AltiVec Technology Programming Environments Manual for Power ISA Processors

ALTIVECPOWERISAPEM
Rev 0
06/2014

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: ALTIVECPOWERISAPEM
Rev 0, 06/2014

Contents

Paragraph Number	Title	Page Number

About This Book

Audience xv
Organization xvi
Suggested Reading xvi
  General Information xvi
  Related Documentation xvii
Conventions xvii
Acronyms and Abbreviations xviii
Terminology Conventions xx

Chapter 1 Overview

1.1 History and Classification 1-1
1.2 Overview 1-1
1.3 AltiVec Technology Overview 1-3
1.3.1 Features Not Defined by AltiVec Architecture 1-4
1.4 AltiVec Programming Model 1-5
1.4.1 AltiVec Registers and Programming Model 1-5
1.4.2 Operand Conventions 1-6
1.4.2.1 Byte Ordering 1-6
1.4.2.2 Floating-Point Conventions 1-7
1.4.3 AltiVec Element Operations 1-7
1.4.4 AltiVec Instruction Set 1-9
1.4.5 AltiVec Interrupt Model 1-10
1.5 Changes from the Original AltiVec Definition 1-10

Chapter 2 AltiVec Register Set

2.1 AltiVec Register Set Overview 2-1
2.2 Registers Defined by AltiVec 2-2
2.2.1 AltiVec Vector Register File (VRF) 2-2
2.2.2 Vector Status and Control Register (VSCR) 2-3
2.2.3 Vector Save/Restore Register (VRSAVE) 2-4
2.2.4 AltiVec IVOR Registers 2-5
2.3 Non-Vector Category Registers Affected 2-6

Contents

Paragraph Number	Title	Page Number
2.4	AltiVec Specific Fields in Supervisor Registers	2-6

Chapter 3 Operand Conventions

3.1	Data Organization in Memory	3-1
3.1.1	Aligned and Misaligned Accesses	3-1
3.1.2	AltiVec Byte Ordering	3-2
3.1.2.1	Big-Endian Byte Ordering	3-2
3.1.3	Quad Word Byte Ordering Example	3-2
3.1.4	Vector Register and Memory Access Alignment	3-3
3.1.5	Quad-Word Data Alignment	3-3
3.1.5.1	Accessing a Misaligned Quad Word in Big-Endian Mode	3-4
3.1.5.2	Scalar Loads and Stores	3-5
3.1.5.3	Misaligned Scalar Loads and Stores	3-6
3.1.6	Mixed-Endian Systems	3-6
3.2	AltiVec Floating-Point Instructions	3-6
3.2.1	Floating-Point Modes	3-7
3.2.1.1	Java Mode	3-7
3.2.1.2	Non-Java Mode	3-7
3.2.2	Floating-Point Infinities	3-8
3.2.3	Floating-Point Rounding	3-8
3.2.4	Floating-Point Exceptions	3-8
3.2.4.1	NaN Operand Exception	3-8
3.2.4.2	Invalid Operation Exception	3-9
3.2.4.3	Zero Divide Exception	3-9
3.2.4.4	Log of Zero Exception	3-10
3.2.4.5	Overflow Exception	3-10
3.2.4.6	Underflow Exception	3-10
3.2.5	Floating-Point NaNs	3-11
3.2.5.1	NaN Precedence	3-11
3.2.5.2	SNaN Arithmetic	3-11
3.2.5.3	QNaN Arithmetic	3-11
3.2.5.4	NaN Conversion to Integer	3-11
3.2.5.5	NaN Production	3-11

Chapter 4 Addressing Modes and Instruction Set Summary

4.1	Conventions	4-1
4.1.1	Computation Modes	4-2

Contents

Paragraph Number	Title	Page Number
4.1.2	Classes of Instructions	4-2
4.1.3	Memory Addressing	4-2
4.2	AltiVec Instructions	4-2
4.2.1	Vector Integer Instructions	4-2
4.2.1.1	Saturation Detection	4-2
4.2.1.2	Vector Integer Arithmetic Instructions	4-4
4.2.1.3	Vector Integer Compare Instructions	4-11
4.2.1.4	Vector Integer Logical Instructions	4-12
4.2.1.5	Vector Integer Rotate and Shift Instructions	4-13
4.2.2	Vector Floating-Point Instructions	4-14
4.2.2.1	Floating-Point Division and Square-Root	4-14
4.2.2.1.1	Floating-Point Division	4-14
4.2.2.1.2	Floating-Point Square-Root	4-15
4.2.2.2	Floating-Point Arithmetic Instructions	4-15
4.2.2.3	Floating-Point Multiply-Add Instructions	4-16
4.2.2.4	Floating-Point Rounding and Conversion Instructions	4-17
4.2.2.5	Floating-Point Compare Instructions	4-18
4.2.2.6	Floating-Point Estimate Instructions	4-20
4.2.3	Load and Store Instructions	4-21
4.2.3.1	Alignment	4-22
4.2.3.2	Load and Store Address Generation	4-22
4.2.3.3	Vector Load Instructions	4-23
4.2.3.4	Vector Load Instructions for Misaligned Vector Handling	4-25
4.2.3.5	Vector Store Instructions	4-27
4.2.3.6	Vector Store Instructions for Misaligned Vector Handling	4-27
4.2.3.7	Using Left, Right, and Swap Load and Store Instructions for Misaligned Vector Access	4-29
4.2.4	Vector Permutation and Formatting Instructions	4-31
4.2.4.1	Vector Pack Instructions	4-32
4.2.4.2	Vector Unpack Instructions	4-33
4.2.4.3	Vector Merge Instructions	4-34
4.2.4.4	Vector Splat Instructions	4-35
4.2.4.5	Vector Permute Instruction	4-36
4.2.4.6	Vector Select Instruction	4-36
4.2.4.7	Vector Shift Instructions	4-37
4.2.4.7.1	Immediate inter-element Shifts/Rotates	4-37
4.2.4.7.2	Computed inter-element Shifts/Rotates	4-38
4.2.4.7.3	Variable inter-element Shifts	4-38
4.2.4.8	AltiVec Status and Control Register Instructions	4-39
4.2.4.9	GPR to AltiVec Move Instructions	4-39

Contents

Paragraph Number — Title — Page Number

Chapter 5
Cache, Interrupts, and Memory Management

5.1 Prioritizing Cache Block Replacement 5-1
5.2 Partially Executed AltiVec Instructions 5-1
5.3 AltiVec Unavailable Interrupt 5-1
5.4 AltiVec Assist Interrupt 5-2

Chapter 6
AltiVec Instructions

6.1 Instruction Formats 6-1
6.1.1 Instruction Fields 6-1
6.1.2 Notation and Conventions 6-2
6.2 AltiVec Instruction Set 6-7

Appendix A
AltiVec Instruction Set Listings

A.1 Instructions Sorted by Mnemonic in Decimal Format 7-1
A.2 Instructions Sorted by Mnemonic in Binary Format 7-7
A.3 Instructions Sorted by Opcode in Decimal Format 7-13
A.4 Instructions Sorted by Opcode in Binary Format 7-19
A.5 Instructions Sorted by Form 7-25

Appendix B
Revision History

Glossary

Figures

Figure Number	Title	Page Number
1-1	Overview of an example implementation with AltiVec Technology	1-4
1-2	AltiVec Top-Level Diagram	1-6
1-3	Big-Endian Byte Ordering for a Vector Register	1-6
1-4	Bit Ordering	1-7
1-5	Intra-Element Example, **vaddsws**	1-7
1-6	inter-element Example, **vperm**	1-8
2-1	AltiVec Register Set	2-1
2-2	Vector Registers (VRs)	2-2
2-3	Vector Status and Control Register (VSCR)	2-3
2-4	32-Bit VSCR Moved to a 128-Bit Vector Register	2-3
2-5	Vector Save/Restore Register (VRSAVE)	2-5
3-1	Big-Endian Mapping of a Quad Word	3-3
3-2	Misaligned Vector in Big-Endian Mode	3-4
3-3	Big-Endian Quad Word Alignment	3-4
4-1	Register Indirect with Index Addressing for Loads/Stores	4-23
6-1	Effects of Example Load/Store Instructions	6-12
6-2	Example **lvexbx** Instruction	6-17
6-3	Example **lvexhx** Instruction	6-18
6-4	Example **lvexwx** Instruction	6-19
6-5	Example **lvtlx** Instruction	6-20
6-6	Example **lvtlxl** Instruction	6-21
6-7	Example **lvtrx** Instruction	6-22
6-8	Example **lvtrxl** Instruction	6-23
6-9	Load Vector for Shift Left	6-25
6-10	Instruction **vperm** Used in Aligning Data	6-26
6-11	Example **lvsm** Instruction	6-28
6-12	Example **lvswx** Instruction	6-30
6-13	Example **lvswxl** Instruction	6-32
6-14	Example **stvexbx** Instruction	6-44
6-15	Example **stvexhx** Instruction	6-45
6-16	Example **stvflx** Instruction	6-47
6-17	Example **stvflxl** Instruction	6-48
6-18	Example **stvfrx** Instruction	6-49
6-19	Example **stvfrxl** Instruction	6-51
6-20	Example **stvswx** Instruction	6-52
6-21	Example **stvswxl** Instruction	6-54
6-22	**vabsdub**—Absolute Differences Sixteen Integer Elements (8-Bit)	6-57
6-23	**vabsduh**—Absolute Differences Eight Integer Elements (16-Bit)	6-58
6-24	**vabsduw**—Absolute Differences Four Integer Elements (32-Bit)	6-59
6-25	**vaddcuw**—Determine Carries of Four Unsigned Integer Adds (32-Bit)	6-60
6-26	**vaddfp**—Add Four Floating-Point Elements (32-Bit)	6-61

Figures

Figure Number	Title	Page Number
6-27	**vaddsbs**—Add Saturating Sixteen Signed Integer Elements (8-Bit)	6-62
6-28	**vaddshs**—Add Saturating Eight Signed Integer Elements (16-Bit)	6-63
6-29	**vaddsws**—Add Saturating Four Signed Integer Elements (32-Bit)	6-64
6-30	**vaddubm**—Add Sixteen Integer Elements (8-Bit)	6-65
6-31	**vaddubs**—Add Saturating Sixteen Unsigned Integer Elements (8-Bit)	6-66
6-32	**vadduhm**—Add Eight Integer Elements (16-Bit)	6-67
6-33	**vadduhs**—Add Saturating Eight Unsigned Integer Elements (16-Bit)	6-68
6-34	**vadduwm**—Add Four Integer Elements (32-Bit)	6-69
6-35	**vadduws**—Add Saturating Four Unsigned Integer Elements (32-Bit)	6-70
6-36	**vand**—Logical Bitwise AND	6-71
6-37	**vand**—Logical Bitwise AND with Complement	6-72
6-38	**vavgsb**—Average Sixteen Signed Integer Elements (8-Bit)	6-73
6-39	**vavgsh**—Average Eight Signed Integer Elements (16-Bit)	6-74
6-40	**vavgsw**—Average Four Signed Integer Elements (32-Bit)	6-75
6-41	**vavgub**—Average Sixteen Unsigned Integer Elements (8-Bit)	6-76
6-42	**vavguh**—Average Eight Signed Integer Elements (16-Bit)	6-77
6-43	**vavguw**—Average Four Unsigned Integer Elements (32-Bit)	6-78
6-44	**vcfsx**—Convert Four Signed Integer Elements to Four Floating-Point Elements (32-Bit)	6-79
6-45	**vcfux**—Convert Four Unsigned Integer Elements to Four Floating-Point Elements (32-Bit)	6-80
6-46	**vcmpbfp**—Compare Bounds of Four Floating-Point Elements (32-Bit)	6-82
6-47	**vcmpeqfp**—Compare Equal of Four Floating-Point Elements (32-Bit)	6-83
6-48	**vcmpequb**—Compare Equal of Sixteen Integer Elements (8-Bit)	6-84
6-49	**vcmpequh**—Compare Equal of Eight Integer Elements (16-Bit)	6-85
6-50	**vcmpequw**—Compare Equal of Four Integer Elements (32-Bit)	6-86
6-51	**vcmpgefp**—Compare Greater-Than-or-Equal of Four Floating-Point Elements (32-Bit)	6-87
6-52	**vcmpgtfp**—Compare Greater-Than of Four Floating-Point Elements (32-Bit)	6-88
6-53	**vcmpgtsb**—Compare Greater-Than of Sixteen Signed Integer Elements (8-Bit)	6-89
6-54	**vcmpgtsh**—Compare Greater-Than of Eight Signed Integer Elements (16-Bit)	6-90
6-55	**vcmpgtsw**—Compare Greater-Than of Four Signed Integer Elements (32-Bit)	6-91
6-56	**vcmpgtub**—Compare Greater-Than of Sixteen Unsigned Integer Elements (8-Bit)	6-92
6-57	**vcmpgtuh**—Compare Greater-Than of Eight Unsigned Integer Elements (16-Bit)	6-93
6-58	**vcmpgtuw**—Compare Greater-Than of Four Unsigned Integer Elements (32-Bit)	6-94
6-59	**vctsxs**—Convert Four Floating-Point Elements to Four Signed Integer Elements (32-Bit)	6-95
6-60	**vctuxs**—Convert Four Floating-Point Elements to Four Unsigned Integer Elements (32-Bit)	6-96
6-61	**vexptefp**—Two Raised to the Exponent Estimate Floating-Point for Four Floating-Point Elements (32-Bit)	6-98

