfword (0x1000 0244)

vsrh**vD,vA,vB****vsrh**

Form: VX

04	vD	vA	vB	580
0	5 6	10 11	15 16	20 21 31

```

do i=0 to 127 by 16
  sh ← (vB)i+12:i+15
  (vD)i:i+15 ← (vA)i:i+15 >>ui sh
end

```

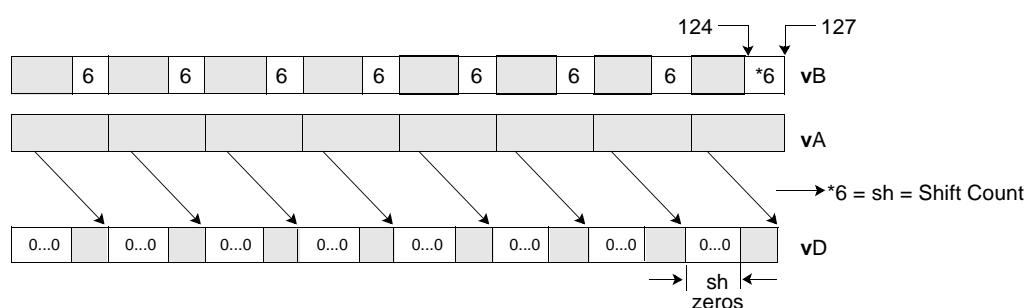
Each element is a halfword. Each element in register **vA** is shifted right by the number of bits specified in the low-order 4 bits of the corresponding element in register **vB**. Bits shifted out of bit [15] of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of register **vD**.

Other registers altered:

- None

Figure 6-123 shows the usage of the **vsrh** instruction. Each of the eight elements in the registers **vA** and **vD** is 16 bits in length.

Figure 6-123. **vsrh**—Shift Bits Right for Eight Integer Elements (16-Bit)

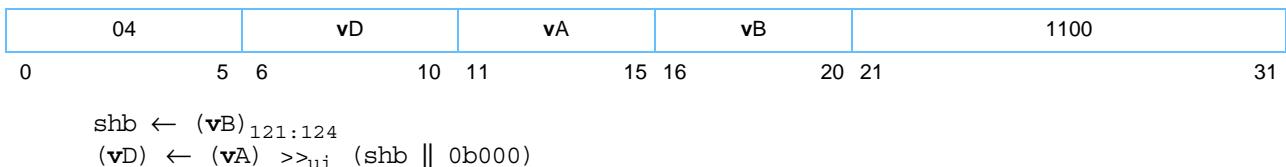


VSRO

Vector Shift Right by Octet (0x1000 044C)

vsro**vD,vA,vB****VSRO**

Form: VX

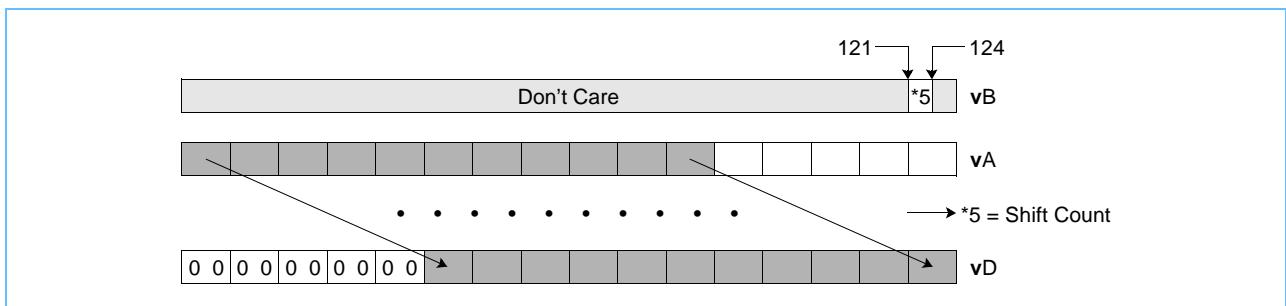


The contents of **vA** are shifted right by the number of bytes specified in **vB[121–124]**. Bytes shifted out of register **vA** are lost. Zeros are supplied to the vacated bytes on the left. The result is placed into register **vD**.

Other registers altered:

- None

Figure 6-124. **vsro**—Vector Shift Right Octet



Vector/SIMD Multimedia Extension Technology**VSRW**

Vector Shift Right Word (0x1000 0284)

vsrw**vD,vA,vB****VSRW**

Form: VX

04	vD	vA	vB	644
0	5 6	10 11	15 16	20 21 31

```
do i=0 to 127 by 32
  sh ← (vB)i+(27):i+31
  (vD)i:i+31 ← (vA)i:i+31 >>ui sh
end
```

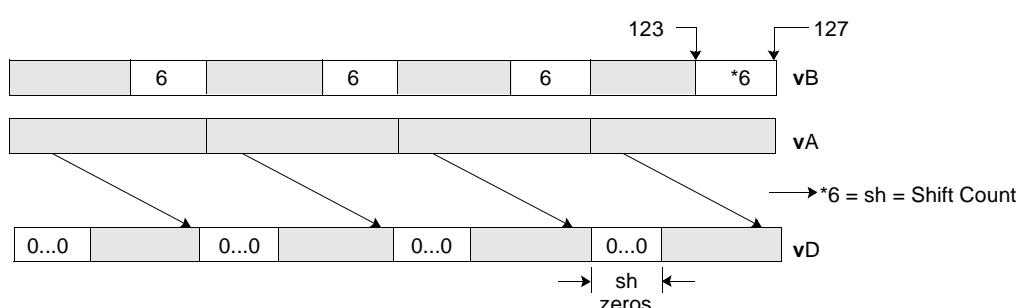
Each element is a word. Each element in register **vA** is shifted right by the number of bits specified in the low-order 5 bits of the corresponding element in register **vB**. Bits shifted out of bit [31] of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of register **vD**.

Other registers altered:

- None

Figure 6-125 shows the usage of the **vsrw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-125. **vsrw**—Shift Bits Right in Four Integer Elements (32-Bit)



vsbcuw

Vector Subtract Carryout Unsigned Word (0x1000 0580)

vsbcuw**vD,vA,vB**

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
    (vD)i:i+31 ← ZeroExtend(temp0, 32)
end

```

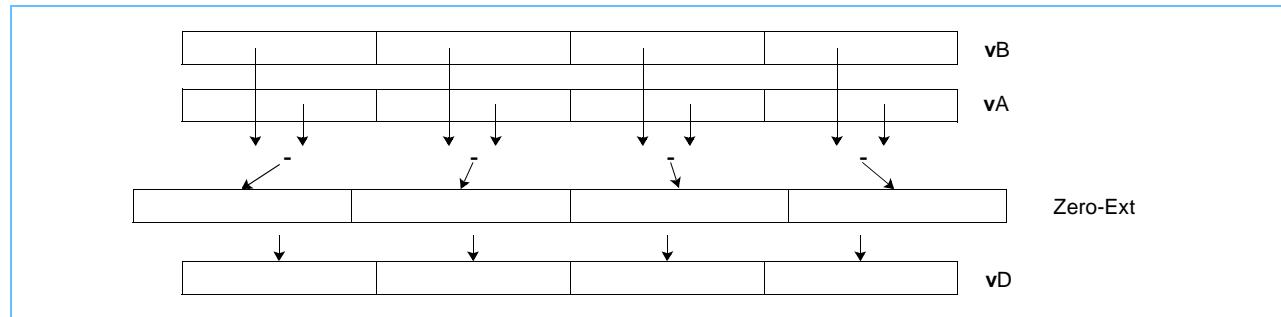
Each unsigned-integer word element in register **vB** is subtracted from the corresponding unsigned-integer word element in register **vA**. The complement of the borrow out of bit [0] of the 32-bit difference is zero-extended to 32 bits and placed into the corresponding word element of register **vD**.

Other registers altered:

- None

Figure 6-126 shows the usage of the **vsbcuw** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-126. **vsbcuw**—Subtract Carryout of Four Unsigned Integer Elements (32-Bit)



vsubfp

Vector Subtract Floating Point (0x1000 004A)

vsubfp**vD,vA,vB****vsubfp**

Form: VX

04	vD	vA	vB	74
0	5 6	10 11	15 16	20 21

```
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 register **vB** is subtracted from the corresponding single-precision floating-point word element in register **vA**. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of register **vD**.

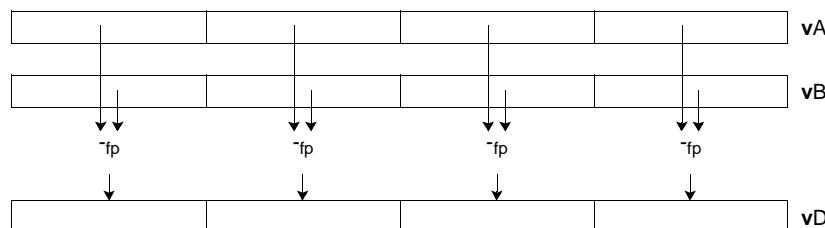
If VSCR[NJ] =‘1’, every denormalized operand element is truncated to a ‘0’ of the same sign before the operation is carried out, and each denormalized result element truncates to a ‘0’ of the same sign.

Other registers altered:

- None

Figure 6-127 shows the usage of the **vsubfp** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-127. vsubfp—Subtract Four Floating Point Elements (32-Bit)



vsubsbs

Vector Subtract Signed Byte Saturate (0x1000 0700)

vsubsbs**vD,vA,vB**

Form: VX

04	vD	vA	vB	1792
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+7 ← SIToSISat(temp0:8, 8)
end

```

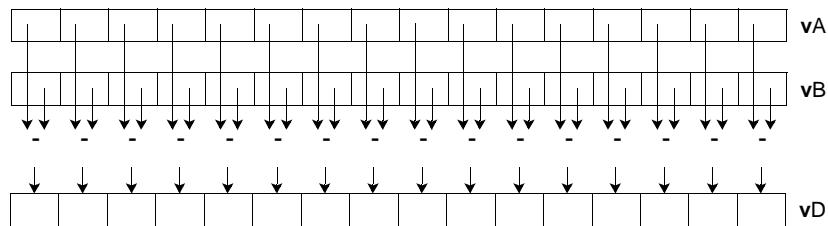
Each element is a byte. Each signed-integer element in register **vB** is subtracted from the corresponding signed-integer element in register **vA**. If the intermediate result is greater than (2^7-1) it saturates to (2^7-1) and if the intermediate result is less than (-2^7) , it saturates to (-2^7) . If saturation occurs, the SAT bit is set. The signed-integer result is placed into the corresponding element of register **vD**.

Other registers altered:

- SAT

Figure 6-128 shows the usage of the **vsubsbs** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-128. **vsubsbs**—Subtract Sixteen Signed Integer Elements (8-Bit)



vsubshs

Vector Subtract Signed Halfword Saturate (0x1000 0740)

vsubshs**vD,vA,vB**

Form: VX

04	vD	vA	vB	1856
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+15 ← SIToSIsat(temp0:16,16)
end

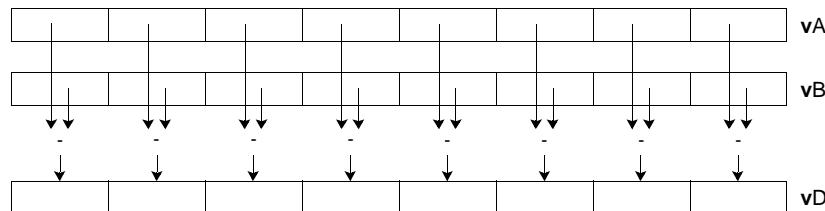
```

Each element is a halfword. Each signed-integer element in register vB is subtracted from the corresponding signed-integer element in register vA. If the intermediate result is greater than ($2^{15}-1$) it saturates to ($2^{15}-1$) and if the intermediate result 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 register vD.