Figures

Figure Number	Title	Page Number
6-62	**vlogefp**—Log$_2$ Estimate Floating-Point for Four Floating-Point Elements (32-Bit)	6-100
6-63	**vmaddfp**—Multiply-Add Four Floating-Point Elements (32-Bit)	6-101
6-64	**vmaxfp**—Maximum of Four Floating-Point Elements (32-Bit)	6-102
6-65	**vmaxsb**—Maximum of Sixteen Signed Integer Elements (8-Bit)	6-103
6-66	**vmaxsh**—Maximum of Eight Signed Integer Elements (16-Bit)	6-104
6-67	**vmaxsw**—Maximum of Four Signed Integer Elements (32-Bit)	6-105
6-68	**vmaxub**—Maximum of Sixteen Unsigned Integer Elements (8-Bit)	6-106
6-69	**vmaxuh**—Maximum of Eight Unsigned Integer Elements (16-Bit)	6-107
6-70	**vmaxuw**—Maximum of Four Unsigned Integer Elements (32-Bit)	6-108
6-71	**vmhaddshs**—Multiply-High and Add Eight Signed Integer Elements (16-Bit)	6-109
6-72	**vmhraddshs**—Multiply-High Round and Add Eight Signed Integer Elements (16-Bit)	6-110
6-73	**vminfp**—Minimum of Four Floating-Point Elements (32-Bit)	6-111
6-74	**vminsb**—Minimum of Sixteen Signed Integer Elements (8-Bit)	6-112
6-75	**vminsh**—Minimum of Eight Signed Integer Elements (16-Bit)	6-113
6-76	**vminsw**—Minimum of Four Signed Integer Elements (32-Bit)	6-114
6-77	**vminub**—Minimum of Sixteen Unsigned Integer Elements (8-Bit)	6-115
6-78	**vminuh**—Minimum of Eight Unsigned Integer Elements (16-Bit)	6-116
6-79	**vminuw**—Minimum of Four Unsigned Integer Elements (32-Bit)	6-117
6-80	**vmladduhm**—Multiply-Add of Eight Integer Elements (16-Bit)	6-118
6-81	**vmrghb**—Merge Eight High-Order Elements (8-Bit)	6-119
6-82	**vmrghh**—Merge Four High-Order Elements (16-Bit)	6-120
6-83	**vmrghw**—Merge Two High-Order Elements (32-Bit)	6-121
6-84	**vmrglb**—Merge Eight Low-Order Elements (8-Bit)	6-122
6-85	**vmrglh**—Merge Four Low-Order Elements (16-Bit)	6-123
6-86	**vmrglw**—Merge Four Low-Order Elements (32-Bit)	6-124
6-87	**vmsummbm**—Multiply-Sum of Integer Elements (8- to 32-Bit)	6-125
6-88	**vmsumshm**—Multiply-Sum of Signed Integer Elements (16- to 32-Bit)	6-126
6-89	**vmsumshs**—Multiply-Sum of Signed Integer Elements (16- to 32-Bit)	6-127
6-90	**vmsumubm**—Multiply-Sum of Unsigned Integer Elements (8- to 32-Bit)	6-128
6-91	**vmsumuhm**—Multiply-Sum of Unsigned Integer Elements (16- to 32-Bit)	6-129
6-92	**vmsumuhs**—Multiply-Sum of Unsigned Integer Elements (16- to 32-Bit)	6-130
6-93	**vmulesb**—Even Multiply of Eight Signed Integer Elements (8-Bit)	6-131
6-94	**vmulesh**—Even Multiply of Four Signed Integer Elements (16-Bit)	6-132
6-95	**vmuleub**—Even Multiply of Eight Unsigned Integer Elements (8-Bit)	6-133
6-96	**vmuleuh**—Even Multiply of Four Unsigned Integer Elements (16-Bit)	6-134
6-97	**vmulosb**—Odd Multiply of Eight Signed Integer Elements (8-Bit)	6-135
6-98	**vmuleuh**—Odd Multiply of Four Unsigned Integer Elements (16-Bit)	6-136
6-99	**vmuloub**—Odd Multiply of Eight Unsigned Integer Elements (8-Bit)	6-137
6-100	**vmulouh**—Odd Multiply of Four Unsigned Integer Elements (16-Bit)	6-138

Figures

Figure Number	Title	Page Number
6-101	**vnmsubfp**—Negative Multiply-Subtract of Four Floating-Point Elements (32-Bit)	6-139
6-102	**vnor**—Bitwise NOR of 128-Bit Vector	6-140
6-103	**vor**—Bitwise OR of 128-Bit Vector	6-141
6-104	**vperm**—Concatenate Sixteen Integer Elements (8-Bit)	6-142
6-105	**vpkpx**—How a Word is Packed to a Half Word	6-143
6-106	**vpkpx**—Pack Eight Elements (32-Bit) to Eight Elements (16-Bit)	6-144
6-107	**vpkshss**—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Signed Integer Elements (8-Bit)	6-145
6-108	**vpkshus**—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)	6-146
6-109	**vpkswss**—Pack Eight Signed Integer Elements (32-Bit) to Eight Signed Integer Elements (16-Bit)	6-147
6-110	**vpkswus**—Pack Eight Signed Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)	6-148
6-111	**vpkuhum**—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)	6-149
6-112	**vpkuhus**—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)	6-150
6-113	**vpkuwum**—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)	6-151
6-114	**vpkuwus**—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)	6-152
6-115	**vrefp**—Reciprocal Estimate of Four Floating-Point Elements (32-Bit)	6-154
6-116	**vrfim**—Round to Minus Infinity of Four Floating-Point Integer Elements (32-Bit)	6-155
6-117	**vrfin**—Nearest Round to Nearest of Four Floating-Point Integer Elements (32-Bit)	6-156
6-118	**vrfip**—Round to Plus Infinity of Four Floating-Point Integer Elements (32-Bit)	6-157
6-119	**vrfiz**—Round-to-Zero of Four Floating-Point Integer Elements (32-Bit)	6-158
6-120	**vrlb**—Left Rotate of Sixteen Integer Elements (8-Bit)	6-159
6-121	**vrlh**—Left Rotate of Eight Integer Elements (16-Bit)	6-160
6-122	**vrlw**—Left Rotate of Four Integer Elements (32-Bit)	6-161
6-123	**vrsqrtefp**—Reciprocal Square Root Estimate of Four Floating-Point Elements (32-Bit)	6-162
6-124	**vsel**—Bitwise Conditional Select of Vector Contents (128-Bit)	6-163
6-125	**vsl**—Shift Bits Left in Vector (128-Bit)	6-164
6-126	**vslb**—Shift Bits Left in Sixteen Integer Elements (8-Bit)	6-165
6-127	**vsldoi**—Shift Left by Bytes Specified	6-166
6-128	**vslh**—Shift Bits Left in Eight Integer Elements (16-Bit)	6-167
6-129	**vslo**—Left Byte Shift of Vector (128-Bit)	6-168
6-130	**vslw**—Shift Bits Left in Four Integer Elements (32-Bit)	6-169
6-131	**vspltb**—Copy Contents to Sixteen Elements (8-Bit)	6-170
6-132	**vsplth**—Copy Contents to Eight Elements (16-Bit)	6-171

Figures

Figure Number	Title	Page Number
6-133	**vspltisb**—Copy Value into Sixteen Signed Integer Elements (8-Bit)	6-172
6-134	**vspltish**—Copy Value to Eight Signed Integer Elements (16-Bit)	6-173
6-135	**vspltisw**—Copy Value to Four Signed Elements (32-Bit)	6-174
6-136	**vspltw**—Copy Contents to Four Elements (32-Bit)	6-175
6-137	**vsr**—Shift Bits Right for Vectors (128-Bit)	6-177
6-138	**vsrab**—Shift Bits Right in Sixteen Integer Elements (8-Bit)	6-178
6-139	**vsrah**—Shift Bits Right for Eight Integer Elements (16-Bit)	6-179
6-140	**vsraw**—Shift Bits Right in Four Integer Elements (32-Bit)	6-180
6-141	**vsrb**—Shift Bits Right in Sixteen Integer Elements (8-Bit)	6-181
6-142	**vsrh**—Shift Bits Right for Eight Integer Elements (16-Bit)	6-182
6-143	**vsro**—Vector Shift Right Octet	6-183
6-144	**vsrw**—Shift Bits Right in Four Integer Elements (32-Bit)	6-184
6-145	**vsubcuw**—Subtract Carryout of Four Unsigned Integer Elements (32-Bit)	6-185
6-146	**vsubfp**—Subtract Four Floating-Point Elements (32-Bit)	6-186
6-147	**vsubsbs**—Subtract Sixteen Signed Integer Elements (8-Bit)	6-187
6-148	**vsubshs**—Subtract Eight Signed Integer Elements (16-Bit)	6-188
6-149	**vsubsws**—Subtract Four Signed Integer Elements (32-Bit)	6-189
6-150	**vsububm**—Subtract Sixteen Integer Elements (8-Bit)	6-190
6-151	**vsububs**—Subtract Sixteen Unsigned Integer Elements (8-Bit)	6-191
6-152	**vsubuhm**—Subtract Eight Integer Elements (16-Bit)	6-192
6-153	**vsubuhs**—Subtract Eight Unsigned Integer Elements (16-Bit)	6-193
6-154	**vsubuwm**—Subtract Four Integer Elements (32-Bit)	6-194
6-155	**vsubuws**—Subtract Four Signed Integer Elements (32-Bit)	6-195
6-156	**vsumsws**—Sum Four Signed Integer Elements (32-Bit)	6-196
6-157	**vsum2sws**—Two Sums in the Four Signed Integer Elements (32-Bit)	6-197
6-158	**vsum4sbs**—Sum of Four Signed Integer Byte Elements with a Word Element (32-Bit)	6-198
6-159	**vsum4shs**—Sum of Two Signed Integer Half-Word Elements with a Word Element (32-Bit)	6-199
6-160	**vsum4ubs**—Sum of Four Unsigned Integer Byte Elements with an Unsigned Integer Word Element (32-Bit)	6-200
6-161	**vupkhpx**—Unpack High-Order Elements (16-Bit) to Elements (32-Bit)	6-201
6-162	**vupkhsb**—Unpack High-Order Signed Integer Elements (8-Bit) to Signed Integer Elements (16-Bit)	6-202
6-163	**vupkhsh**—Unpack Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)	6-203
6-164	**vupklpx**—Unpack Low-Order Elements (16-Bit) to Elements (32-Bit)	6-204
6-165	**vupklsb**—Unpack Low-Order Elements (8-Bit) to Elements (16-Bit)	6-205
6-166	**vupklsh**—Unpack Low-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)	6-206
6-167	**vxor**—Bitwise XOR (128-Bit)	6-207