Other registers altered:

- SAT

Figure 6-129 shows the usage of the **vsubshs** instruction. Each of the eight elements in the registers vA, vB, and vD is 16 bits in length.

Figure 6-129. **vsubshs**—Subtract Eight Signed Integer Elements (16-Bit)

vsbsws

Vector Subtract Signed Word Saturate (0x1000 0780)

vsbsws**vD,vA,vB**

Form: VX

04	vD	vA	vB	1920
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+31 ← SIToSISat(temp0:32, 32)
end

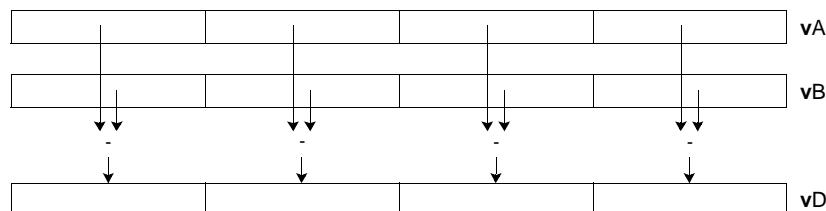
```

Each element is a word. Each signed-integer element in register vB is subtracted from the corresponding signed-integer element in register vA. If the intermediate result is greater than ($2^{31}-1$) it saturates to ($2^{31}-1$) and if the intermediate result 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 register vD.

Other registers altered:

- SAT

Figure 6-130 shows the usage of the **vsbsws** instruction. Each of the four elements in the registers vA, vB, and vD is 32 bits in length.

Figure 6-130. **vsbsws**—Subtract Four Signed Integer Elements (32-Bit)

vsububm

Vector Subtract Unsigned Byte Modulo (0x1000 0400)

vsububm**vD,vA,vB**

Form: VX

04	vD	vA	vB	1024
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 8
  (vD)i:i+7 ← (vA)i:i+7 +int - (vB)i:i+7 +int 1
end

```

Each element of **vsububm** is a byte. Each integer element in register **vB** is subtracted from the corresponding integer element in register **vA**. The integer result is placed into the corresponding element of register **vD**.

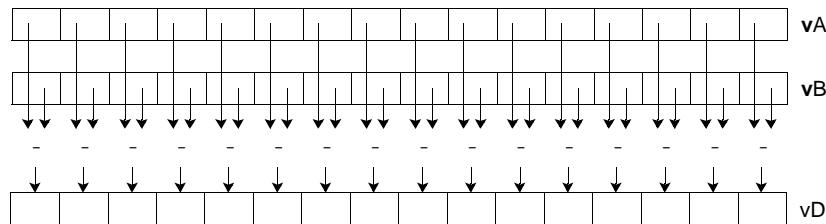
Other registers altered:

- None

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

Figure 6-131 shows the usage of the **vsububm** instruction. Each of the sixteen elements in the registers **vA**, **vB**, and **vD** is 8 bits in length.

Figure 6-131. vsububm—Subtract Sixteen Integer Elements (8-Bit)



vsububs

Vector Subtract Unsigned Byte Saturate (0x1000 0600)

vsububs**vD,vA,vB**

Form: VX

04	vD	vA	vB	1536
0	5 6	10 11	15 16	20 21

```

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 +int 1
    (vD)i:i+7 ← SIToUIsat(temp0:8, 8)
end

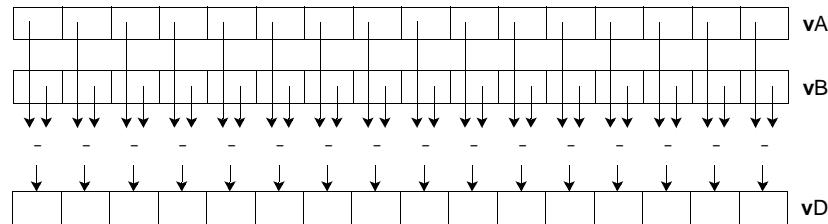
```

Each element is a byte. Each unsigned-integer element in register vB is subtracted from the corresponding unsigned-integer element in register vA. If the intermediate result is less than '0' it saturates to '0'. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding element of register vD.

Other registers altered:

- SAT

Figure 6-132 shows the usage of the **vsububs** instruction. Each of the sixteen elements in the registers vA, vB, and vD is 8 bits in length.

Figure 6-132. **vsububs**—Subtract Sixteen Unsigned Integer Elements (8-Bit)

vsubuhm

Vector Subtract Signed Halfword Modulo (0x1000 0440)

vsubuhm**vD,vA,vB**

Form: VX

04	vD	vA	vB	1088
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 16
  (vD)i:i+15 ← (vA)i:i+15 +int - (vB)i:i+15 +int 1
end

```

Each element is a halfword. Each integer element in register **vB** is subtracted from the corresponding integer element in register **vA**. The integer result is placed into the corresponding element of register **vD**.

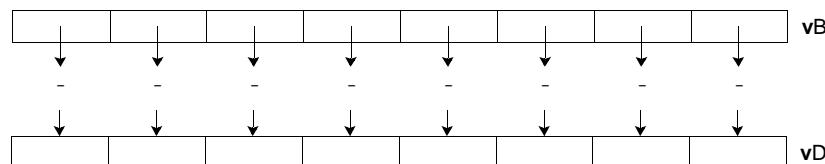
Other registers altered:

- None

Notes: **vsubuhm** instruction can be used for unsigned or signed integers.

Figure 6-133 shows the usage of the **vsubuhm** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-133. **vsubuhm**—Subtract Eight Integer Elements (16-Bit)



vsabuh

Vector Subtract Signed Halfword Saturate (0x1000 0640)

vsabuh**vD,vA,vB**

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
    (vD)i:i+15 ← SItouIsat(temp0:16,16)
end

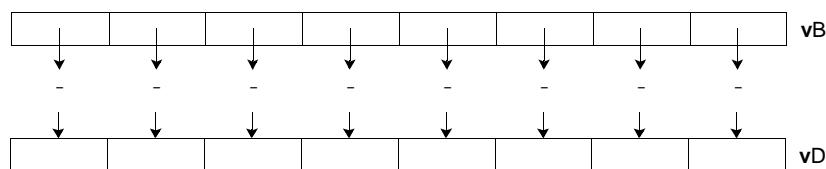
```

Each element is a halfword. Each unsigned-integer element in register **vB** is subtracted from the corresponding unsigned-integer element in register **vA**. If the intermediate result is less than '0' it saturates to '0'. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding element of register **vD**.

Other registers altered:

- SAT

Figure 6-134 shows the usage of the **vsabuh** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-134. **vsabuh**—Subtract Eight Signed Integer Elements (16-Bit)

vsubuwm

Vector Subtract Unsigned Word Modulo (0x1000 0480)

vsubuwm**vD,vA,vB**

Form: VX

04	vD	vA	vB	1152
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  (vD)i:i+31 ← (vA)i:i+31 +int - (vB)i:i+31 +int 1
end

```

Each element of **vsubuwm** is a word. Each integer element in register **vB** is subtracted from the corresponding integer element in register **vA**. The integer result is placed into the corresponding element of register **vD**.

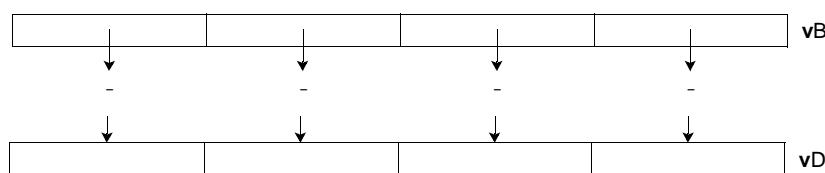
Other registers altered:

- None

Note: The **vsubuwm** instruction can be used for unsigned or signed integers.

Figure 6-135 shows the usage of the **vsubuwm** instruction. Each of the four elements in the registers **vA**, **vB**, and **vD** is 32 bits in length.

Figure 6-135. vsubuwm—Subtract Four Integer Elements (32-Bit)



vsbuws

Vector Subtract Unsigned Word Saturate (0x1000 0680)

vsbuws**vD,vA,vB**

Form: VX

04	vD	vA	vB	1664
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
    (vD)i:i+31 ← SIToUIsat(temp0:32, 32)
end

```

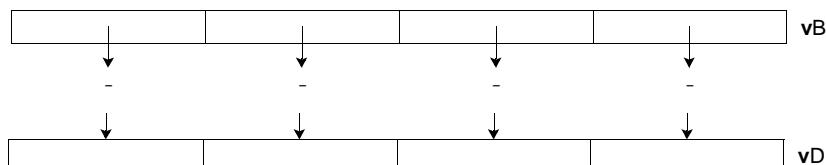
Each element is a word. Each unsigned-integer element in register vB is subtracted from the corresponding unsigned-integer element in register vA. If the intermediate result is less than '0' it saturates to '0'. If saturation occurs, the SAT bit is set. The unsigned-integer result is placed into the corresponding element of register vD.

Other registers altered:

- SAT

Figure 6-135 shows the usage of the **vsbuws** instruction. Each of the four elements in the registers vA, vB, and vD is 32 bits in length.

Figure 6-136. **vsbuws**—Subtract Four Signed Integer Elements (32-Bit)



vsumsws

Vector Sum Across Signed Word Saturate (0x1000 0788)

vsumsws**vD,vA,vB**

Form: VX

04	vD	vA	vB	1928
0	5 6	10 11	15 16	20 21

```

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

```

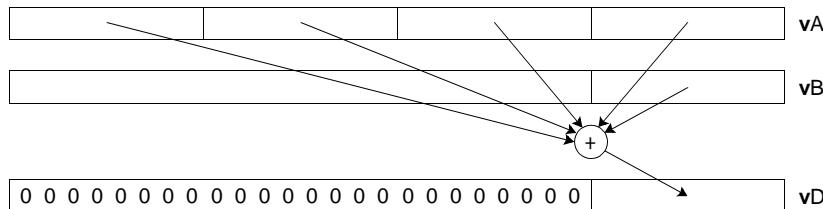
The signed-integer sum of the four signed-integer word elements in register vA is added to the signed-integer word element in bits of vB[96-127]. If the intermediate result is greater than ($2^{31}-1$) it saturates to ($2^{31}-1$) and if it is less than (-2^{31}) it saturates to (-2^{31}). If saturation occurs, the SAT bit is set. The signed-integer result is placed into bits vD[96–127]. Bits vD[0–95] are cleared.

Other registers altered:

- SAT

Figure 6-137 shows the usage of the **vsumsws** instruction. Each of the four elements in the registers vA, vB, and vD 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 (0x1000 0688)

vsum2sws**vD,vA,vB**

Form: VX

04	vD	vA	vB	1672
0	5 6	10 11	15 16	20 21

```

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
  (vD)i:i+63 ← 320 || SIToSIsat( temp0:33, 32)
end

```

The signed-integer sum of the first two signed-integer word elements in register vA is added to the signed-integer word element in vB[32–63]. If the intermediate result is greater than (2^{31} -1) it saturates to (2^{31} -1) and if the intermediate result 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 vD[32–63].