Figures

Figure Number	Title	Page Number

Tables

Table Number	Title	Page Number
i	Acronyms and Abbreviated Terms	xviii
ii	Terminology Conventions	xx
iii	Instruction Field Conventions	xx
1-1	Summary of Changes from Original AltiVec Definition	1-10
2-1	VSCR Field Descriptions	2-3
2-2	VRSAVE Bit Settings	2-5
2-3	AltiVec IVOR Registers	2-5
2-4	CR6 Field's Bit Settings for Vector Compare Instructions	2-6
3-1	Memory Operand Alignment	3-1
4-1	Vector Integer Arithmetic Instructions	4-4
4-2	CR6 Field Bit Settings for Vector Integer Compare Instructions	4-11
4-3	Vector Integer Compare Instructions	4-11
4-4	Vector Integer Logical Instructions	4-12
4-5	Vector Integer Rotate Instructions	4-13
4-6	Vector Integer Shift Instructions	4-13
4-7	Floating-Point Arithmetic Instructions	4-15
4-8	Floating-Point Multiply-Add Instructions	4-17
4-9	Floating-Point Rounding and Conversion Instructions	4-17
4-10	Common Mathematical Predicates	4-18
4-11	Other Useful Predicates	4-19
4-12	Floating-Point Compare Instructions	4-20
4-13	Floating-Point Estimate Instructions	4-20
4-14	Effective Address Alignment	4-22
4-15	Vector Load Instructions	4-23
4-16	Vector Load Instructions Supporting Alignment	4-24
4-17	Shift Values for **lvsl** Instruction	4-24
4-18	Shift Values for **lvsr** Instruction	4-25
4-19	Vector Load (for misaligned vectors) Instructions	4-26
4-20	lvsm Instructions Supporting Alignment	4-27
4-21	Vector Store Instructions	4-27
4-22	Vector Store (for misaligned vectors) Instructions	4-28
4-23	Vector Pack Instructions	4-32
4-24	Vector Unpack Instructions	4-33
4-25	Vector Merge Instructions	4-35
4-26	Vector Splat Instructions	4-36
4-27	Vector Permute Instruction	4-36
4-28	Vector Select Instruction	4-36
4-29	Vector Shift Instructions	4-37
4-30	Coding Various Shifts and Rotates with the **vsidoi** Instruction	4-37
4-31	Move To/From Vector Status and Control Register Instructions	4-39
4-32	Move To Vector Register from GPR Instructions	4-39

Tables

Table Number	Title	Page Number
6-1	Instruction Syntax Conventions	6-1
6-2	Notation and Conventions	6-2
6-3	Instruction Field Conventions	6-6
6-4	Precedence Rules	6-7
6-5	Special Values of the Element in **v**B	6-97
6-6	Special Values of the Element in **v**B	6-99
6-7	Special Values of the Element in **v**B	6-153
6-8	Special Values of the Element in **v**B	6-162
A-1	Instructions Sorted by Mnemonic in Decimal Format	7-1
A-2	Instructions Sorted by Mnemonic in Binary Format	7-7
A-3	Instructions Sorted by Opcode in Decimal Format	7-13
A-4	Instructions Sorted by Opcode in Binary Format	7-19
A-5	VA-Form	7-25
A-6	VX-Form	7-25
A-7	X-Form	7-28
A-8	VXR-Form	7-30

About This Book

The primary objective of this manual is to help programmers provide software that is compatible with processors that implement Power ISA™ and the AltiVec (category Vector) technology. This book describes how AltiVec technology is defined for Freescale processors that implement Power ISA.

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

This book is one of two that discuss the AltiVec technology for Freescale Power ISA Processors. The two books are as follows.

- *AltiVec Technology Programming Interface Manual for Power ISA™ Processors* (ALTIVECPOWERISAPIM) is a reference guide for high-level programmers. The AltiVec Power ISA PIM describes how programmers can access AltiVec functionality from programming languages such as C and C++. The AltiVec Power ISA PIM defines a programming model for use with the AltiVec instruction set.
- *AltiVec Technology Programming Environments Manual for Power ISA™ Processors* (ALTIVECPOWERISAPEM) is used as a reference guide for assembler programmers. The AltiVec Power ISA PEM uses a standardized format to describe each instruction, showing syntax, instruction format, register translation language (RTL) code that describes how the instruction works, and a listing of which, if any, registers are affected. At the bottom of each instruction entry is a figure that shows the operations on elements within source operands and where the results of those operations are placed in the destination operand.

Most of the discussions on the AltiVec technology in this book is related to user-level features. Supervisor level features such as interrupts related to AltiVec technology are more thoroughly discussed in the *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture® Processors*. For ease in reference, this book has arranged the architecture information into topics that build on one another, beginning with a description and complete summary of registers and instructions and progressing to more specialized topics such as the cache, exception, and memory management models.

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

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

Audience

This manual is intended for system software and hardware developers and application programmers who want to develop products using AltiVec technology from Power Architecture® processors. It is assumed that the reader understands operating systems, microprocessor system design, and the basic principles of RISC processing and details of Power ISA.

Organization

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

- Chapter 1, "Overview," is useful for those who want a general understanding of the features and functions of the AltiVec technology. This chapter provides an overview of how the AltiVec technology defines the register set, operand conventions, addressing modes, instruction set, cache model, and interrupt model.
- Chapter 2, "AltiVec Register Set," is useful for software engineers who need to understand the PowerPC® programming model for the three programming environments. The chapter also discusses the functionality of the AltiVec technology registers and how they interact with the other Power ISA registers.
- Chapter 3, "Operand Conventions," describes how the AltiVec technology interacts with the Power ISA conventions for storing data in memory, including information regarding alignment, and single-precision floating-point conventions.
- Chapter 4, "Addressing Modes and Instruction Set Summary," provides an overview of the AltiVec technology addressing modes and a brief description of the AltiVec technology instructions organized by function.
- Chapter 5, "Cache, Interrupts, and Memory Management," provides a discussion of the cache, memory model, and interrupt model as defined by Power ISA.
- Chapter 6, "AltiVec Instructions," functions as a handbook for the AltiVec instruction set. Instructions are sorted by mnemonic. Each instruction description includes the instruction formats and figures where it helps in understanding what the instruction does.
- Appendix A, "Revision History," lists the major differences between revisions of the *AltiVec Technology Programming Environments Manual for Power ISA™ Processors*.
- Appendix A, "AltiVec Instruction Set Listings," list all of the AltiVec instructions, grouped according to mnemonic, opcode, and form, in both decimal and binary order.
- This manual also includes a glossary and an index.

Suggested Reading

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

General Information

The following documentation, available through Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information about the PowerPC architecture and computer architecture in general:

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

- *Computer Organization and Design: The Hardware/Software Interface*, Third Edition, David A. Patterson and John L. Hennessy.

Related Documentation

- *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture® Processors*—Describes resources defined by the EIS, Freescale's implementation of Power ISA.
- Reference manuals—These manuals provide details about individual implementations and are intended for use with the *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture® Processors.*
- Addenda/errata to reference manuals—Because some processors have follow-on parts, an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding reference manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Product brief—Each device has a product brief that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's reference manual.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to http://www.freescale.com.

Conventions

This document uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold
italics	Italics indicate variable command parameters, for example, **bcctr***x*
	Book titles in text are set in italics
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify a source general-purpose register (GPR)
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source floating-point register (FPR)
frD	Instruction syntax used to identify a destination FPR
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets.
vA, **v**B, **v**C	Instruction syntax used to identify a source vector register (VR)
vD	Instruction syntax used to identify a destination VR

x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.
x	An italicized *x* indicates an alphanumeric variable
n	An italicized *n* indicates an numeric variable
¬	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 ones or zeros, they are always read as zeros.

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

Acronyms and Abbreviations

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

Table i. Acronyms and Abbreviated Terms

Term	Meaning
AltiVec PEM	*AltiVec Technology Programming Environments Manual*
AltiVec PowerISA PEM	*AltiVec Technology Programming Environments Manual for Power ISA™ Processors*
AltiVec PIM	*AltiVec Technology Programming Interface Manual*
AltiVec PowerISA PIM	*AltiVec Technology Programming Interface Manual for Power ISA™ Processors*
ALU	Arithmetic logic unit
CR	Condition register
CTR	Count register
DEC	Decrementer register
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

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
IU	Integer unit
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
LSQ	Least-significant quad-word
lsq	Least-significant quad-word
LSU	Load/store unit
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSQ	Most-significant quad-word
msq	Most-significant quad-word
MSR	Machine state register
NaN	Not a number
NIA	Next instruction address
No-op	No operation
OEA	Operating environment architecture
PEM	*Programming Environments Manual*
PMC*n*	Performance monitor counter register
PTE	Page table entry
PVR	Processor version register
RISC	Reduced instruction set computing
RTL	Register transfer language
SIMM	Signed immediate value
SPR	Special-purpose register
TB	Time base facility
TBL	Time base lower register
TBU	Time base upper register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value
UISA	User instruction set architecture

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
UPMC*n*	User performance monitor counter registers
VA	Virtual address
VEA	Virtual environment architecture
VPU	Vector permute unit
VR	Vector register
VSCR	Vector status and control register
XER	Register used for indicating conditions such as carries and overflows for integer operations

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
Extended mnemonics	Simplified mnemonics
Fixed-point unit (FXU)	Integer unit (IU)
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
Store in	Write back
Store through	Write through

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

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)

Table iii. Instruction Field Conventions (continued)

The Architecture Specification	Equivalent to:
FXM	CRM
/, //, ///	0...0 (shaded)
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM
VA, VB, VT, VS	**v**A, **v**B, **v**D, **v**S (respectively)
VEC	AltiVec technology

Chapter 1
Overview

This chapter provides an overview of the AltiVec technology, including general concepts that help in understanding the features that AltiVec technology provides. It also includes information on how AltiVec technology works with Power Architecture® technology.

1.1 History and Classification

AltiVec technology was defined as an extension to the original PowerPC® architecture as defined by Apple, IBM, and Motorola's semiconductor products sector (SPS) (now Freescale). It was internally called VMX (for Vector Multimedia Extensions) and Apple branded it later as the "Velocity Engine", and Motorola (Freescale) branded it as AltiVec. Over the years, only the AltiVec brand has persisted in the public. Beginning in Power ISA™ 2.03, AltiVec technology was formally defined in the architecture as category Vector.

Freescale Embedded Implementation Standards (EIS) define AltiVec technology using category Vector as well, although EIS category Vector includes some instructions not defined in Power ISA, and Power ISA may contain instructions not defined in EIS. Parts of AltiVec technology related to supervisor level software are not fully defined here, but are defined in EREF. EIS is defined by *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture Processors*.

The full content of this document is considered part of category Vector from EIS. Normally individual sections would be marked with the appropriate designation (<V>) but such designations for the Vector category are omitted from this document. When another category or corequisite category is referenced, the appropriate designation for that category appears. For a complete list of categories and category designations, see *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture Processors*.

1.2 Overview

AltiVec technology provides a software model that accelerates the performance of various software applications as they run on Power Architecture® microprocessors. AltiVec ISA is based on separate vector/SIMD-style (single instruction stream, multiple data streams) execution units that have high data parallelism. That is, AltiVec technology operates on multiple data items in a single instruction which allows for a highly efficient way to process large quantities of information. High degrees of parallelism are achievable with simple in-order instruction dispatch and low-instruction time processing. However, the architecture is designed so as not to impede additional parallelism through dispatch to multiple execution units or multithreaded execution unit pipelines.

The term 'vector' in this document refers to the spatial parallel processing of short, fixed-length one-dimensional matrices held in a vector register and performed by an execution unit. It should not be confused with the parallel processing of long, variable-length vectors in memory performed by classical vector machines.

AltiVec technology is an architecture that defines a set of registers and execution units that can be used for efficient SIMD processing. All instructions are designed to be easily pipelined with pipeline latencies no greater than the scalar, single-precision, floating-point multiply-add. AltiVec instructions, like other Power ISA instructions, can be interleaved with any other instructions. The AltiVec technology is defined such that there are few shared resources allowing less complex implementations to be constructed and performance to be optimized. AltiVec technology's SIMD-style extension provides an approach to accelerating the processing of data streams. That is, in SIMD parallel processing, the vector unit will interpret instructions and process multiple pieces of data simultaneously. By processing whole streams of data at once, it provides a fast and efficient way to manipulate large quantities of information. AltiVec instructions provide a significant speed-up for communications, multimedia, signal processing, and other performance-driven applications by using the data-level parallelism and keeping processing of data to the vector register file. By using the SIMD parallelism in AltiVec technology, performance can be accelerated on Power Architecture processors over scalar processing to a level that allows real-time processing of one or more data streams at the same time.