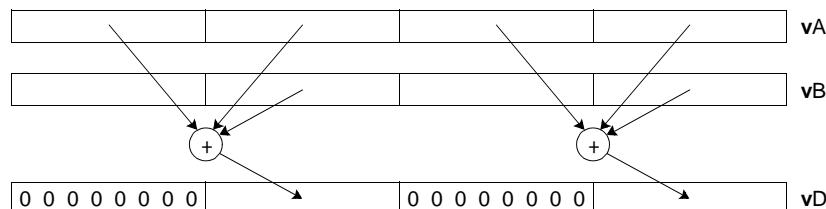
The signed-integer sum of the last two signed-integer word elements in register vA is added to the signed-integer word element in vB[96–127]. If the intermediate result is greater than (2^{31} -1) it saturates to (2^{31} -1) and if it is less than (- 2^{31}) it saturates to (- 2^{31}). If saturation occurs, the SAT bit is set. The signed-integer result is placed into vD[96–127]. The bits vD[0–31,64–95] are cleared to '0'.

Other registers altered:

- SAT

Figure 6-138 shows the usage of the **vsum2sws** instruction. Each of the four elements in the registers vA, vB, and vD 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 (0x1000 0708)

vsum4sbs**vD,vA,vB**

Form: VX

04	vD	vA	vB	1800
0	5 6	10 11	15 16	20 21

```

do i=0 to 127 by 32
  temp0:32 ← SignExtend( (vB)i:i+31, 33)
  do j=0 to 31 by 8
    temp0:32 ← temp0:32 + int SignExtend( (vA)i+j:i+j+7, 33)
  end
  (vD)i:i+31 ← SIToSISat( temp0:32, 32)
end

```

For each word element in register vB, the following operations are performed in the order shown:

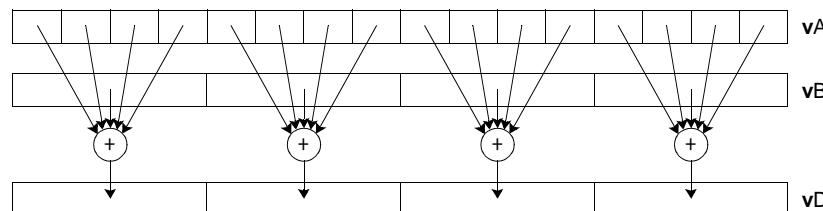
- The signed-integer sum of the four signed-integer byte elements contained in the corresponding word element of register vA is added to the signed-integer word element in register vB.
- If the intermediate result is greater than ($2^{31}-1$) it saturates to ($2^{31}-1$) and if it is less than (-2^{31}) it saturates to (-2^{31}). If saturation occurs, the SAT bit is set.
- The signed-integer result is placed into the corresponding word element of register vD.

Other registers altered:

- SAT

Figure 6-139 shows the usage of the **vsum4sbs** instruction. Each of the sixteen elements in the register vA, is 8 bits in length. Each of the four elements in the registers vB and vD is 32 bits in length.

Figure 6-139. **vsum4sbs**—Four Sums in the Integer Elements (32-Bit)



vsum4shs**vsum4shs**

Vector Sum Across Partial (1/4) Signed Halfword Saturate (0x1000 0648)

vsum4shs**vD,vA,vB**

Form: VX

04	vD	vA	vB	1608
0	5 6	10 11	15 16	20 21

```

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
    (vD)i:i+31 ← SIToSIsat( temp0:32, 32)
end

```

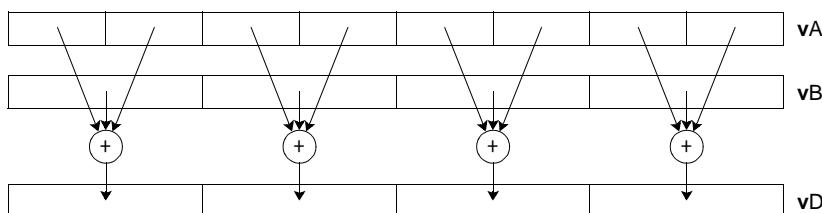
For each word element in register vB, the following operations are performed in the order shown:

- The signed-integer sum of the two signed-integer halfword elements contained in the corresponding word element of register vA is added to the signed-integer word element in vB.
- 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} . If saturation occurs, the SAT bit is set.
- The signed-integer result is placed into the corresponding word element of register vD.

Other registers altered:

- SAT

Figure 6-140 shows the usage of the **vsum4shs** instruction. Each of the eight elements in the register vA is 16 bits in length. Each of the four elements in the registers vB and vD 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 (0x1000 0608)

vsum4ubs**vD,vA,vB**

Form: VX

04	vD	vA	vB	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
  (vD)i:i+31 ← UItoUIsat(temp0:32, 32)
end

```

For each word element in vB, the following operations are performed in the order shown:

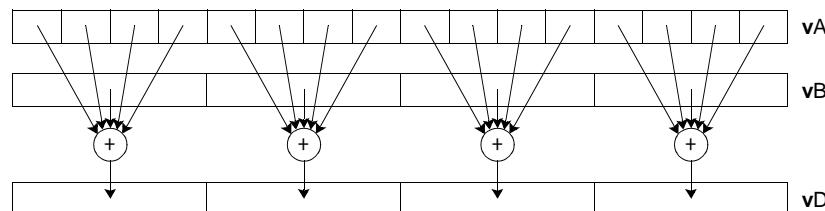
- The unsigned-integer sum of the four unsigned-integer byte elements contained in the corresponding word element of register vA is added to the unsigned-integer word element in register vB.
- If the intermediate result is greater than $(2^{32}-1)$ it saturates to $(2^{32}-1)$. If saturation occurs, the SAT bit is set.
- The unsigned-integer result is placed into the corresponding word element of vD.

Other registers altered:

- SAT

Figure 6-141 shows the usage of the **vsum4ubs** instruction. Each of the four elements in the register vA is 8 bits in length. Each of the four elements in the registers vB and vD is 32 bits in length.

Figure 6-141. **vsum4ubs**—Four Sums in the Integer Elements (32-Bit)



vupkhpx

Vector Unpack High Pixel16 (0x1000 034E)

vupkhpx**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	846
0	5 6	10 11	15 16	20 21

```

do i=0 to 63 by 16
  (vD)i*2:(i*2)+7 ← SignExtend((vB)i, 8)
  (vD)(i*2)+8:(i*2)+15 ← ZeroExtend((vB)i+1:i+5, 8)
  (vD)(i*2)+16:(i*2)+23 ← ZeroExtend((vB)i+6:i+10, 8)
  (vD)(i*2)+24:(i*2)+31 ← ZeroExtend((vB)i+11:i+15, 8)
end

```

Each halfword element in the high-order half of register **vB** is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of register **vD**.

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

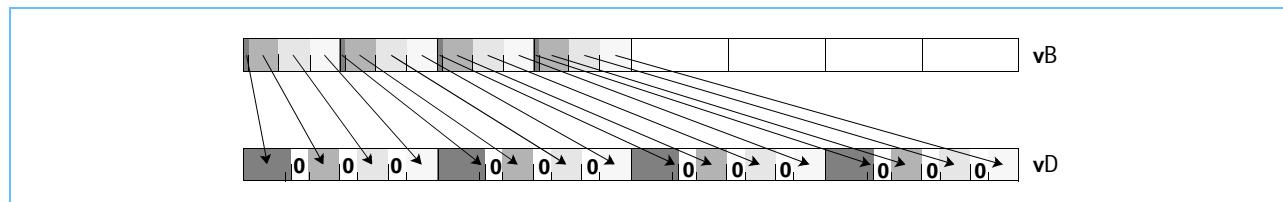
Other registers altered:

- 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** instruction. Each of the eight elements in the register **vB** is 16 bits in length. Each of the four elements in the register **vD** is 32 bits in length.

Figure 6-142. **vupkhpx**—Unpack High-Order Elements (16 bit) to Elements (32-Bit)



vupkhsb

Vector Unpack High Signed Byte (0x1000 020E)

vupkhsb**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	526
0	5 6	10 11	15 16	20 21

```
do i=0 to 63 by 8
  (vD)i*2:(i*2)+15 ← SignExtend( (vB)i:i+7,16)
end
```

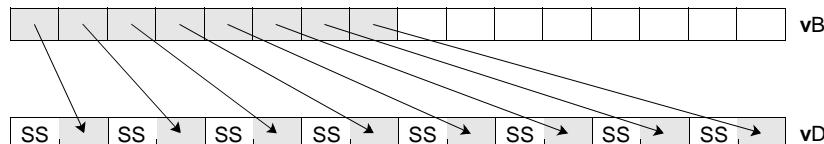
Each signed integer byte element in the high-order half of register **vB** is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight halfwords of register **vD**.

Other registers altered:

- None

*Figure 6-143 shows the usage of the **vupkhsb** instruction. Each of the sixteen elements in the register **vB** is 8 bits in length. Each of the eight elements in the register **vD** is 16 bits in length.*

*Figure 6-143. **vupkhsb**—Unpack High-Order Signed Integer Elements (8-Bit) to Signed Integer Elements (16-Bit)*



vupkhsh

Vector Unpack High Signed Halfword (0x1000 024E)

vupkhsh**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	590
0	5 6	10 11	15 16	20 21

```
do i=0 to 63 by 16
  (vD)_{i*2:(i*2)+31} ← SignExtend( (vB)_{i:i+15}, 32)
end
```

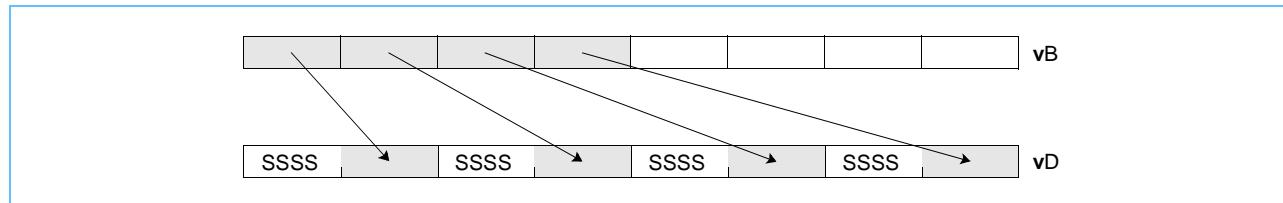
Each signed integer halfword element in the high-order half of register **vB** is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **vD**.

Other registers altered:

- None

Figure 6-144 shows the usage of the **vupkhsh** instruction. Each of the eight elements in the registers **vB** and **vD** is 16 bits in length.

Figure 6-144. **vupkhsh**—Unpack Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)



vupklpx

Vector Unpack Low Pixel16 (0x1000 03CE)

vupklpx**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	974
0	5 6	10 11	15 16	20 21

```

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 **vB** is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of register **vD**.

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

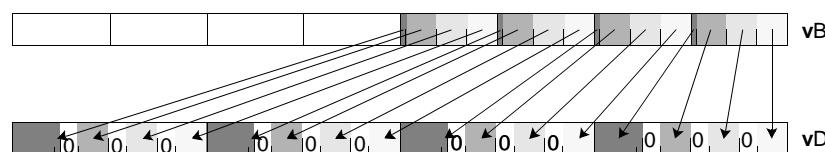
Other registers altered:

- 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** instruction. Each of the eight elements in register **vB** is 16 bits in length. Each of the four elements in the register **vD** is 32 bits in length.