A majority of audio and visual applications require no more than 8- or 16-bit data types to represent satisfactory color and sound. AltiVec ISA can help accelerate the processing of the following types of 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 POTS lines and inserts it onto the Internet. Digital subscriber loop access multiplexer (DSLAM) pulls data off at a switch and immediately routes it to the Internet. This allows it to concentrate ADSL digital traffic at the switch and offload the network.
- Speech recognition. Speech processing allows voice recognition for use in applications such as directory assistance and automatic dialing.
- Voice/sound processing (audio encode and decode): Voice processing uses signal processing to improve sound quality on lines.
- Communications:
  - Multi-channel modems
  - Modem banks can use AltiVec technology to replace signal processors in DSP farms
- 2D and 3D graphics: Arcade-type games
- Image and video processing: JPEG, filters, video encoding and decoding
- Echo cancellation. Echo cancellation is used to eliminate echo on long delay calls (250–500 ms, as in satellite communications).
- Array number processing
- Base-station processing: Cellular base station compresses digital voice data for transmission within the Internet.
- Video conferencing: H.261, H.263

In this document, the term 'implementation' refers to a hardware device (typically a microprocessor) that complies with EIS and Power ISA.

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

- Programming model
  - Instruction set.
    The AltiVec instruction set specifies instructions that extend the scalar instruction set. These instructions are organized similar to the other Power ISA instructions (vector integer, vector floating-point, vector load/store, and vector permutation and formatting instructions). The specific instructions, and the forms used for encoding them, are provided in Appendix A, "AltiVec Instruction Set Listings."
  - Register set.
    The AltiVec programming model defines vector registers, a vector status and control register (VSCR), an SPR for software to specify which vector registers are considered non-volatile, and interrupt vector offset registers (IVORs) to specify the address of where an AltiVec interrupt executes. The model also addresses memory conventions including details regarding the byte ordering for quad words.
- Memory model.
  AltiVec technology originally specified additional cache management instructions for software-directed data prefetching. Such instructions are deprecated, but are still defined in this document. Newer implementations will treat these instructions as no-ops.
- Interrupt model.
  AltiVec technology provides very few interrupts, so processing is efficient. AltiVec interrupts are defined fully in EREF, but are partially described in this document for convenience.
- Memory management model.
  The memory model for AltiVec technology is the same as that described in EREF. AltiVec memory accesses are always performed to a naturally aligned vector address. Some load and store instruction use a misaligned address to reorder the data in vector register for a load or reorder data in memory for a store. Such reordering helps the programmer deal more easily with vectors in memory that are not aligned.

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

1.3 AltiVec Technology Overview

AltiVec technology can be thought of as a set of registers and execution units that are a category of Power ISA analogous to how floating-point registers and execution units are a category of Power ISA. Floating-point instructions provide support for high-precision scientific calculations, and AltiVec instructions accelerate the next level of performance-driven, high-bandwidth communications and computing applications. Figure 1-1 provides a high-level overview of an example implementation with AltiVec technology.

AltiVec technology is purposefully simple so that there are minimal exceptions, no hardware misaligned access support, and no complex functions. AltiVec technology is scaled down to the necessary pieces only, in order to facilitate efficient cycle time, latency, and throughput on hardware implementations.

AltiVec technology defines the following:

- Fixed 128-bit-wide vector length that can be subdivided into sixteen 8-bit bytes, eight 16-bit half-words, or four 32-bit words
- Vector register file (VRF) separate from floating-point registers (FPRs) and general-purpose registers (GPRs)
- Vector integer and floating-point arithmetic
- Four operands for some instructions (three source operands and one result)
- Saturation clamping. That is, unsigned results are clamped to zero on underflow and to the maximum positive integer value ($2^n–1$), for example, 255 for byte fields on overflow. For signed results, saturation clamps results to the smallest representable negative number ($–2^{n–1}$), for example, –128 for byte fields) on underflow, and to the largest representable positive number ($2^{n–01}–1$), for example, +127 for byte fields on overflow.
- Operations selected based on utility to digital signal processing algorithms (including 3D)

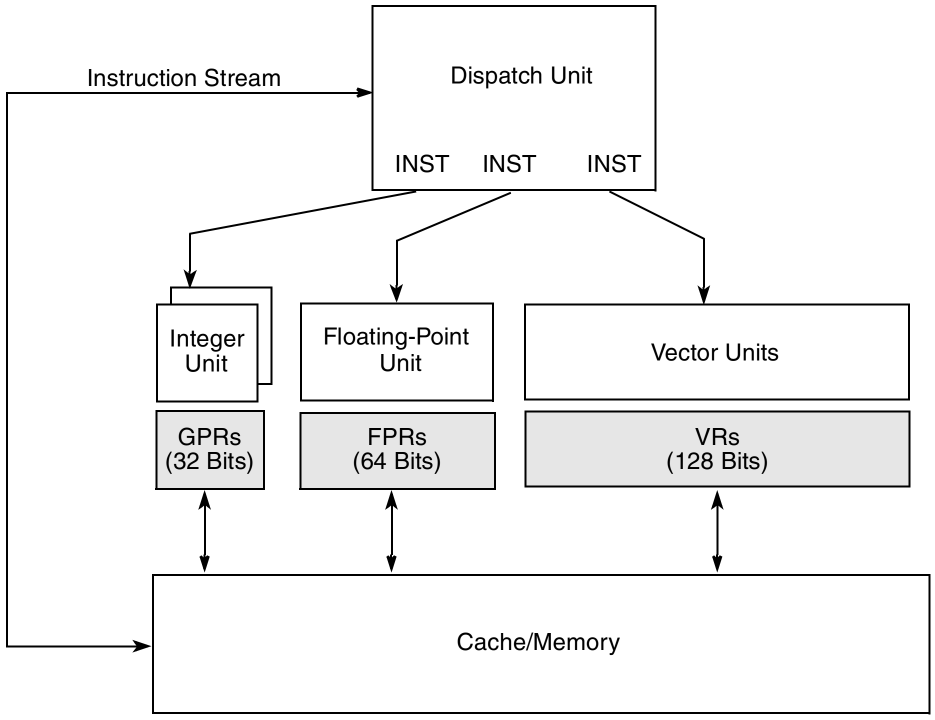


Figure 1-1. Overview of an example implementation with AltiVec Technology

- AltiVec instructions provide a vector compare and select mechanism to implement conditional execution as the preferred way to control data flow in AltiVec programs.
- Instructions for storing and loading vectors to and from memory

1.3.1 Features Not Defined by AltiVec Architecture

Because flexibility is an important design goal of AltiVec technology, there are many aspects of the microprocessor design, typically relating to the hardware implementation, that AltiVec architecture does not define. For example, the number and the nature of execution units are not defined. AltiVec ISA is a vector/SIMD architecture, and as such makes it easier to implement pipelining instructions and parallel execution units to maximize instruction throughput. However, AltiVec architecture 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.

1.4 AltiVec Programming Model

This section provides overviews of the following aspects defined by the AltiVec architecture, which are discussed in further detail throughout the rest of the book:

- AltiVec Registers and Programming Model
- Operand Conventions
- AltiVec Element Operations and AltiVec Instruction Set
- AltiVec Interrupt Model

1.4.1 AltiVec Registers and Programming Model

In AltiVec technology, the ALU operates on from one to three source vectors and produces a single destination vector on each instruction. The ALU is a SIMD-style arithmetic unit that performs the same operation on all the data elements comprising 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. An example vector unit and vector register file are shown in Figure 1-2.

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

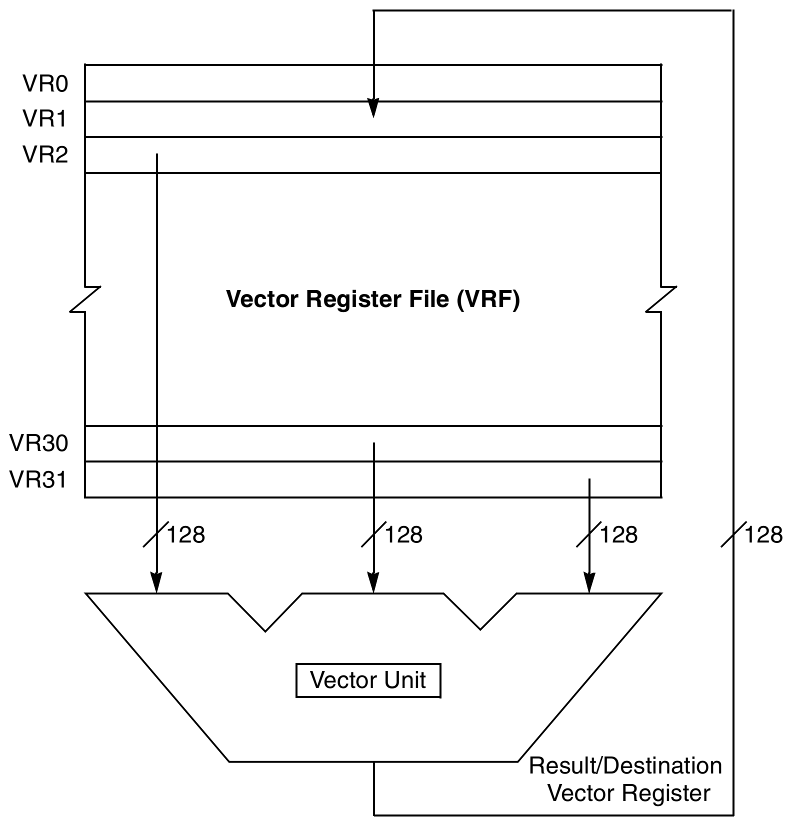


Figure 1-2. AltiVec Top-Level Diagram

1.4.2 Operand Conventions

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

1.4.2.1 Byte Ordering

The default memory access ordering for AltiVec ISA is big-endian. AltiVec load and store instructions always treat memory as big-endian regardless of the endian mode specified.

Big-endian byte ordering is shown in Figure 1-3.

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

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

As shown in Figure 1-3, the elements in vector registers are numbered using big-endian byte ordering. For example, the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15.

When defining high order and low order for elements in a vector register, be careful not to confuse its meaning based on the bit numbering. That is, in Figure 1-4, the high-order half-word for word 0 (bits 0–31) would be half-word 0 (bits 0–15), and the low-order half-word for word 0 would be half-word 1 (bits 16–31).

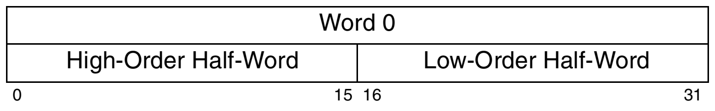


Figure 1-4. Bit Ordering

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

1.4.2.2 Floating-Point Conventions

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

1.4.3 AltiVec Element Operations

AltiVec ISA supports both intra-element and inter-element operations. In an intra-element operation, elements work in parallel with 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 intra-element operation is the Vector Add Signed Word Saturate (**vaddsws)** instruction shown in Figure 1-5.

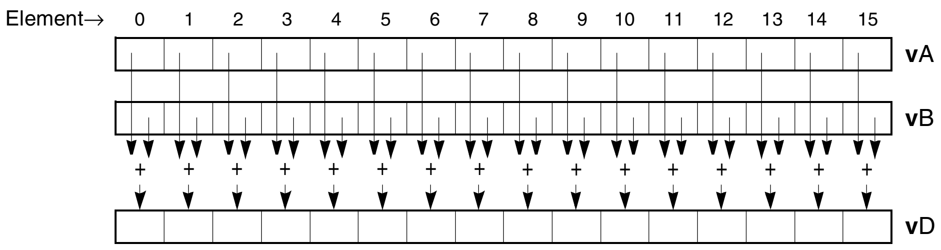
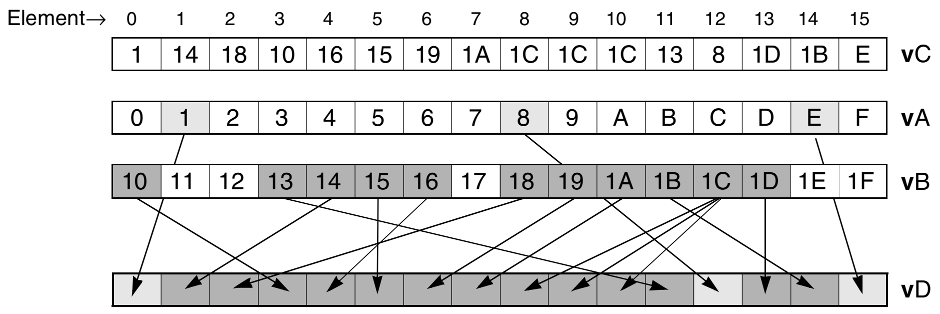


Figure 1-5. Intra-Element Example, vaddsws

In this example, the 16 elements (8 bits per element) in register **v**A are added to the corresponding 16 elements (8 bits per element) in register **v**B and the 16 results are placed in the corresponding elements in register **v**D.