Figure 6-145. **vupklpx**—Unpack Low-Order Elements (16-Bit) to Elements (32-Bit)



vupklsb

Vector Unpack Low Signed Byte (0x1000 028E)

vupklsb**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	654
0	5 6	10 11	15 16	20 21

```
do i=0 to 63 by 8
  (vD)i*2:(i*2)+15 ← SignExtend( (vB)i+64:i+71, 16)
end
```

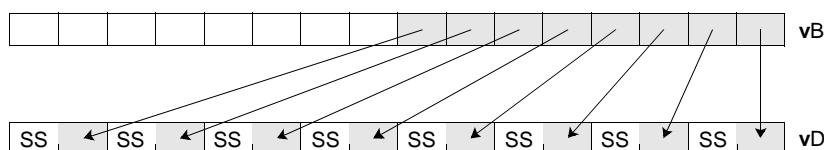
Each signed integer byte element in the low-order half of register **vB** is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight halfwords of register **vD**.

Other registers altered:

- None

Figure 6-146 shows the usage of the **vaddubs** instruction. Each of the sixteen elements in the registers **vB** and **vD** is 8 bits in length.

Figure 6-146. **vupklsb**—Unpack Low-Order Elements (8-Bit) to Elements (16-Bit)



vupklsh

Vector Unpack Low Signed Halfword (0x1000 02CE)

vupklsh**vD,vB**

Form: VX

04	vD	0 0 0 0	vB	718
0	5 6	10 11	15 16	20 21

```
do i=0 to 63 by 16
  (vD)i*2:(i*2)+31 ← SignExtend( (vB)i+64:i+79, 32)
end
```

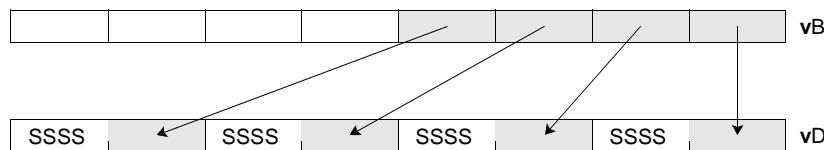
Each signed integer halfword element in the low-order half of register **vB** is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **vD**.

Other registers altered:

- None

Figure 6-147 shows the usage of the **vupklpx** instruction. Each of the eight elements in the registers **vA**, **vB**, and **vD** is 16 bits in length.

Figure 6-147. vupklsh—Unpack Low-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)



VXOR

Vector Logical XOR (0x1000 04C4)

vxor

vD,vA,vB

VXOR

Form: VX

04	vD	vA	vB	1220
0	5 6	10 11	15 16	20 21

$$(vD) \leftarrow (vA) \oplus (vB)$$

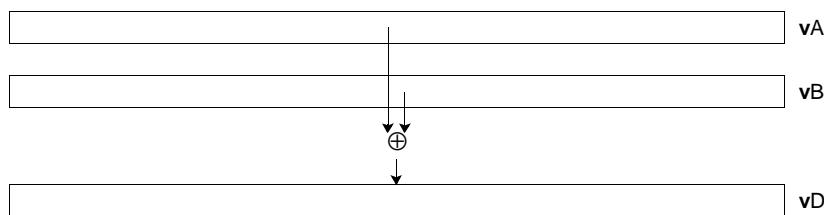
The contents of register **vA** are XORed with the contents of register **vB** and the result is placed into register **vD**.

Other registers altered:

- None

*Figure 6-148 shows the usage of the **vxor** instruction.*

Figure 6-148. vxor—Bitwise XOR (128-Bit)



Appendix A. Vector Processing Instruction Set Listings

This appendix lists the instruction set for the vector/SIMD multimedia extension 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, or optional, or both.

Note: 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 vector architecture in alphabetical order by mnemonic.

Key:

 Reserved bits

Table A-1. Complete Instruction List Sorted by Mnemonic (Page 1 of 5)

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
dss¹	31		A	0	0	STRM		0	0	0	0		0	0	0	0		822									0	
dssall¹	31		A	0	0	STRM		0	0	0	0		0	0	0	0		822									0	
dst¹	31		T	0	0	STRM			rA			rB						342									0	
dstst¹	31		T	0	0	STRM			rA			rB						374									0	
dststt¹	31		1	0	0	tag			rA			rB					11		22								0	
dstt¹	31		1	0	0	tag			rA			rB						0									0	
lvebx	31			vD					rA			rB					7									0		
lvehx	31			vD					rA			rB					39									0		
lvewx	31			vD					rA			rB					71									0		
lvlx	31			vD					rA			rB					519									0		
lvlxl	31			vD					rA			rB					775									0		
lvrx	31			vD					rA			rB					551									0		
lvrxl	31			vD					rA			rB					807									0		
lvsl	31			vD					rA			rB					6									0		
lvsr	31			vD					rA			rB					38									0		
lvx	31			vD					rA			rB					103									0		
lvxl	31			vD					rA			rB					359									0		
mfvscr	04			vD					0	0	0	0		0	0	0	0		1540									
mtvscr	04			///					0	0	0	0		vB				1604										
stvebx	31			vS					rA			rB					135									0		
stvehx	31			vS					rA			rB					167									0		
stvewx	31			vS					rA			rB					199									0		
stvlx	31			vS					rA			rB					647									0		
stvlxl	31			vS					rA			rB					903									0		
stvrx	31			vS					rA			rB					679									0		

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.

**Vector/SIMD Multimedia Extension Technology***Table A-1. Complete Instruction List Sorted by Mnemonic* (Page 2 of 5)

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
stvrxl	31			vS			rA			rB																	935	0
stvx	31			vS			rA			rB																	231	0
stvxl	31			vS			rA			rB																	487	0
vaddcuw	04			vD			vA			vB																	384	0
vaddfp	04			vD			vA			vB																	10	0
vaddsbs	04			vD			vA			vB																	768	0
vaddshs	04			vD			vA			vB																	832	0
vaddsws	04			vD			vA			vB																	896	0
vaddubm	04			vD			vA			vB																	0	0
vaddubs	04			vD			vA			vB																	512	0
vadduhm	04			vD			vA			vB																	64	0
vadduhs	04			vD			vA			vB																	576	0
vadduwm	04			vD			vA			vB																	128	0
vadduws	04			vD			vA			vB																	640	0
vand	04			vD			vA			vB																	1028	0
vandc	04			vD			vA			vB																	1092	0
vavgsb	04			vD			vA			vB																	1282	0
vavgsh	04			vD			vA			vB																	1346	0
vavgsw	04			vD			vA			vB																	1410	0
vavgub	04			vD			vA			vB																	1026	0
vavguh	04			vD			vA			vB																	1090	0
vavguw	04			vD			vA			vB																	1154	0
vcfsx	04			vD				UIMM			vB															842		
vcfux	04			vD				UIMM			vB															778		
vcmplibpx	04			vD			vA			vB		Rc														966		
vcmpeqfx	04			vD			vA			vB		Rc														198		
vcmpequbx	04			vD			vA			vB		Rc														6		
vcmpequhx	04			vD			vA			vB		Rc														70		
vcmpequwx	04			vD			vA			vB		Rc														134		
vcmpgefpx	04			vD			vA			vB		Rc														454		
vcmpgtfpx	04			vD			vA			vB		Rc														710		
vcmpgtsbx	04			vD			vA			vB		Rc														774		
vcmpgtshx	04			vD			vA			vB		Rc														838		
vcmpgtswx	04			vD			vA			vB		Rc														902		
vcmpgtubx	04			vD			vA			vB		Rc														518		
vcmpgtuhx	04			vD			vA			vB		Rc														582		
vcmpgtuwx	04			vD			vA			vB		Rc														646		
vctsxs	04			vD				UIMM			vB															970		
vctuxs	04			vD				UIMM			vB															906		
vexptefp	04			vD				0 0 0 0 0			vB														394			

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Table A-1. Complete Instruction List Sorted by Mnemonic (Page 3 of 5)

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
vlogefp	04				vD			0	0	0	0	0			vB								458					
vmaddfp	04				vD					vA				vB				vC						46				
vmaxfp	04				vD					vA				vB											1034			
vmaxsb	04				vD					vA				vB											258			
vmaxsh	04				vD					vA				vB											322			
vmaxsw	04				vD					vA				vB											386			
vmaxub	04				vD					vA				vB											2			
vmaxuh	04				vD					vA				vB											66			
vmaxuw	04				vD					vA				vB											130			
vhaddshs	04				vD					vA				vB			vC								32			
vhaddshs	04				vD					vA				vB			vC								33			
vminfp	04				vD					vA				vB											1098			
vminsbt	04				vD					vA				vB											770			
vminsh	04				vD					vA				vB											834			
vminsw	04				vD					vA				vB											898			
vminub	04				vD					vA				vB											514			
vminuh	04				vD					vA				vB											578			
vminuw	04				vD					vA				vB											642			
vladduhm	04				vD					vA				vB			vC								34			
vmrghb	04				vD					vA				vB											12			
vmrghh	04				vD					vA				vB											76			
vmrghw	04				vD					vA				vB											140			
vmrglb	04				vD					vA				vB											268			
vmrglh	04				vD					vA				vB											332			
vmrglw	04				vD					vA				vB											396			
vmsummbm	04				vD					vA				vB			vC								37			
vmsumshm	04				vD					vA				vB			vC								40			
vmsumshs	04				vD					vA				vB			vC								41			
vmsumubm	04				vD					vA				vB			vC								36			
vmsumuham	04				vD					vA				vB			vC								38			
vmsumuhs	04				vD					vA				vB			vC								39			
vmulesb	04				vD					vA				vB											776			
vmulesh	04				vD					vA				vB											840			
vmuleub	04				vD					vA				vB											520			
vmuleuh	04				vD					vA				vB											584			
vmulosb	04				vD					vA				vB											264			
vmulosh	04				vD					vA				vB											328			
vmuloub	04				vD					vA				vB											8			
vmulouh	04				vD					vA				vB											72			
vnmsubfp	04				vD					vA				vB			vC								47			

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.

Vector/SIMD Multimedia Extension Technology*Table A-1. Complete Instruction List Sorted by Mnemonic* (Page 4 of 5)

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
vnor	04		vD		vA		vB																			1284		
vor	04		vD		vA		vB																			1156		
vperm	04		vD		vA		vB											vC								43		
vpkpx	04		vD		vA		vB																			782		
vpkshss	04		vD		vA		vB																			398		
vpkshus	04		vD		vA		vB																			270		
vpkswss	04		vD		vA		vB																			462		
vpkuhum	04		vD		vA		vB																			14		
vpkuhus	04		vD		vA		vB																			142		
vpkuwum	04		vD		vA		vB																			78		
vpkuwus	04		vD		vA		vB																			206		
vrefp	04		vD		0 0 0 0 0				vB																266			
vrfim	04		vD		0 0 0 0 0				vB																714			
vrfin	04		vD		0 0 0 0 0				vB																522			
vrfip	04		vD		0 0 0 0 0				vB																650			
vrfiz	04		vD		0 0 0 0 0				vB																586			
vrlb	04		vD		vA		vB																			4		
vrlh	04		vD		vA		vB																			68		
vrlw	04		vD		vA		vB																			132		
vrsqrtefp	04		vD		0 0 0 0 0				vB																330			
vsel	04		vD		vA		vB									vC									42			
vsl	04		vD		vA		vB																			452		
vslb	04		vD		vA		vB																			260		
vstdoi	04		vD		vA		vB									0	SH									44		
vslh	04		vD		vA		vB																			324		
vslo	04		vD		vA		vB																			1036		
vslw	04		vD		vA		vB																			388		
vspltb	04		vD		UIMM				vB																524			
vsplth	04		vD		UIMM				vB																588			
vspltisb	04		vD		SIMM				vB																780			
vspltish	04		vD		SIMM				0 0 0 0 0																	844		
vspltisw	04		vD		SIMM				0 0 0 0 0																	908		
vspltw	04		vD		UIMM				vB																652			
vsr	04		vD		vA		vB																			708		
vsrab	04		vD		vA		vB																			772		
vsrah	04		vD		vA		vB																			836		
vsraw	04		vD		vA		vB																			900		
vsrb	04		vD		vA		vB																			516		
vsrh	04		vD		vA		vB																			580		
vsro	04		vD		vA		vB																			1100		

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Table A-1. Complete Instruction List Sorted by Mnemonic (Page 5 of 5)

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
vsrw	04		vD		vA		vB																			644		
vsubcuw	04		vD		vA		vB																			1408		
vsubfp	04		vD		vA		vB																			74		
vsubbsb	04		vD		vA		vB																			1792		
vsubshs	04		vD		vA		vB																			1856		
vsubsws	04		vD		vA		vB																			1920		
vsububm	04		vD		vA		vB																			1024		
vsububus	04		vD		vA		vB																			1536		
vsubuhm	04		vD		vA		vB																			1088		
vsubuhhs	04		vD		vA		vB																			1600		
vsubuwmm	04		vD		vA		vB																			1152		
vsubuws	04		vD		vA		vB																			1664		
vsumsws	04		vD		vA		vB																			1928		
vsum2sws	04		vD		vA		vB																			1672		
vsum4sbs	04		vD		vA		vB																			1800		
vsum4shs	04		vD		vA		vB																			1608		
vsum4ubs	04		vD		vA		vB																			1544		
vupkhpx	04		vD		0 0 0 0 0		vB																		846			
vupkhsb	04		vD		0 0 0 0 0		vB																		526			
vupkhsh	04		vD		0 0 0 0 0		vB																		590			
vupklpx	04		vD		0 0 0 0 0		vB																		974			
vupklsb	04		vD		0 0 0 0 0		vB																		654			
vupklsh	04		vD		0 0 0 0 0		vB																		718			
vxor	04		vD		vA		vB																			1220		

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.

Vector/SIMD Multimedia Extension Technology**A.2 Instructions Sorted by Opcode**

Figure A-2 lists the vector instructions grouped by opcode.

Key:



Table A-2. Instructions Sorted by Opcode (Page 1 of 6)

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	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		vA		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	
vadduwmm	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	

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Table A-2. Instructions Sorted by Opcode (Page 2 of 6)

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
vsubuh	000100				vD					vA			vB												110 0100 0000			
vsubuw	000100				vD					vA			vB												110 1000 0000			
vsubbs	000100				vD					vA			vB												111 0000 0000			
vsubsh	000100				vD					vA			vB												111 0100 0000			
vsubsw	000100				vD					vA			vB												111 1000 0000			
vmaxub	000100				vD					vA			vB												000 0000 0010			
vmaxuh	000100				vD					vA			vB												000 0100 0010			
vmaxuw	000100				vD					vA			vB												000 1000 0010			
vmaxsb	000100				vD					vA			vB												001 0000 0010			
vmaxsh	000100				vD					vA			vB												001 0100 0010			
vmaxsw	000100				vD					vA			vB												001 1000 0010			
vminub	000100				vD					vA			vB												010 0000 0010			
vminuh	000100				vD					vA			vB												010 0100 0010			
vminuw	000100				vD					vA			vB												010 1000 0010			
vminsbt	000100				vD					vA			vB												011 0000 0010			
vminsh	000100				vD					vA			vB												011 0100 0010			
vminsw	000100				vD					vA			vB												011 1000 0010			
vavgub	000100				vD					vA			vB												100 0000 0010			
vavguh	000100				vD					vA			vB												100 0100 0010			
vavguw	000100				vD					vA			vB												100 1000 0010			
vavgsbt	000100				vD					vA			vB												101 0000 0010			
vavgsh	000100				vD					vA			vB												101 0100 0010			
vavgsw	000100				vD					vA			vB												101 1000 0010			
vrlb	000100				vD					vA			vB												000 0000 0100			
vrlh	000100				vD					vA			vB												000 0100 0100			
vrlw	000100				vD					vA			vB												000 1000 0100			
vslb	000100				vD					vA			vB												001 0000 0100			
vslh	000100				vD					vA			vB												001 0100 0100			
vslw	000100				vD					vA			vB												001 1000 0100			
vsl	000100				vD					vA			vB												001 1100 0100			
vsrh	000100				vD					vA			vB												010 0000 0100			
vsrw	000100				vD					vA			vB												010 0100 0100			
vsr	000100				vD					vA			vB												010 1100 0100			
vsrab	000100				vD					vA			vB												011 0000 0100			

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Vector/SIMD Multimedia Extension Technology

Table A-2. Instructions Sorted by Opcode (Page 3 of 6)

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
vsrah	000100		vD		vA		vB										011 0100 0100											
vsraw	000100		vD		vA		vB										011 1000 0100											
vand	000100		vD		vA		vB										100 0000 0100											
vandc	000100		vD		vA		vB										100 0100 0100											
vor	000100		vD		vA		vB										100 1000 0100											
vxor	000100		vD		vA		vB										100 1100 0100											
vnor	000100		vD		vA		vB										101 0000 0100											
mfvscr	000100		vD		0 0 0 0 0		0 0 0 0 0										110 0000 0100											
mtvscr	000100		0 0 0 0 0		0 0 0 0 0		vB										110 0100 0100											
vcmpequbx	000100		vD		vA		vB		Rc								00 0000 0110											
vcmpequhx	000100		vD		vA		vB		Rc								00 0100 0110											
vcmpequwx	000100		vD		vA		vB		Rc								00 1000 0110											
vcmpeqfpfx	000100		vD		vA		vB		Rc								00 1100 0110											
vcmpgefpfx	000100		vD		vA		vB		Rc								01 1100 0110											
vcmpgtubx	000100		vD		vA		vB		Rc								10 0000 0110											
vcmpgtuhx	000100		vD		vA		vB		Rc								10 0100 0110											
vcmpgtuwx	000100		vD		vA		vB		Rc								10 1000 0110											
vcmpgtfpx	000100		vD		vA		vB		Rc								10 1100 0110											
vcmpgtsbx	000100		vD		vA		vB		Rc								11 0000 0110											
vcmpgtsbx	000100		vD		vA		vB		Rc								11 0100 0110											
vcmpgtswx	000100		vD		vA		vB		Rc								11 1000 0110											
vcmpbfpx	000100		vD		vA		vB		Rc								11 1100 0110											
vmuloub	000100		vD		vA		vB										000 0000 1000											
vmulouh	000100		vD		vA		vB										000 0100 1000											
vmulosb	000100		vD		vA		vB										001 0000 1000											
vmulosh	000100		vD		vA		vB										001 0100 1000											
vmuleub	000100		vD		vA		vB										010 0000 1000											
vmuleuh	000100		vD		vA		vB										010 0100 1000											
vmulesb	000100		vD		vA		vB										011 0000 1000											
vmulesh	000100		vD		vA		vB										011 0100 1000											
vsum4ubs	000100		vD		vA		vB										110 0000 1000											
vsum4sbs	000100		vD		vA		vB										111 0000 1000											
vsum4shs	000100		vD		vA		vB										110 0100 1000											
vsum2sws	000100		vD		vA		vB										110 1000 1000											
vsumsws	000100		vD		vA		vB										111 1000 1000											

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Table A-2. Instructions Sorted by Opcode (Page 4 of 6)

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
vaddfp	000100				vD						vA			vB										000 0000 1010				
vsubfp	000100				vD						vA			vB											000 0100 1010			
vrefp	000100				vD						0 0 0 0 0			vB										001 0000 1010				
vrsqrtefp	000100				vD						0 0 0 0 0			vB										001 0100 1010				
vexptefp	000100				vD						0 0 0 0 0			vB										001 1000 1010				
vlogefp	000100				vD						0 0 0 0 0			vB										001 1100 1010				
vrfin	000100				vD						0 0 0 0 0			vB										010 0000 1010				
vrfiz	000100				vD						0 0 0 0 0			vB										010 0100 1010				
vrfip	000100				vD						0 0 0 0 0			vB										010 1000 1010				
vrfim	000100				vD						0 0 0 0 0			vB										010 1100 1010				
vcfux	000100				vD						UIMM			vB										011 0000 1010				
vcfsx	000100				vD						UIMM			vB										011 0100 1010				
vctuxs	000100				vD						UIMM			vB										011 1000 1010				
vctsxs	000100				vD						UIMM			vB										011 1100 1010				
vmaxfp	000100				vD						vA			vB										100 0000 1010				
vminfp	000100				vD						vA			vB										100 0100 1010				
vmrghb	000100				vD						vA			vB										000 0000 1100				
vmrghh	000100				vD						vA			vB										000 0100 1100				
vmrghw	000100				vD						vA			vB										000 1000 1100				
vmrglb	000100				vD						vA			vB										001 0000 1100				
vmrglh	000100				vD						vA			vB										001 0100 1100				
vmrglw	000100				vD						vA			vB										001 1000 1100				
vspltb	000100				vD						UIMM			vB										010 0000 1100				
vsplth	000100				vD						UIMM			vB										010 0100 1100				
vspltw	000100				vD						UIMM			vB										010 1000 1100				
vspltsb	000100				vD						SIMM		0 0 0 0 0										011 0000 1100					
vspltish	000100				vD						SIMM		0 0 0 0 0										011 0100 1100					
vspltisw	000100				vD						SIMM		0 0 0 0 0										011 1000 1100					
vslo	000100				vD						vA			vB										100 0000 1100				
vsro	000100				vD						vA			vB										100 0100 1100				
vpkuhum	000100				vD						vA			vB										000 0000 1110				
vpkuwum	000100				vD						vA			vB										000 0100 1110				
vpkuhus	000100				vD						vA			vB										000 1000 1110				
vpkuwus	000100				vD						vA			vB										000 1100 1110				
vpkshus	000100				vD						vA			vB										001 0000 1110				

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.

Vector/SIMD Multimedia Extension Technology**Table A-2. Instructions Sorted by Opcode** (Page 5 of 6)

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
vpkswus	000100																											
vpkshss	000100																											
vpkswss	000100																											
vupkhsb	000100																											
vupkhsh	000100																											
vupklsb	000100																											
vupklsh	000100																											
vpkpx	000100																											
vupkhpx	000100																											
vupklpx	000100																											
lvsl	011111																											0
lvsr	011111																											0
dst¹	011111	T	0 0		STRM																						0	
dstt¹	011111	1	0 0 0		tag																						0	
dstst¹	011111	T	0 0		STRM																						0	
dststt¹	011111	1	0 0 0		tag																						0	
dss¹	011111	A	0 0		STRM																						0	
dssall¹	011111	A	0 0		STRM																						0	
lvebx	011111				vd																							0
lvehx	011111				vd																							0
lviewx	011111				vd																							0
lvx	011111				vd																							0
lvxl	011111				vd																							0
stvebx	011111				vs																							0
stvehx	011111				vs																							0
stviewx	011111				vs																							0
stvx	011111				vs																							0
stvxl	011111				vs																							0
lvlx	011111				vd																							0
lvlxl	011111				vd																							0
lvrx	011111				vd																							0
lvrxl	011111				vd																							0
stvlx	011111				vs																							0

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Table A-2. Instructions Sorted by Opcode (Page 6 of 6)

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
stvlxl	011111				vS				rA			rB						11 1000 0111								0		
stvrx	011111				vS				rA			rB						10 1010 0111								0		
stvr xl	011111				vS				rA			rB						11 1010 0111								0		

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.