In inter-element operations, data paths cross over. That is, different elements from each source operand are used in the resulting destination operand. An example of an inter-element operation is the Vector Permute (**vperm**) instruction shown in Figure 1-6.



Element→ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 14 18 10 16 15 19 1A 1C 1C 1C 13 8 1D 1B E
vC
0 1 2 3 4 5 6 7 8 9 A B C D E F
vA
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
vB
vD

Figure 1-6. inter-element Example, vperm

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

Most arithmetic and logical instructions are intra-element operations. The crossover data paths have been restricted as much as possible to the inter-element manipulation instructions (unpack, pack, permute, etc.) with the idea to implement the ALU and shift/permute as separate execution units. The following list of instructions distinguishes between intra-element and inter-element instructions:

- Vector intra-element 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 inter-element 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

1.4.4 AltiVec Instruction Set

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

AltiVec instructions can be grouped as follows:

- Vector integer arithmetic instructions
  - Vector integer arithmetic instructions
  - Vector integer compare instructions
  - Vector integer logical instructions
  - Vector integer rotate and shift instructions
- Vector floating-point arithmetic instructions
  - 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
- Vector permutation and formatting instructions
  - Vector pack instructions
  - Vector unpack instructions
  - Vector merge instructions
  - Vector splat instructions
  - Vector permute instructions
  - Vector select instructions
  - Vector shift instructions
- Processor control instructions—These instructions are used to read and write from the AltiVec status and control register (VSCR).
- Memory control instructions—These instructions are used for managing of caches. The instructions are deprecated and newer implementations treat these as no-ops.

1.4.5 AltiVec Interrupt Model

AltiVec vector instructions generate very few interrupts. An AltiVec unavailable interrupt can be generated if the MSR[SPV] bit is not set when an AltiVec instruction is executed. Some implementations may also produce an AltiVec assist interrupt to have software deal with denormalized floating-point values. AltiVec loads and stores can produce data tlb errors and data storage interrupts like any other load and store instruction.

The AltiVec unit does not report IEEE interrupts; 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.

1.5 Changes from the Original AltiVec Definition

This section summarizes the changes from the original definition of AltiVec as an extension to the PowerPC architecture. The current definition of AltiVec technology applies to Freescale Power ISA processors.

The original definition of AltiVec is described in AltiVec Technology Programming Environments Manual Rev.3 (for PowerPC Architecture). The differences between the original definition and this document are summarized in the table below.

Table 1-1. Summary of Changes from Original AltiVec Definition

Feature	Original AltiVec Definition	Power ISA AltiVec Definition	Description
Little-endian	Supported	Not Supported	Little-endian byte ordering is not supported on Power ISA AltiVec definition.
Data stream instructions	Supported	Not Supported	**dss**, **dssall**, **dst**, **dstt**, **dstst**, and **dststt** instructions are not supported on Power ISA AltiVec definition.
IVORs	Not Supported	Supported	IVORs added for AltiVec unavailable interrupt and AltiVec assist interrupt.
Move from GPR to VR	Not Supported	Supported	**mvidsplt** <64> and **mviwsplt** instructions move data from 2 GPRs into a vector register.
Absolute differences	Not Supported	Supported	Absolute difference instructions **vabsdub**, **vabsduh**, and **vabsduw** compute the unsigned absolute differences. These are useful for motion estimation in video processing.
Extended support for misaligned vectors	Not Supported	Supported	Load vector to left and right (**lvtlx**[**l**], **lvtrx**[**l**]), load vector with left-right swap (**lvswx**[**l**]), load vector for swap merge (**lvsm**). Store vector from left and right (**stvflx**[**l**], **stvfrx**[**l**]), store vector with left-right swap (**stvswx**[**l**]).

Table 1-1. Summary of Changes from Original AltiVec Definition

Feature	Original AltiVec Definition	Power ISA AltiVec Definition	Description
Extended support for handling head and tail of vectors	Not Supported	Supported	Load vector element indexed [byte, half-word, word] indexed (**lvexbx**, **lvexhx**, **lvexwx**) loads specified elements from an arbitrary address zeroing the rest of the register. Store vector element indexed [byte, half-word, word] indexed (**stvexbx**, **stvexhx**, **stvexwx**) stores specified elements to an arbitrary address.
External PID instructions for loading and storing VRs	Not Supported	Supported	Load and store vector by external PID (**lvepx**[**l**], **stvepx**[**l**]) for moving data efficiently accross address spaces.

Chapter 2
AltiVec Register Set

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

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

Software should always ensure that 0s are only written to reserved bits in registers.

2.1 AltiVec Register Set Overview

AltiVec registers, shown in Figure 2-1, can be accessed by user- or supervisor-level instructions. The vector registers (VRs) are accessed as instruction operands. Access to the registers can be explicit (that is, through the use of specific instructions for that purpose such as Move from Vector Status and Control Register (**mfvscr)** and Move to Vector Status and Control Register (**mtvscr**) instructions) or implicit as part of the execution of an instruction. Condition register field 6 (**crf**6) can be set implicitly by AltiVec instruction which have the record (Rc) bit set.

The number to the right of the register name indicates the number used in the syntax of the instruction operands to access the register (for example, the number used to access the VRSAVE is SPR 256).

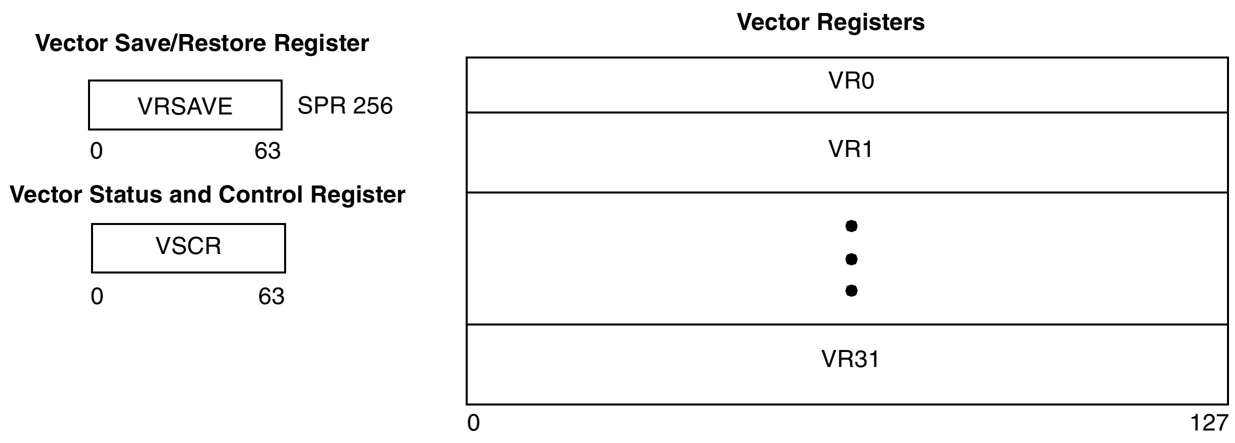

Vector Save/Restore Register
VRSAVE
SPR 256
0
63
Vector Status and Control Register
VSCR
0
63
Vector Registers
VR0
VR1
VR31
0
127

Figure 2-1. AltiVec Register Set

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

- Vector registers (VRs): The vector register file consists of 32 VRs designated as VR0–VR31. The VRs serve as vector source and vector destination registers for all vector instructions. See Section 2.2.1, “AltiVec Vector Register File (VRF),” for more information.
- 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 (VSCR),” 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. The bits in the VRSAVE can indicate whether the vector register is live (1) or dead (0). See Section 2.2.3, “Vector Save/Restore Register (VRSAVE),” for more information.

2.2 Registers Defined by AltiVec

2.2.1 AltiVec Vector Register File (VRF)

The VRs, shown in Figure 2-2, consist of 32 registers, each 128 bits wide. Each vector register can hold sixteen 8-bit elements, eight 16-bit elements, or four 32-bit elements.

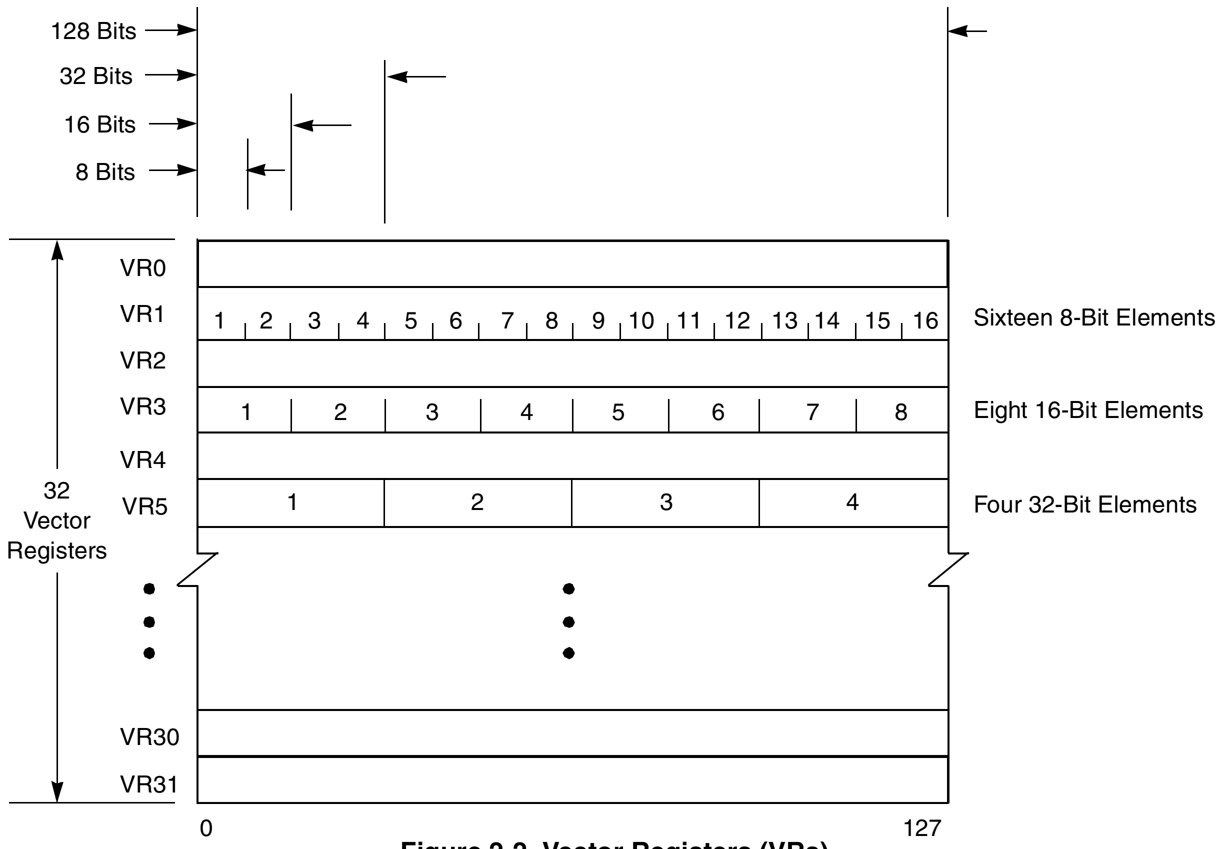


Figure 2-2. Vector Registers (VRs)

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

2.2.2 Vector Status and Control Register (VSCR)

The vector status and control register (VSCR) is a 32-bit register (shown using 128-bit numbering format) that is read and written in a manner similar to the FPSCR in the floating-point category. The VSCR is shown in Figure 2-3.