Vector/SIMD Multimedia Extension Technology**A.3 Instructions Sorted by Form**

Table A-3 through *Table A-6* list the vector instructions grouped by form.

Key:

 Reserved bits
Table A-3. VA-Form

OPCD	vD	vA	vB	vC	XO	
OPCD	vD	vA	vB	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		vD		vA		vB		vC																				32
vmhraddshs	04		vD		vA		vB		vC																				33
vmladduhm	04		vD		vA		vB		vC																				34
vmsumubm	04		vD		vA		vB		vC																				36
vmsummbm	04		vD		vA		vB		vC																				37
vmsumuhm	04		vD		vA		vB		vC																				38
vmsumuhs	04		vD		vA		vB		vC																				39
vmsumshm	04		vD		vA		vB		vC																				40
vmsumshs	04		vD		vA		vB		vC																				41
vsel	04		vD		vA		vB		vC																				42
vperm	04		vD		vA		vB		vC																				43
vsldoi	04		vD		vA		vB	0	SH																				44
vmaddfp	04		vD		vA		vB		vC																				46
vnmsubfp	04		vD		vA		vB		vC																				47



Table A-4. VX-Form

OPCD	vD	vA	vB	XO	
OPCD	vD	0 0 0 0 0	0 0 0 0 0	XO	0
OPCD	0 0 0 0 0	0 0 0 0 0	vB	XO	0
OPCD	vD	0 0 0 0 0	vB	XO	
OPCD	vD	UIMM	vB	XO	
OPCD	vD	SIMM	0 0 0 0 0	XO	

Name	0	5	6	7	8	9	10	11	12	13	14	15	Specific Instructions													
													16	17	18	19	20	21	22	23	24	25	26	27	28	29
vaddubm	04		vD			vA			vB															0		
vadduhm	04		vD			vA			vB															64		
vadduwm	04		vD			vA			vB															128		
vaddcuw	04		vD			vA			vB															384		
vaddubs	04		vD			vA			vB															512		
vadduhs	04		vD			vA			vB															576		
vadduws	04		vD			vA			vB															640		
vaddsbs	04		vD			vA			vB															768		
vaddshs	04		vD			vA			vB															832		
vaddsws	04		vD			vA			vB															896		
vsububm	04		vD			vA			vB															1024		
vsubuhm	04		vD			vA			vB															1088		
vsubuwm	04		vD			vA			vB															1152		
vsubcuw	04		vD			vA			vB															1408		
vsububs	04		vD			vA			vB															1536		
vsubuhs	04		vD			vA			vB															1600		
vsubuws	04		vD			vA			vB															1664		
vsubsbs	04		vD			vA			vB															1792		
vsubshs	04		vD			vA			vB															1856		
vsubsws	04		vD			vA			vB															1920		
vmaxub	04		vD			vA			vB															2		
vmaxuh	04		vD			vA			vB															66		
vmaxuw	04		vD			vA			vB															130		
vmaxsb	04		vD			vA			vB															258		
vmaxsh	04		vD			vA			vB															322		
vmaxsw	04		vD			vA			vB															386		
vminub	04		vD			vA			vB															514		
vminuh	04		vD			vA			vB															578		


Vector/SIMD Multimedia Extension Technology

Name	0	Specific Instructions																											
		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	
vminuw	04	vD		vA		vB																						642	
vminsbt	04	vD		vA		vB																						770	
vminsh	04	vD		vA		vB																						834	
vminsw	04	vD		vA		vB																						898	
vavgub	04	vD		vA		vB																						1026	
vavguh	04	vD		vA		vB																						1090	
vavguw	04	vD		vA		vB																						1154	
vavgsbt	04	vD		vA		vB																						1282	
vavgsht	04	vD		vA		vB																						1346	
vavgswt	04	vD		vA		vB																						1410	
vrlb	04	vD		vA		vB																						4	
vrlh	04	vD		vA		vB																						68	
vrlw	04	vD		vA		vB																						132	
vslb	04	vD		vA		vB																						260	
vslh	04	vD		vA		vB																						324	
vslw	04	vD		vA		vB																						388	
vsl	04	vD		vA		vB																						452	
vsrb	04	vD		vA		vB																						516	
vsrh	04	vD		vA		vB																						580	
vsrw	04	vD		vA		vB																						644	
vsr	04	vD		vA		vB																						708	
vsrab	04	vD		vA		vB																						772	
vsrah	04	vD		vA		vB																						836	
vsraw	04	vD		vA		vB																						900	
vand	04	vD		vA		vB																						1028	
vandc	04	vD		vA		vB																						1092	
vor	04	vD		vA		vB																						1156	
vnor	04	vD		vA		vB																						1284	
mfvscr	04	vD	0 0 0 0 0		0 0 0 0 0		vB																						1540
mtvscr	04	0 0 0 0 0		0 0 0 0 0		vB																							1604
vmuloub	04	vD		vA		vB																							8
vmulouh	04	vD		vA		vB																							72
vmulosb	04	vD		vA		vB																							264
vmulosh	04	vD		vA		vB																							328
vmuleub	04	vD		vA		vB																							520
vmuleuh	04	vD		vA		vB																							584
vmulesb	04	vD		vA		vB																							776



Name	0	Specific Instructions																											
		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	
vmulesh	04		vD		vA		vB																						840
vsum4ubs	04		vD		vA		vB																						1544
vsum4sbs	04		vD		vA		vB																						1800
vsum4shs	04		vD		vA		vB																						1608
vsum2sws	04		vD		vA		vB																						1672
vsumsws	04		vD		vA		vB																						1928
vaddfp	04		vD		vA		vB																						10
vsubfp	04		vD		vA		vB																						74
vrefp	04		vD		0 0 0 0 0					vB																		266	
vrsqrtefp	04		vD		0 0 0 0 0					vB																		330	
vexptefp	04		vD		0 0 0 0 0					vB																		394	
vlogefp	04		vD		0 0 0 0 0					vB																		458	
vrfin	04		vD		0 0 0 0 0					vB																		522	
vrfiz	04		vD		0 0 0 0 0					vB																		586	
vrfip	04		vD		0 0 0 0 0					vB																		650	
vrfim	04		vD		0 0 0 0 0					vB																		714	
vcfux	04		vD		UIMM					vB																			778
vcfsx	04		vD		UIMM					vB																			842
vctuxs	04		vD		UIMM					vB																			906
vctsxs	04		vD		UIMM					vB																			970
vmaxfp	04		vD		vA					vB																			1034
vminfp	04		vD		vA					vB																			1098
vmrghb	04		vD		vA					vB																			12
vmrghh	04		vD		vA					vB																			76
vmrghw	04		vD		vA					vB																			140
vmrglb	04		vD		vA					vB																			268
vmrglh	04		vD		vA					vB																			332
vmrglw	04		vD		vA					vB																			396
vspltb	04		vD		UIMM					vB																			524
vsplth	04		vD		UIMM					vB																			588
vspltw	04		vD		UIMM					vB																			652
vspltisb	04		vD		SIMM					0 0 0 0 0																			780
vspltish	04		vD		SIMM					0 0 0 0 0																			844
vspltisw	04		vD		SIMM					0 0 0 0 0																			908
vslo	04		vD		vA					vB																			1036
vsro	04		vD		vA					vB																			1100
vpkuhum	04		vD		vA					vB																			14



Vector/SIMD Multimedia Extension Technology

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	Specific Instructions							
vpkuwum	04	vD		vA		vB																		78												
vpkuhus	04	vD		vA		vB																142														
vpkuwus	04	vD		vA		vB														206																
vpkshus	04	vD		vA		vB												270																		
vpkswus	04	vD		vA		vB												334																		
vpkshss	04	vD		vA		vB												398																		
vpkswss	04	vD		vA		vB												462																		
vupkhsb	04	vD		0 0 0 0 0		vB												526																		
vupkhsh	04	vD		0 0 0 0 0		vB												590																		
vupklsb	04	vD		0 0 0 0 0		vB												654																		
vupklsh	04	vD		0 0 0 0 0		vB												718																		
vpkpx	04	vD		vA		vB						12				782																				
vupkhpx	04	vD		0 0 0 0 0		vB												846																		
vupklpx	04	vD		0 0 0 0 0		vB												974																		
vxor	04	vD		vA		vB												1220																		



Table A-5. X-Form

OPCD	vD			vA	vB	XO			0
OPCD	vS			vA	vB	XO			0
OPCD	T	0 0	STRM	A	B	XO			0

Name	0	5	6	7	8	9	10	Specific Instructions												31									
								11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
dstst¹	31	T	0 0	STRM			rA		rB																	0			
dststt¹	31	1	0 0 0	tag		rA		rB										11			22					0			
lvebx	31		vD			rA		rB											7							0			
lvehx	31		vD			rA		rB											39							0			
lviewx	31		vD			rA		rB											71							0			
lvlx	31		vD			rA		rB											519							0			
lvlxl	31		vD			rA		rB											775							0			
lvrx	31		vD			rA		rB											551							0			
lvrxl	31		vD			rA		rB											807							0			
lvsl	31		vD			rA		rB											6							0			
lvsr	31		vD			rA		rB											38							0			
lvx	31		vD			rA		rB											103							0			
lvxl	31		vD			rA		rB											359							0			
stvebx	31		vS			rA		rB											135							0			
stvehx	31		vS			rA		rB											167							0			
stviewx	31		vS			rA		rB											199							0			
stvlx	31		vS			rA		rB											647							0			
stvxl	31		vS			rA		rB											903							0			
stvrx	31		vS			rA		rB											679							0			
stvrl	31		vS			rA		rB											935							0			
stvx	31		vS			rA		rB											231							0			
stvxl	31		vS			rA		rB											487							0			

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.

Vector/SIMD Multimedia Extension Technology*Table A-6. VXR-Form*

	OPCD	vD	vA	vB	Rc	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	vD	vA	vB	Rc	
vcmpbfpx	04				Rc	966	
vcmpfqpx	04				Rc	198	
vcmequbx	04				Rc	6	
vcmequhx	04				Rc	70	
vcmequwx	04				Rc	134	
vcmpgefpx	04				Rc	454	
vcmpgtfp	04				Rc	710	
vcmpgtsbx	04				Rc	774	
vcmpgtshx	04				Rc	838	
vcmpgtswx	04				Rc	902	
vcmpgtubx	04				Rc	518	
vcmpgtuhx	04				Rc	582	
vcmpgtuwx	04				Rc	646	

A.4 Instruction Set Legend

Table A-7 provides general information about the vector instruction set such as the architectural level, privilege level, and form.

Table A-7. Vector Instruction Set Legend (Page 1 of 5)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
dststt¹		X				VX
dstt¹		X				VX
lvebx	X					X
lvehx	X					X
lviewx	X					X
lvlx	X					X
lvlxl	X					X
lvrx	X					X
lvrxl	X					X
lvsl	X					X
lvsr	X					X
lvx	X					X
lvxl	X					X
mfvscr	X					VX
mtvscr	X					VX
stvebx	X					X
stvehx	X					X
stviewx	X					X
stvlx	X					X
stvlxl	X					X
stvrx	X					X
stvrxl	X					X
stvx	X					X
stvxl	X					X
vaddcuw	X					VX
vaddfp	X					VX
vaddsbs	X					VX
vaddshs	X					VX
vaddsws	X					VX
vaddubm	X					VX
vaddubs	X					VX

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Vector/SIMD Multimedia Extension Technology

Table A-7. Vector Instruction Set Legend (Page 2 of 5)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
vadduhm	X					VX
vadduhs	X					VX
vadduwm	X					VX
vadduws	X					VX
vand	X					VX
vandc	X					VX
vavgsb	X					VX
vavgsh	X					VX
vavgsw	X					VX
vavgub	X					VX
vavguh	X					VX
vavguw	X					VX
vcfux	X					VX
vcfsx	X					VX
vcmpbfpx	X					VXR
vcmpeqfx	X					VXR
vcmpequbx	X					VXR
vcmpequhx	X					VXR
vcmpequwx	X					VXR
vcmpgefp	X					VXR
vcmpgtfp	X					VXR
vcmpgtsbx	X					VXR
vcmpgtshx	X					VXR
vcmpgtswx	X					VXR
vcmpgtubx	X					VXR
vcmpgtuhx	X					VXR
vcmpgtuw	X					VXR
vctsxs	X					VX
vctuxs	X					VX
vexptefp	X					VX
vlogefp	X					VX
vmaddfp	X					VA
vmaxfp	X					VX
vmaxsb	X					VX
vmaxsh	X					VX

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Table A-7. Vector Instruction Set Legend (Page 3 of 5)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
vmaxsw	X					VX
vmaxub	X					VX
vmaxuh	X					VX
vmaxuw	X					VX
vmhaddshs	X					VA
vmhraddshs	X					VA
vminfp	X					VX
vminsbt	X					VX
vminsh	X					VX
vminsw	X					VX
vminub	X					VX
vminuh	X					VX
vminuw	X					VX
vmladduhm	X					VA
vmrghb	X					VX
vmrghh	X					VX
vmrghw	X					VX
vmrglb	X					VX
vmrglh	X					VX
vmrglw	X					VX
vmsummbm	X					VA
vmsumshm	X					VA
vmsumshs	X					VA
vmsumubm	X					VA
vmsumuhm	X					VA
vmsumuhs	X					VA
vmulesb	X					VX
vmulesh	X					VX
vmuleub	X					VX
vmuleuh	X					VX
vmulosb	X					VX
vmulosh	X					VX
vmuloub	X					VX
vmulouh	X					VX
vnmsubfp	X					VA

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Vector/SIMD Multimedia Extension Technology

Table A-7. Vector Instruction Set Legend (Page 4 of 5)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
vnor	X					VX
vor	X					VX
vperm	X					VA
vpkpx	X					VX
vpkshss	X					VX
vpkshus	X					VX
vpkswss	X					VX
vpkuhum	X					VX
vpkuhus	X					VX
vpkwus	X					VX
vakuwum	X					VX
vakuwus	X					VX
vrefp	X					VX
vrfim	X					VX
vrfin	X					VX
vrfip	X					VX
vrfiz	X					VX
vrlb	X					VX
vrlh	X					VX
vrlw	X					VX
vrsqrtefp	X					VX
vsel	X					VA
vsl	X					VX
vslb	X					VX
vsldoi	X					VA
vslh	X					VX
vsllo	X					VX
vsllw	X					VX
vspltb	X					VX
vsplth	X					VX
vspltisb	X					VX
vspltish	X					VX
vspltisw	X					VX
vspltw	X					VX
vsr	X					VX

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.



Table A-7. Vector Instruction Set Legend (Page 5 of 5)

Name	UISA	VEA	OEA	Supervisor Level	Optional	Form
vsrab	X					VX
vsrah	X					VX
vsraw	X					VX
vsrb	X					VX
vsrh	X					VX
vsro	X					VX
vsrw	X					VX
vsubcuw	X					VX
vsubfp	X					VX
vsubbsbs	X					VX
vsubshs	X					VX
vsubsws	X					VX
vsububm	X					VX
vsubuhm	X					VX
vsububs	X					VX
vsubuhs	X					VX
vsubuw	X					VX
vsubuws	X					VX
vsumsws	X					VX
vsum2sws	X					VX
vsum4sbs	X					VX
vsum4shs	X					VX
vsum4ubs	X					VX
vupkhpx	X					VX
vupkhsb	X					VX
vupklsh	X					VX
vupkhpx	X					VX
vupklsb	X					VX
vupklsh	X					VX
vxor	X					VX

Note:

1. The Cell Broadband Engine processor treats this instruction as a no-op.

Glossary

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: 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 <i>implementations</i> .
asynchronous exception	<i>Exceptions</i> that are caused by events external to the processor's execution. In this document, the term 'asynchronous exception' is used interchangeably with the word <i>interrupt</i> .
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 Iwarx/stwcx . (Idarx/stdcx . in 64-bit implementations) instruction pair.

B

biased exponent	An <i>exponent</i> 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 <i>most-significant byte</i> . In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most-significant byte.
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 or instructions (subset of main memory).
cache block	A small region of contiguous memory that is copied from memory into a <i>cache</i> . The size of a cache block may vary among processors; the maximum block size is one <i>page</i> . In PowerPC processors, <i>cache coherency</i> is maintained on a cache-block basis. Note that the term ‘cache block’ is often used interchangeably with ‘cache line’.
changed bit	One of two <i>page history bits</i> found in each <i>page table entry</i> (PTE). The processor sets the changed bit if any store is performed into the <i>page</i> . See also Page access history bits and Referenced bit.
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 <i>cache</i> is bypassed and the load or store is performed to or from main memory.
cast-outs	<i>Cache blocks</i> that must be written to memory when a cache miss causes a cache block to be replaced.
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 <i>exception</i> , that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are <i>fetched</i> 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 <i>cache block</i> is copied back to memory.

D

denormalized number	A nonzero floating-point number whose <i>exponent</i> has a reserved value, typically 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.

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 <i>physical memory</i> 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 canceling 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 <i>primary opcode</i> , identifying the type of instruction. See <i>also</i> 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 <i>also</i> Biased exponent.

F

feed-forwarding	A feature that reduces the number of clock cycles that an execution unit must wait to use a register. When the source register of the current instruction is the same as the destination register of the previous instruction, the result of the previous instruction is routed to the current instruction at the same time that it is written to the register file. With feed-forwarding, the destination bus is gated to the waiting execution unit over the appropriate source bus, saving the cycles which would be used for the write and read.
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 <i>significand</i> that lies to the right of its implied binary point.

Vector/SIMD Multimedia Extension Technology

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 <i>page table</i> 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 <i>optional</i> 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 <i>synchronous exception</i> 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	The act of changing the machine state in response to an 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. The key bits determine whether supervisor or user programs can access a <i>page</i> within that <i>segment</i> .
kill	An operation that causes a <i>cache block</i> 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.
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)	<i>Cache coherency</i> 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.

Vector/SIMD Multimedia Extension Technology

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 <i>effective (logical) address</i> 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.
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.
multiprocessing	The capability of software, especially operating systems, to support execution on more than one processor at the same time.

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 KB area of memory, aligned on a 4 KB boundary or a large page size which is implementation dependent.
page access history bits	The <i>changed</i> and <i>referenced</i> 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 resides within a <i>page</i> not currently resident in <i>physical memory</i> . On PowerPC processors, a page fault exception condition occurs when a matching, valid <i>page table entry</i> (PTE[V] = 1) cannot be located.
page table	A table in memory that consists of <i>page table entries</i> , 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)	A 16-byte data structure containing information used to translate a virtual page address to a physical page address. A page is either 4 KB or an implementation-specific sized large page.
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 that instructions that preceded the faulting instruction can complete, and subsequent instructions can be flushed and redispached 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. See Secondary opcode.

Vector/SIMD Multimedia Extension Technology

protection boundary	A boundary between <i>protection domains</i> .
protection domain	A protection domain is a segment, a virtual page, 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

quadword	A group of 16 contiguous locations starting at an address divisible by 16.
quiet NaN	A type of <i>NaN</i> 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 <i>Signaling NaN</i> .

R

rA	The rA instruction field is used to specify a GPR to be used as a source or destination.
rB	The rB instruction field is used to specify a GPR to be used as a source.
rD	The rD instruction field is used to specify a GPR to be used as a destination.
real address mode	An MMU mode when no address translation is performed and the <i>effective address</i> 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 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 <i>page history bits</i> found in each <i>page table entry</i> (PTE). The processor sets the <i>referenced bit</i> whenever the page is accessed for a read or write. See also <i>Page access history bits</i> .
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 <i>cache block</i> of memory space when it executes an Iwarx or Idarx 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 <i>implementation-dependent</i> . 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 <i>architecture</i> 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 <i>implementations</i> specific for a wide range of purposes, and in particular implementations of significantly greater performance or functionality or both 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 <i>virtual memory</i> that is the most basic memory space defined by the PowerPC Architecture. Each segment is configured through a unique <i>segment descriptor</i> .
segment descriptors	Information used to generate the interim <i>virtual address</i> . The segment descriptors reside in 16 on-chip segment registers for 32-bit implementations. For 64-bit implementations, the segment descriptors reside as <i>segment table entries</i> in a hashed segment table in memory.
segment table	A 4 KB (1-page) data structure that defines the mapping between effective segments and virtual segments for a process. Segment tables are implemented on 64-bit processors only.
segment table entry (STE)	Data structures containing information used to translate <i>effective address</i> to physical address. STEs are implemented on 64-bit processors only.
set (<i>v</i>)	To write a nonzero value to a bit or bit field; the opposite of <i>clear</i> . The term 'set' may also be used to generally describe the updating of a bit or bit field.
set (<i>n</i>)	A subdivision of a <i>cache</i> . 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 <i>Set-associative</i> .
set-associative	Aspect of cache organization in which the cache space is divided into sections, called <i>sets</i> . 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 <i>NaN</i> that generates an invalid operation program exception when it is specified as arithmetic operands. See <i>Quiet NaN</i> .

Vector/SIMD Multimedia Extension Technology

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.
SLB (segment lookaside buffer)	An optional cache that holds recently-used <i>segment table entries</i> .
splat	A splat instruction takes one element and replicates (splats) that value into a vector register. The purpose of the splat instruction is to ensure that all elements have the same value so that 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 hardware about the direction a branch is likely to take.
sticky bit	A bit that when <i>set</i> 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 <i>in order</i> . See Context synchronization and Execution synchronization.
synchronous exception	An <i>exception</i> that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous exceptions, <i>precise</i> and <i>imprecise</i> .
system memory	The physical memory available to a processor.

T

TLB (translation lookaside buffer)	A cache that holds recently-used <i>page table entries</i> .
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 <i>denormalized</i> 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

UIISA (user instruction set architecture)	The level of the architecture to which user-level software should conform. The UIISA 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 <i>exponent</i> 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 vA instruction field is used to specify a vector register to be used as a source or destination.
vB	The vB instruction field is used to specify a vector register to be used as a source.
vC	The vC instruction field is used to specify a vector register to be used as a source.
vD	The vD instruction field is used to specify a vector register to be used as a destination.
vS	The vS instruction field is used to specify a vector register to be used as a source.
VEA (virtual environment architecture)	The level of the <i>architecture</i> 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. <i>Implementations</i> that conform to the PowerPC VEA also adhere to the UIISA, 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 vector instructions.

Vector/SIMD Multimedia Extension Technology

virtual address	An intermediate address used in the translation of an <i>effective address</i> 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 <i>physical memory</i> .

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 <i>cast out</i> 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.