	96–110	111	112–126	127
Field	Reserved	NJ	Reserved	SAT
Reset	Implementation Specific			
R/W	R/W with **mfvscr** or **mtvscr** Instruction			

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

The VSCR has two defined bits, the AltiVec non-Java mode (NJ) bit (VSCR[111]) and the AltiVec saturation (SAT) bit (VSCR[127]); 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 contents of VSCR from and to a vector register. When moved to or from a vector register, the 32-bit VSCR is right-justified in the 128-bit vector register. When moved to a vector register, the upper 96 bits VR*n* [0–95] of the vector register are cleared, so the VSCR in a vector register looks as shown in Figure 2-4.

0–95	96–110	111	112–126	127
Reserved	Reserved	NJ	Reserved	SAT

Figure 2-4. 32-Bit VSCR Moved to a 128-Bit Vector Register

VSCR bit settings are shown in Table 2-1.

Table 2-1. VSCR Field Descriptions

Bit	Name	Description
96–110	—	Reserved
111	NJ	Non-Java This bit determines whether AltiVec floating-point operations are performed in a Java-IEEE-C9X-compliant mode or a possibly faster non-Java/non-IEEE mode. 0 The Java-IEEE-C9X-compliant mode is selected. Denormalized values are handled as specified by Java, IEEE, and the C9X standard. 1 The non-Java/non-IEEE-compliant mode is selected. If an element in a source vector register contains a denormalized value, the value 0 is used instead. If an instruction causes an underflow exception, the corresponding element in the target VR is cleared to 0. In both cases, the 0 has the same sign as the denormalized or underflowing value. This mode is described in detail in the floating–point overview, Section 3.2.1, “Floating-Point Modes.”

Table 2-1. VSCR Field Descriptions (continued)

Bit	Name	Description
111–126	—	Reserved The handling of reserved bits is the same as that for other Power ISA™ registers. 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.
127	SAT	Saturation A sticky status bit indicating that some field in a saturating instruction saturated since the last time SAT was cleared. In other words, when SAT = 1 it remains set to 1 until it is cleared to 0 by an **mtvscr** instruction. For further discussion refer to Section 4.2.1.1, "Saturation Detection." 0 Indicates no saturation occurred; **mtvscr** can explicitly clear this bit. 1 The AltiVec saturate instruction is set when saturation occurs for the results of one of AltiVec instructions having saturate in its name as follows: Move to VSCR (**mtvscr**) Vector Add Integer with Saturation (**vaddubs**, **vadduhs**, **vadduws**, **vaddsbs**, **vaddshs**, **vaddsws**) Vector Subtract Integer with Saturation (**vsububs**, **vsubuh**s, **vsubuws**, **vsubsbs**, **vsubshs**, **vsubsws**) Vector Multiply-Add Integer with Saturation (**vmhaddshs**, **vmhraddshs**) Vector Multiply-Sum with Saturation (**vmsumuhs**, **vmsumshs**, **vsumsws**) Vector Sum-Across with Saturation (**vsumsws**, **vsum2sws**, **vsum4sbs**, **vsum4shs**, **vsum4ubs**) Vector Pack with Saturation (**vpkuhus**, **vpkuwus**, **vpkshus**, **vpkswus**, **vpkshss**, **vpkswss**) Vector Convert to Fixed-Point with Saturation (**vctuxs**, **vctsxs**)

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

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

2.2.3 Vector Save/Restore 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. The VRSAVE is SPR 256 and is entirely maintained and managed by software.

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Field	VR0	VR1	VR2	VR3	VR4	VR5	VR6	VR7	VR8	VR9	VR10	VR11	VR12	VR13	VR14	VR15
Reset	0000_0000_0000_0000															
R/W	R/W with **mfspr** or **mtspr** Instruction															

	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Field	VR16	VR17	VR18	VR19	VR20	VR21	VR22	VR23	VR24	VR25	VR26	VR27	VR28	VR29	VR30	VR31
Reset	0000_0000_0000_0000															
R/W	R/W with **mfspr** or **mtspr** Instructions															
SPR	SPR 256															

Figure 2-5. Vector Save/Restore Register (VRSAVE)

VRSAVE bit settings are shown in Figure 2-2.

Table 2-2. VRSAVE Bit Settings

Bits	Name	Description
32–63	VR*n*	Each bit in the VRSAVE register indicates whether the corresponding VR contains data in use by the executing process. 0 VR*n* is not being used for the current process 1 VR*n* is using VR*n* for the current process

The VRSAVE register can be accessed only by the **mfspr** and **mtspr** instructions. Each bit in this register corresponds to a vector register (VR) and indicates whether the corresponding register contains data that is currently in use by the executing process. Therefore, the operating system needs to save and restore only those VRs when an interrupt occurs. If this approach is taken, it must be applied rigorously; if a program fails to indicate that a given VR is in use, software errors may occur that are difficult to detect and correct because they are timing-dependent. Some operating systems save and restore VRSAVE only for programs that also use other AltiVec registers.

2.2.4 AltiVec IVOR Registers

Interrupt vector offset registers (IVOR) are used specify the lower 16 bits of the address of where execution should resume when an interrupt occurs. AltiVec defines 2 interrupts and the associated IVOR register which determine where execution resumes as the result of an AltiVec interrupt. Table 2-3 shows the IVORs and their associated interrupts. IVORs are more completely described in EREF.

Table 2-3. AltiVec IVOR Registers

IVOR	Interrupt
IVOR32	AltiVec unavailable interrupt
IVOR33	AltiVec assist interrupt vector

2.3 Non-Vector Category Registers Affected

The only user accessible register not defined by category Vector that can be affected by normal AltiVec instruction execution is the condition register (CR). The CR is a 32-bit register, divided into eight 4-bit fields, CR0–CR7, that reflects the results of certain arithmetic operations and provides a mechanism for testing and branching. For more details refer to Chapter 2, "Register Model," in *EREF*.

For AltiVec instructions, the CR6 field can be set if an AltiVec instruction field's record bit (Rc) is set in a vector compare instruction. CR6 is updated according to Table 2-4.

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

CR Bit	CR6 Field Bit	Vector Compare	Vector Compare Bounds
56	0	1 Relation is true for all element pairs	0
57	1	0	0
58	2	1 Relation is false for all element pairs 0 All fields were in bounds	1 All fields are in bounds for the **vcmpbfp** instruction so the result code of all fields is 0b00 0 One of the fields is out of bounds for the **vcmpbfp** instruction
59	3	0	0

The Rc bit should be used sparingly because when Rc = 1 it can cause a somewhat longer latency or be more disruptive to instruction pipeline flow than when Rc = 0. Therefore, techniques of accumulating results and testing infrequently are advised.

2.4 AltiVec Specific Fields in Supervisor Registers

The machine state register AltiVec available bit (MSR[SPV]) is provided to allow the operating system to be notified when an attempt is made to execute an AltiVec instruction. When MSR[SPV] is clear an attempt to execute an AltiVec instruction will result in an AltiVec unavailable interrupt. This allows the operating system to only save and restore AltiVec register state (VRs, VSCR) for processes which are using the AltiVec facility. This bit is also present in interrupt save/restore registers which are used to save MSR state when an interrupt occurs. The MSR and save/restore registers are only accessible in supervisor state. For more details on the MSR and save/restore registers, see Chapter 2, "Register Model," in *EREF*.

The exception syndrome register SPV bit (ESR[SPV] and GESR[SPV]) is provided to allow the operating system to detect that the exception status was caused by the attempt to execute an AltiVec instruction when an exception and subsequent interrupt occurs. For more details, see Chapter 2, "Register Model," in *EIS*.

Chapter 3
Operand Conventions

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

3.1 Data Organization in Memory

In addition to supporting byte, half-word, and word operands as defined in EIS, AltiVec supports quad-word (128-bit) operands.

The following sections describe the concepts of alignment and byte ordering of data for quad words; otherwise, alignment is the same as described in Chapter 3, "Operand Conventions," and in Chapter 4, "Instruction Model" in *EREF 2.0: A Programmer's Reference Manual for Freescale for Power Architecture Processors.*

3.1.1 Aligned and Misaligned Accesses

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

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) [1]
Byte	8 bits (1 byte)	xxxx
Half-word	2 bytes	xxx0
Word	4 bytes	xx00
Quad word	16 bytes	0000

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

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

It is important to understand that AltiVec memory operands are assumed to be aligned, and AltiVec memory accesses are performed as if the appropriate number of low-order bits of the specified effective address were zero. This assumption is different from integer and floating-point memory access instructions where alignment is not always assumed. Thus, for AltiVec ISA, the low-order bit of the effective address is ignored for half-word AltiVec memory access instructions, and the low-order 4 bits of the effective address are ignored for quad-word AltiVec memory access instructions. The effect is to load or store the memory operand of the specified length that contains the byte addressed by the effective address.

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

3.1.2 AltiVec Byte Ordering

The smallest addressable memory unit is a byte (8 bits), and scalars are composed of one or more sequential bytes. AltiVec supports only big-endian byte ordering, however processors compliant with PowerPC® architecture prior to Power ISA™ also supported little-endian byte ordering. Little-endian byte ordering is not discussed in this book. Users should refer to *AltiVec Technology Programming Environments Manual* Rev.3 (for PowerPC processors) for little-endian usage.

Power ISA and EIS allow for little-endian data accesses as an attribute for a page, however, AltiVec instructions ignore this attribute and always load and store data in a big-endian format.

3.1.2.1 Big-Endian Byte Ordering

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

3.1.3 Quad Word Byte Ordering Example

The idea of big-endian byte ordering is best illustrated in an example of a quad word such as 0x0011_2233_4455_6677_8899_AABB_CCDD_EEFF located in memory. This quad word is used throughout this section to demonstrate how the bytes that comprise a quad word are mapped into memory.

The quad word (0x0011_2233_4455_6677_8899_AABB_CCDD_EEFF) is shown in big-endian mapping in Figure 3-1. A hexadecimal representation is used for showing address values and the values in the contents of each byte. The address is shown below each byte's contents. The big-endian model addresses the quad word at address 0x00, which is the MSB (0x00), proceeding to the address 0x0F, which contains the LSB (0xFF).

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Quad Word															
Contents	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
	↑ MSB															↑ LSB

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

3.1.4 Vector Register and Memory Access Alignment

When loading an aligned byte, half-word, or word memory operand into a vector register, the element that receives the data is the element that would have received the data had the entire aligned quad word containing the memory operand addressed by the effective address been loaded. Similarly, when an element in a vector register is stored into an aligned memory operand, the element selected to be stored is the element that would have been stored into the memory operand addressed by the effective address had the entire vector register been stored to the aligned quad word containing the memory operand addressed by the effective address (Byte memory operands are always aligned).

For aligned byte, half-word, and word memory operands, if the corresponding element number is known when the program is written, the appropriate vector splat and vector permute instructions can be used to copy or replicate the data contained in the memory operand after loading the operand into a vector register. Vector splat instructions will take the contents of an element in a vector register and replicates them 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 3.1.5, "Quad-Word Data Alignment." Another method is to replicate the element across an entire vector register before storing it into an arbitrary aligned memory operand of the same length; the replication ensures that the correct data is stored regardless of the offset of the memory operand in its aligned quad word in memory.

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

3.1.5 Quad-Word Data Alignment

The AltiVec ISA does not provide for alignment interrupts for loading and storing data. When performing vector loads and stores, the effect is as if the low-order four bits of the address are 0x0, regardless of the actual effective address generated. Because vectors may often be misaligned due to the nature of the algorithm, the AltiVec ISA provides support for post-alignment of quad-word loads and pre-alignment for quad-word stores.

Figure 3-2 shows misaligned vectors in memory for big-endian ordering. The example assumes that the desired vector begins at address 0x03. In the figure, HI denotes high-order quad word, and LO means low-order quad word.

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

Figure 3-2. Misaligned Vector in Big-Endian Mode

Figure 3-2 shows how such misaligned data causes data to be split across aligned quad words; only aligned quad words are loaded or stored by AltiVec load/store instructions. To align this vector, a program must load both (aligned) quad words that contain a portion of the misaligned vector data and then execute a Vector Permute (**vperm**) instruction to align the result.