Revision Log

Revision Date	Contents of Modification
Version 2.07 October 26, 2006	<p>Version 2.07</p> <ul style="list-style-type: none">• Changed “vector multimedia register (VMR)” to “vector register (VR)” and changed “vector multimedia register file (VMRF)” to “vector register file (VRF)” throughout.• Changed “command” to “instruction” throughout.• Deleted redundant notes indicating that the Cell Broadband Engine processor does not support little-endian mode.• Changed the notes indicating that the data stream instructions are not supported on the Cell Broadband Engine processor to indicate that they are treated as no-ops.• Removed all discussion of little-endian mode. (No current implementations of the vector/SIMD multimedia extension technology support little-endian mode.)<ul style="list-style-type: none">– Deleted <i>Section 3.1.2.1 Big-Endian Byte Ordering</i>, <i>Section 3.1.2.2 Little-Endian Byte Ordering</i>, <i>Section 3.1.3 Quadword Byte Ordering Example</i>, <i>Section 3.1.4 Aligned Scalars in Little-Endian Mode</i>., and <i>Section 3.1.7 Mixed-Endian Systems</i>.– Replaced <i>Section 3.1.6 Quadword Data Alignment</i> with revised information (see <i>Section 3.1.4 Quadword Data Alignment</i> on page 44).– Changed the RTL, or the description, or both of the following instructions: lvebx, lvehx, lvewx, stvebx, stvehx, stvewx, stvx, lvebx, lvehx, lvewx, lvlx, lvlxl, lvrx, lvrxl, lvx, lvxl, stvebx, stvehx, stvewx, stvxl, stvrx, strvxl, stvx.• Changed the definition of DSLAM from “digital subscriber loop access multiplexer” to “digital subscriber line access multiplexer” (see <i>Section 1 Overview</i> on page 23).• Removed a statement indicating that vector load and store instructions that attempt to access a direct-store segment will cause a DS1 exception (see <i>Section 1.2.6 Vector Exception Model</i> on page 32).• Corrected the description of the VRSAVE Register and the VRn field (see <i>Section 2.2.3 VRSAVE Register (VRSAVE)</i> on page 38).• Clarified what happens to SRR1 when a VPU unavailable exception occurs (see <i>Section 2.4.2.2 Machine Status Save/Restore Register 1 (SRR1)</i> on page 42).• Simplified the discussion of alignment (see <i>Section 3.1.1 Aligned and Misaligned Accesses</i> on page 43).• Revised the description of vector load and store instructions (see <i>Section 4.2.3 Vector Load and Store Instructions</i> on page 75).• Added a note to <i>Table 4-15 Vector Load Instructions</i> on page 78.• Added a note to <i>Table 4-16 Vector Load Instructions Supporting Alignment</i> on page 80.• Added the Vector Shift Left instruction to <i>Table 4-24 Vector Shift Instructions</i> on page 87.• Removed a note indicating that data stream instructions are not supported on the Cell Broadband Engine (see <i>Section 5.1 PowerPC Shared Memory</i> on page 95).• Moved a note indicating that the Cell Broadband Engine implements an extended definition of dcbt and corrected a reference (see <i>Section 5.1 PowerPC Shared Memory</i> on page 95 and <i>Section 5.2.1 Software-Directed Prefetch</i> on page 96).• Removed references to 32-bit mode (see <i>Section 5.2.1.1 Data Stream Touch (dst)</i> on page 96).• Reworded the description of transient streams (see <i>Section 5.2.1.2 Transient Streams (dstt)</i> on page 98).• Clarified the reasons why dst instructions should be issued periodically (see <i>Section 5.2.1.8 Stream Usage Notes</i> on page 101).• Expanded the list of instructions treated as no-ops by certain processors and clarified the description of the functionality of these instructions (see <i>Section 5.2.1.9 Stream Implementation Assumptions</i> on page 103).• Provided a reference to <i>PowerPC Operating Environment Architecture, Book III</i> (see <i>Section 5.2.3 Partially Executed Vector Instructions</i> on page 103).• Revised the description of the DS1 exception (see <i>Section 5.3 DS1 Exception—Data Address Breakpoint</i> on page 104).• Revised the description of the UIMM field (see <i>Table 6-1 Instruction Syntax Conventions</i> on page 107).• Corrected the meaning of SImaximum(X,Y) and SIminimum(X,Y) (see <i>Table 6-2 Notation and Conventions</i> on page 108).• Indicated that Block Stride is a signed integer (see <i>Data Stream Touch (0x7C00 02AC)</i> on page 115 and <i>Data Stream Touch for Store (0x7C00 02EC)</i> on page 117).• Corrected operands in the vcmpbfpx instruction (see <i>Vector Compare Bounds Floating Point (0x1000 03C6)</i> on page 164).



Vector/SIMD Multimedia Extension Technology

Revision Date	Contents of Modification
Version 2.07 October 26, 2006 (continued)	<ul style="list-style-type: none"> • Indicated that vB should be negated before comparison (see <i>Figure 6-29 vcmpbf—Compare Bounds of Four Floating-Point Elements</i> on page 165). • Deleted notes that were not applicable to the data type (see <i>Vector Compare Greater-Than Signed Byte (0x1000 0306)</i> on page 172, <i>Vector Compare Greater-Than Signed Halfword (0x1000 0346)</i> on page 173, <i>Vector Compare Greater-Than Signed Word (0x1000 0386)</i> on page 174, <i>Vector Compare Greater-Than Unsigned Byte (0x1000 0206)</i> on page 175, <i>Vector Compare Greater-Than Unsigned Halfword (0x1000 0246)</i> on page 176, and <i>Vector Compare Greater-Than Unsigned Word (0x1000 0286)</i> on page 177). • Corrected the description of the vlogefp instruction (see <i>Vector Log2 Estimate Floating Point (0x1000 01CA)</i> on page 182). • Corrected the number of elements in registers vA and vB and the length of vD in the introduction to several figures (see <i>Figure 6-57 vminsb—Minimum of Sixteen Signed Integer Elements</i> on page 195, <i>Figure 6-58 vminsh—Minimum of Eight Signed Integer Elements</i> on page 196, <i>Figure 6-60 vminub—Minimum of Sixteen Unsigned Integer Elements</i> on page 198, <i>Figure 6-69 vmrglw—Merge Two Low-Order Elements</i> on page 207). • Deleted a sentence indicating “if the intermediate result is less than ‘0’, it saturates to ‘0’” (see <i>Vector Multiply Sum Unsigned Halfword Saturate (0x1000 0027)</i> on page 213). • Added notes indicating that the result of the floating-point operation is independent of VSCR[NJ] (see <i>Vector Round to Floating-Point Integer toward Minus Infinity (0x1000 02CA)</i> on page 237 and <i>Vector Round to Floating-Point Integer toward Plus Infinity (0x1000 028A)</i> on page 239). • Corrected the size of the UIMM field in the following instructions: vspltb, vspith, vspltw (see <i>Vector Splat Byte (0x1000 020C)</i> on page 252, <i>Vector Splat Halfword (0x1000 024C)</i> on page 253, and <i>Vector Splat Word (0x1000 028C)</i> on page 257). • Changed “bit[n-1]” to “bit[7]” in the description of the vsrab instruction (see <i>Vector Shift Right Algebraic Byte (0x1000 0304)</i> on page 260). • Changed the unary minus operator to a logical NOT in the pseudocode for the following instructions: vsubcuw, vsubbsb, vsubshs, vsubsws, vsububm, vsububs, vsubuhm, vsubuhs, vsubuwm, vsubuws (see <i>Vector Subtract Carryout Unsigned Word (0x1000 0580)</i> on page 267, <i>Vector Subtract Signed Byte Saturate (0x1000 0700)</i> on page 269, <i>Vector Subtract Signed Halfword Saturate (0x1000 0740)</i> on page 270, <i>Vector Subtract Signed Word Saturate (0x1000 0780)</i> on page 271, <i>Vector Subtract Unsigned Byte Modulo (0x1000 0400)</i> on page 272, <i>Vector Subtract Unsigned Byte Saturate (0x1000 0600)</i> on page 273, <i>Vector Subtract Signed Halfword Modulo (0x1000 0440)</i> on page 274, <i>Vector Subtract Signed Halfword Saturate (0x1000 0640)</i> on page 275, <i>Vector Subtract Unsigned Word Modulo (0x1000 0480)</i> on page 276, <i>Vector Subtract Unsigned Word Saturate (0x1000 0680)</i> on page 277). • Corrected two entries in the Glossary (see <i>Glossary</i> on page 315). • Made other editorial changes for consistency and clarity.
Version 2.06 September 30, 2005	<p>Version 2.06</p> <ul style="list-style-type: none"> • Added Cell Broadband Engine processor instructions: lvlx, lvxl, lvrx, lvrl, stvlx, stvxl, stvrx, stvrl. • Added note that instructions: dss, dssall, dst, dstt, dstst, dststt are not supported on the Cell Broadband Engine processor. • Changed quad word to quadword, half word to halfword, double word to doubleword, and so forth. • Removed 32-bit implementation information. • Various editing corrections. • Updated trademarks. • Corrected the pseudocode for the following instructions: vmsumshm, vmsumshs, vmsumuhm, vmsumuhs, vsl, vsr. • Changed the description of lvxl and stvxl. • Changed “vector register (VR)” to “vector multimedia register (VMR)”; changed “VXU register file (VRF)” to “vector multimedia register file (VMRF).” • In <i>Table 4-27 User-Level Cache Instructions</i> on page 92, revised the descriptions of the transient and nontransient Data Stream Touch and Data Stream Touch for Store instructions.



Revision Date	Contents of Modification
July 10, 2003	<p>Updated format and cross references to match IBM template.</p> <p>Chapter 2 corrections as follows:</p> <ul style="list-style-type: none">• Updated description of VRSAVE register (<i>Section 2.2.3 VRSAVE Register (VRSAVE)</i>).• Fixed <i>Figure 2-1 Programming Model—All Registers</i>. <p>Chapter 4 corrections as follows:</p> <ul style="list-style-type: none">• Fixed <i>Table 4-9 Vector Floating-Point Rounding and Conversion Instructions</i> as follows:• Mnemonic for Vector Round to Floating-Point Integer Nearest is vrfin (not fvrfin)• Mnemonic for Vector Round to Floating-Point Integer toward Zero is vrfiz (not fvrifiz)• Mnemonic for Vector Round to Floating-Point Integer toward Positive Infinity is vrfip (not fvrifip)• Mnemonic for Vector Round to Floating-Point Integer toward Minus Infinity is vrfim (not fvrifim) <p>Chapter 6 corrections/enhancements as follows:</p> <ul style="list-style-type: none">• Add hex codes for instructions (in chapter 6).• Fixed mfvscr bit description table (bit 31 should not be zero).• Fixed mtvscr bit description table (bit 31 should not be zero).• Changed vexptefp bit description from 458 to 394. <p>Updated the Appendix as follows:</p> <ul style="list-style-type: none">• Appendix A-1, <i>Table A-1</i>, changed mfvscr bit 31 (should not be zero).• Appendix A-1, <i>Table A-1</i>, changed mtvscr bit 31 (should not be zero) and vD should be vB.• Appendix A-1, <i>Table A-1</i>, corrected vcfux bit 31 (merged with bits 21 to 31).• Appendix A-2, <i>Table A-2</i>, changed mfvscr encoding for bits 21 to 31 to: 110 0000 0100.• Appendix A-2, <i>Table A-2</i>, changed mtscr encoding for bits 21 to 31 to: 110 0100 0100.• Appendix A-3, <i>Table A-4</i>, changed mfvscr bit 31 (should not be zero).• Appendix A-3, <i>Table A-4</i>, changed mtvscr bit 31 (should not be zero).