3.1.5.1 Accessing a Misaligned Quad Word in Big-Endian Mode

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

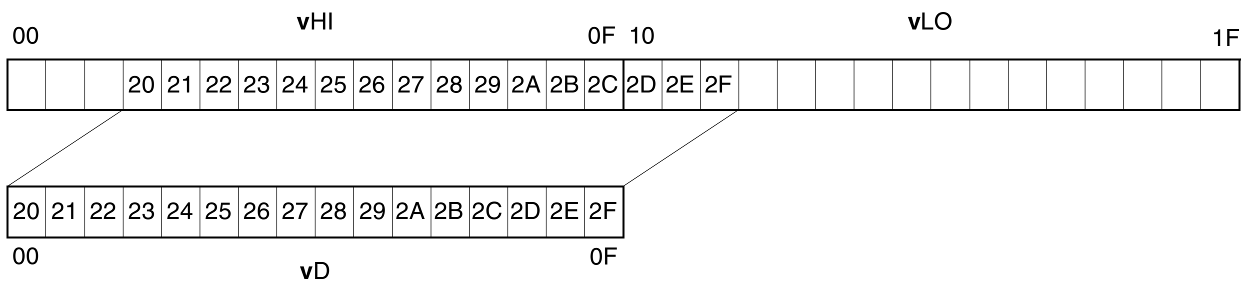

00
vHI
0F
10
vLO
1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
00
vD
0F

Figure 3-3. Big-Endian Quad Word Alignment

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

The complete C code sequence for an unaligned load case is as follows:

```
vector unsigned char vectorLoadUnaligned( vector unsigned char *where)
{
        vector unsigned char permuteVector = vec_lvsl( 0,where );
        vector unsigned char low = vec_ld( 0, where);
        vector unsigned char high = vec_ld( 15, where );
        return vec_perm( low, high, permuteVector );
}
```

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

The process of storing a misaligned vector is a bit different. It is possible to have data change in the process of storing two aligned vectors. To avoid a thread safety problem the data is stored using vector element stores.

The complete C code sequence for an unaligned store case is as follows:

```
void StoreUnaligned( vector unsigned char v, unsigned char *where)
{
        vector unsigned char tmp;
                tmp = vec_perm( v, v, vec_lvsr( 0, where) );
                vec_ste( (vector unsigned char) tmp, 0, (unsigned char*) where);
                vec_ste( (vector unsigned short)tmp,1,(unsigned short*) where);
                vec_ste( (vector unsigned int) tmp, 3, (unsigned int*) where);
                vec_ste( (vector unsigned int) tmp, 4, (unsigned int*) where);
                vec_ste( (vector unsigned int) tmp, 8, (unsigned int*) where);
                vec_ste( (vector unsigned int) tmp, 12, (unsigned int*) where);
                vec_ste( (vector unsigned short)tmp,14,(unsigned short*) where);
                vec_ste( (vector unsigned char) tmp,15,(unsigned char*) where);
}
```

3.1.5.2 Scalar Loads and Stores

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

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

3.1.5.3 Misaligned Scalar Loads and Stores

Although no direct support of misaligned scalars is provided, the load-aligning sequence for big-endian vectors described in Section 3.1.5.1, "Accessing a Misaligned Quad Word in Big-Endian Mode," can be used to position the scalar to the left vector element, which can then be used as the source for a splat. That is, the address of a scalar is also the address of the left-most element of the quad word at that address. Similarly, the read-modify-write sequences, with the mask adjusted for the scalar size, can be used to store misaligned scalars.

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

3.1.6 Mixed-Endian Systems

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

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

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

3.2 AltiVec Floating-Point Instructions

There are two kinds of floating-point instructions defined for Power ISA and AltiVec:

- Computational
- Noncomputational

Computational instructions are defined by IEEE 754 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. AltiVec does not define any floating-point specific load or store instructions, therefore only the computational instructions are considered floating-point operations throughout this chapter.

The single-precision format, value representations, and computational model to be defined in Chapter 3, "Register Model," in the *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture Processors*, apply to AltiVec floating-point except as follows:

- In general, no status bits are set to reflect the results of floating-point operations. The only exception is that VSCR[SAT] may be set by the Vector Convert to Fixed-Point Word instructions.

- With the exception of the two Vector Convert to Fixed-Point Word (**vctuxs**, **vctsxs**) instructions and three of the four Vector Round to Floating-Point Integer (**vrfiz**, **vrfip**, **vrfim**) instructions, all AltiVec floating-point instructions that round use the round-to-nearest rounding mode.
- Floating-point exceptions cannot cause the system error handler to be invoked.

If a function is required that is specified by the IEEE standard, is not supported by AltiVec, and cannot be emulated satisfactorily using the functions that are supported by AltiVec, the functions provided by the scalar floating-point processor should be used; see Chapter 4, “Instruction Model,” in the *EREF 2.0: A Programmer’s Reference Manual for Freescale Power Architecture Processors.*

3.2.1 Floating-Point Modes

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

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

3.2.1.1 Java Mode

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

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

Java compliance requires the following aspects of the IEEE standard:

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

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

3.2.1.2 Non-Java Mode

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

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

3.2.2 Floating-Point Infinities

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

3.2.3 Floating-Point Rounding

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

3.2.4 Floating-Point Exceptions

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

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

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

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

3.2.4.1 NaN Operand Exception

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

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

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

The following actions can be taken:

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

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

- 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.
- If the instruction is Vector Compare Bounds Floating-Point (**vcmpbfp**[**.**]), the corresponding result is 0xC000_0000.
- If the instruction is one of the other three vector floating-point compare instructions (**vcmpeqfp**[**.**], **vcmpfgefp**[.], **vcmpbfp**[.]), the corresponding result is 0x0000_0000.

3.2.4.2 Invalid Operation Exception

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

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

The corresponding result is the QNaN 0x7FC0_0000. This is the single-precision format analogy of the double precision format generated QNaN described in Chapter 3, “Register Model,” in the *EREF 2.0: A Programmer’s Reference Manual for Freescale Power Architecture Processors.*

3.2.4.3 Zero Divide Exception

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

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

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

3.2.4.4 Log of Zero Exception

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

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

3.2.4.5 Overflow Exception

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

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

The following actions can be taken:

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

3.2.4.6 Underflow Exception

Underflow exceptions occur only for AltiVec instructions that would normally produce floating-point results. Underflow is detected before rounding. Underflow 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 can be taken:

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

3.2.5 Floating-Point NaNs

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

3.2.5.1 NaN Precedence

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

3.2.5.3 QNaN Arithmetic

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

3.2.5.4 NaN Conversion to Integer

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

3.2.5.5 NaN Production

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

Chapter 4
Addressing Modes and Instruction Set Summary

This chapter describes instructions and addressing modes defined by AltiVec. These instructions are divided into the following categories:

- Vector integer arithmetic instructions
  These include arithmetic, logical, compare, rotate, and shift instructions, described in Section 4.2.1, “Vector Integer Instructions.”
- Vector floating-point arithmetic instructions
  These include floating-point arithmetic instructions as well as a discussion on floating-point modes, described in Section 4.2.2, “Vector Floating-Point Instructions.”
- Vector load and store instructions
  These include load and store instructions for vector registers, described in Section 4.2.3, “Load and Store Instructions.”
- Vector permutation and formatting instructions
  These include pack, unpack, merge, splat, permute, select, and shift instructions, described in Section 4.2.4, “Vector Permutation and Formatting Instructions.”
- Processor control instructions
  These instructions are used to read and write from the AltiVec Status and Control Register, described in Section 4.2.4.8, “AltiVec Status and Control Register Instructions.”

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

AltiVec integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision operands. AltiVec provides for byte, half-word, and word operand fetches and stores between memory and the vector registers (VRs).

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation for an arithmetic or logical instruction, the following steps are taken:

1. The memory contents must be loaded into a register with a load instruction.
2. The contents are then modified.
3. The modified contents are written to the target location using a store instruction.

4.1 Conventions

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

4.1.1 Computation Modes

AltiVec supports both 32-bit and 64-bit implementations. AltiVec instructions behave the same, regardless of the computation mode, except that effective addresses (EAs) use only the low-order 32 bits of the computed EA when addressing memory on 32-bit mode and use all 64 bits of the EA in 64-bit mode when load and store instructions are executed.

4.1.2 Classes of Instructions

AltiVec instructions are considered “defined” instructions per the description of instructions in *EREF*. Reserved fields in instructions are ignored by the processor as they are for any other instructions in Power ISA™.

4.1.3 Memory Addressing

Memory addressing for AltiVec instructions is the same as other Power ISA load and store instructions. EAs, RAs, and LAs <E.HV> have the same meaning and semantics except that AltiVec loads and stores are performed to naturally aligned boundaries and do not cause alignment exceptions. See chapter 4, “Instruction Model,” in *EREF* for more information.

4.2 AltiVec Instructions

This section discusses the instructions defined by AltiVec.

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 vector registers (VRs) as source operands and place results into VRs as well. Setting the Rc bit of a vector compare instruction causes the condition register (CR, specifically CR field 6) to be updated.

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

4.2.1.1 Saturation Detection

Most integer instructions have both signed and unsigned versions and many have both modulo (wrap-around) and saturating clamping modes. Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero on underflow and to the maximum positive integer value ($2^n – 1$, for example, 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number ($–2^{n–1}$, for example, –128 for byte fields) on underflow, and to the largest representable positive number ($2^{n–1}–1$, for example, +127 for byte fields) on overflow. When a modulo instruction is used, the resultant number truncates overflow or underflow for the length (byte, half-word, word, quad word) and type of operand (unsigned, signed). AltiVec 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**, **vsubuh**s, **vsubuws**, **vsubsbs**, **vsubshs**, **vsubsws**)
- Vector multiply-add integer with saturation (**vmhaddshs**, **vmhraddshs**)
- Vector multiply-sum with saturation (**vmsumuhs**, **vmsumshs**, **vsumsws**)
- Vector sum-across with saturation (**vsumsws**, **vsum2sws**, **vsum4sbs**, **vsum4shs**, **vsum4ubs**)
- Vector pack with saturation (**vpkuhus**, **vpkuwus**, **vpkshus**, **vpkswus**, **vpkshss**, **vpkswss**)
- Vector convert to fixed-point with saturation (**vctuxs**, **vctsxs**)

Note that only instructions that explicitly call for saturation can set VSCR[SAT]. Modulo integer instructions and floating-point arithmetic instructions never set VSCR[SAT]. For further details see Section 2.2.2, “Vector Status and Control Register (VSCR).”

4.2.1.2 Vector Integer Arithmetic Instructions

Table 4-1 lists the AltiVec vector integer arithmetic instructions.

Table 4-1. Vector Integer Arithmetic Instructions

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

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

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

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

Name	Mnemonic	Syntax	Operation
Vector Multiply Even Unsigned Integer [b,h] Modulo	**vmuleub** **vmuleuh**	vD,vA, vB	Place the unsigned integer products of (**v**A) * (**v**B) into **v**D using modulo arithmetic mode. • For **b**, byte, integer length = 8 bits =1 byte, multiply 8 even-numbered unsigned integer byte elements from **v**A to 8 even-numbered unsigned integer byte elements from **v**B resulting in 8 unsigned integer half-word products in v**D**. • For **h**, half-word, integer length = 16 bits = 2 bytes, multiply 4 even-numbered unsigned integer half-word elements from **v**A to 4 even-numbered unsigned integer half-word elements from **v**B resulting in 4 unsigned integer word products in v**D**.
Vector Multiply Even Signed Integer [b,h] Modulo	**vmulesb** **vmulesh**	vD,vA, vB	Place the signed integer product of (**v**A) * (**v**B) into **v**D using modulo arithmetic mode. • For **b**, byte, integer length = 8 bits = 1 byte, multiply 8 even-numbered signed integer byte elements from **v**A to 8 even-numbered signed integer byte elements from **v**B resulting in 8 signed integer half-word products in v**D**. • For **h**, half-word, integer length = 16 bits = 2 bytes, multiply 4 even-numbered signed integer half-word elements from **v**A to 4 even-numbered signed integer half-word elements from **v**B resulting in 4 signed integer word products in v**D**.
Vector Multiply-High and Add Signed Half-Word Saturate	**vmhaddshs**	vD,vA, vB,vC	The 17 most significant bits (msb's)of the product of (**v**A) * (**v**B) adds to sign-extended **v**C and places the result into **v**D. • For **h**, half-word, integer length = 16 bits = 2 bytes, multiply the 8 signed half-words from **v**A with the corresponding 8 signed half-words from **v**B to produce a 32-bit intermediate product and then take the 17 msbs (bits 0–16) of the 8 intermediate products and add them to the 8 sign-extended half-words in **v**C, place the 8 half-word saturated results in **v**D. If the intermediate product is as follows: > (2^{15}–1) saturate to (2^{15}–1) and if < –2^{15} saturate to –2^{15}. If the results saturates, VSCR[SAT] is set.
Vector Multiply-High Round and Add Signed Half-Word Saturate	**vmhraddshs**	vD,vA, vB,vC	Add the rounded product of (**v**A) * (**v**B) to sign-extended **v**C and place the result into **v**D. • For **h**, half-word, integer length = 16 bits = 2 bytes, multiply the eight signed integers from **v**A to the corresponding eight signed integers from **v**B and then round the 8 immediate products by adding the value 0x0000_4000 to it. Then add the most significant bits (msb), bits 0–16, of the 8 rounded immediate products to the 8 sign-extended values in **v**C and place the eight signed half-word saturated results into **v**D. If the intermediate product is: > (2^{15}–1) saturate to (2^{15}–1), or if < –2^{15} saturate to –2^{15}. If the result saturates, VSCR[SAT] is set.
Vector Multiply-Low and Add Unsigned Half-Word Modulo	**vmladduhm**	vD,vA, vB,vC	Add the product of (**v**A) * (**v**B) to zero-extended **v**C and place into **v**D. • For **h**, half-word, integer length =16 bits = 2 bytes, multiply the 8 signed integers from **v**A to the corresponding 8 signed integers from **v**B to produce a 32-bit intermediate product. The 16-bit value in **v**C is zero-extended to 32 bits and added to the intermediate product and the lower 16 bits of the sum (bit 16–31) is placed in **v**D. Note that unsigned or signed integers can be used with these instructions.

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

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

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

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

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

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

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

Name	Mnemonic	Syntax	Operation
Vector Minimum Signed Integer [b,h,w]	**vminsb** **vminsh** **vminsw**	**v**D,**v**A, **v**B	Compare the minimum of **v**A and **v**B signed integers for each integer value and which ever value is smaller, place that signed integer value into **v**D. • For **b**, byte, integer length = 8 bits = 1 byte, compare 16 signed integers from **v**A with 16 signed integers from **v**B. • For **h**, half-word, integer length = 16 bits = 2 bytes, compare 8 signed integers from **v**A with 8 signed integers from **v**B. • For **w**, word, integer length = 32 bits = 4 bytes, compare 4 signed integers from **v**A with 4 signed integers from **v**B.
Vector Absolute Differences Unsigned [Byte, Half-word, Word]	**vabsdub** **vabsduh** **vabsduw**	**v**D,**v**A, **v**B	Place the absolute value of the difference of the unsigned integer values of **v**A and **v**B into **v**D. • For **b**, byte, integer length = 8 bits = 1 byte, compute the absolute value of the difference of 16 unsigned integers from **v**A and 16 unsigned integers from **v**B. • For **h**, half-word, integer length = 16 bits = 2 bytes, compute the absolute value of the difference of 8 unsigned integers from **v**A and 8 unsigned integers from **v**B. • For **w**, word, integer length = 32 bits = 4 bytes, compute the absolute value of the difference of 4 unsigned integers from **v**A and 4 unsigned integers from **v**B.

4.2.1.3 Vector Integer Compare Instructions

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

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

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

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

Table 4-3 summarizes the vector integer compare instructions.

Table 4-3. Vector Integer Compare Instructions

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

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

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

4.2.1.4 Vector Integer Logical Instructions

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

Table 4-4. Vector Integer Logical Instructions

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

4.2.1.5 Vector Integer Rotate and Shift Instructions

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

Table 4-5. Vector Integer Rotate Instructions

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

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

Table 4-6. Vector Integer Shift Instructions

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

4.2.2 Vector Floating-Point Instructions

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

- Arithmetic
- Rounding and conversion
- Compare
- Estimate

The AltiVec floating-point data format complies with the ANSI/IEEE-754 standard. A quantity in this format represents a signed normalized number, a signed denormalized number, a signed zero, a signed infinity, a quiet not a number (QNaN), or a signaling 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." AltiVec ISA does not report IEEE exceptions but rather produces default results as specified by the Java/IEEE/C9X Standard. For further details on exceptions, see Section 3.2.4, "Floating-Point Exceptions."

4.2.2.1 Floating-Point Division and Square-Root

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

4.2.2.1.1 Floating-Point Division

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

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

This is implemented in the AltiVec ISA as follows:

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

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

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

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

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

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

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

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

4.2.2.1.2 Floating-Point Square-Root

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

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

That can be implemented as follows:

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

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

4.2.2.2 Floating-Point Arithmetic Instructions

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

Table 4-7. Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax	Operation
Vector Add Floating-Point	**vaddfp**	vD,vA, vB	Add the 4-word (32-bit) floating-point elements in **v**A to the 4-word (32-bit) floating-point elements in **v**B. Round the 4 intermediate results to the nearest single-precision number and placed into **v**D.
Vector Subtract Floating-Point	**vsubfp**	vD,vA, vB	The 4-word (32-bit) floating-point values in **v**B are subtracted from the four 32-bit values in **v**B. The 4 intermediate results are rounded to the nearest single-precision floating-point and placed into **v**D.

Table 4-7. Floating-Point Arithmetic Instructions (continued)

Name	Mnemonic	Syntax	Operation
Vector Maximum Floating-Point	**vmaxfp**	vD,vA, vB	Compare each of the 4 single-precision word elements in **v**A to the corresponding 4 single-precision word elements in **v**B and place the larger value within each pair into the corresponding word element in **v**D. **vmaxfp** is sensitive to the sign of 0.0. When both operands are ±0.0: max(+0.0,±0.0) = max(±0.0,+0.0) ⇒ +0.0 max(–0.0,–0.0) ⇒ –0.0 max(NaN,x) ⇒ QNaN, where x = any value
Vector Minimum Floating-Point	**vminfp**	vD,vA, vB	Compare each of the 4 single-precision word elements in **v**A to the corresponding 4 single-precision word elements in **v**B For each of the four elements, place the smaller value within each pair into **v**D. **vminfp** is sensitive to the sign of 0.0. When both operands are ±0.0: min(–0.0,±0.0) = min(±0.0,-0.0) ⇒ –0.0 min(+0.0,+0.0) ⇒ +0.0 min(NaN,x) ⇒ QNaN where x = any value

4.2.2.3 Floating-Point Multiply-Add Instructions

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

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

Be careful as Java-compliant programs 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 AltiVec ISA use multiply-add instructions with a zero addend (for example, **vmaddfp v**D,**v**A,**v**C,**v**B where (**v**B = 0.0).

Note that 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 ⇒ +0.0, and –0.0 + -0.0 ⇒ –0.0). When the sign of a resulting 0.0 is not important, then use +0.0 as the addend that may, in some cases, avoiding the need for a second register to hold a –0.0 in addition to the integer 0/floating-point +0.0 that may already be available.

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

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

Name	Mnemonic	Syntax	Operation
Vector Multiply-Add Floating-Point	**vmaddfp**	vD,vA, vC,vB	Multiply the 4-word floating-point elements in **v**A by the corresponding 4-word elements in **v**C. Add the 4-word elements in **v**B to the 4 intermediate products. Round the results to the nearest single-precision numbers and place the corresponding word elements into **v**D.
Vector Negative Multiply-Subtract Floating-Point	**vnmsubfp**	vD,vA, vC,vB	Multiply the 4-word floating-point elements in **v**A by the corresponding 4-word elements in **v**C. Subtract the 4-word floating-point elements in **v**B from the 4 intermediate products and invert the sign of the difference. Round the results to the nearest single-precision numbers and place the corresponding word elements into **v**D.

4.2.2.4 Floating-Point Rounding and Conversion Instructions

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

AltiVec 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). The floating-point rounding instructions are described in Table 4-9.

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

Name	Mnemonic	Syntax	Operation
Vector Round to Floating-Point Integer Nearest	**vrfin**	vD,vB	Round to the nearest the 4-word floating-point elements in **v**B and place the 4 corresponding word elements into **v**D.
Vector Round to Floating-Point Integer Toward Zero	**vrfiz**	vD,vB	Round towards zero the 4-word floating-point elements in **v**B and place the 4 corresponding word elements into **v**D.
Vector Round to Floating-Point Integer Toward Positive Infinity	**vrfip**	vD,vB	Round towards +Infinity the 4-word floating-point elements in **v**B and place the 4 corresponding word elements into **v**D.

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

Name	Mnemonic	Syntax	Operation
Vector Round to Floating-Point Integer Toward Minus Infinity	**vrfim**	vD,vB	Round towards –Infinity the 4-word floating-point elements in **v**B and place the 4 corresponding word elements into **v**D.
Vector Convert from Unsigned Fixed-Point Word	**vcfux**	vD,vB, UIMM	Convert each of the 4 unsigned fixed-point integer word elements in **v**B to the nearest single-precision value. Divide the result by $2^{\mathbf{UIMM}}$ and place into the corresponding word element of **v**D.
Vector Convert from Signed Fixed-Point Word	**vcfsx**	vD,vB, UIMM	Convert each signed fixed-point integer word element in **v**B to the nearest single-precision value. Divide the result by $2^{\mathbf{UIMM}}$ and place into the corresponding word element of **v**D.
Vector Convert to Unsigned Fixed-Point Word Saturate	**vctuxs**	vD,vB, UIMM	Multiply each of the 4 single-precision word elements in **v**B by $2^{\mathbf{UIMM}}$. The products are converted to unsigned fixed-point integers using the Round toward Zero mode. If the intermediate results are $>2^{32}–1$ saturate to $2^{32}–1$ and if it is <0 saturate to 0. Place the unsigned integer results into the corresponding word elements of **v**D.
Vector Convert to Signed Fixed-Point Word Saturate	**vctsxs**	vD,vB, UIMM	Multiply each of the 4 single-precision word elements in **v**B by $2^{\mathbf{UIMM}}$. The products are converted to signed fixed-point integers using Round toward Zero mode. If the intermediate results are $>2^{32}–1$ saturate to $2^{32}–1$ and if it is $<–2^{31}$ saturate to $–2^{31}$. Place the unsigned integer results into the corresponding word elements of **v**D.

4.2.2.5 Floating-Point Compare Instructions

This section describes floating-point unordered compare instructions.

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

Compares always return a Boolean mask (TRUE = 0xFFFF_FFFF, FALSE = 0x0000_0000) and never return a NaN. The **vcmpeqfp** instruction is recommended as the Isnan(**v**X) test. No explicit unordered compare instructions or traps are provided. However, the greater-than-or-equal-to predicate (≥) (**vcmpgefp**) is provided—in addition to the > and = predicates available for integer comparison—specifically to enable IEEE unordered comparison that would not be possible with just the > and = predicates.

Table 4-10 lists the six common mathematical predicates and how they would be realized in AltiVec code.

Table 4-10. Common Mathematical Predicates

Case	Mathematical Predicate	AltiVec Realization	Relations			
			a > b	a < b	a = b	unordered
1	a = b	a = b	F	F	T	F
2	a ≠ b (?<>)	¬ (a = b)	T	T	F	T

Table 4-10. Common Mathematical Predicates (continued)

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

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

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

Table 4-11. Other Useful Predicates

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

The vector floating-point compare instructions 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 **v**A are within the bounds specified by the corresponding element in **v**B, as explained in the instruction description. A single-precision value x is said to be within the bounds specified by a single-precision value y if (–y ≤ x ≤ y).

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

Table 4-12. Floating-Point Compare Instructions

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

4.2.2.6 Floating-Point Estimate Instructions

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

Table 4-13. Floating-Point Estimate Instructions

Name	Mnemonic	Syntax	Operation
Vector Reciprocal Estimate Floating-Point	**vrefp**	vD,vB	Place estimates of the reciprocal of each of the 4-word floating-point source elements in **v**B in the corresponding 4-word elements in **v**D.
Vector Reciprocal Square Root Estimate Floating-Point	**vrsqrtefp**	vD,vB	Place estimates of the reciprocal square-root of each of the 4-word source elements in **v**B in the corresponding 4-word elements in **v**D.

Table 4-13. Floating-Point Estimate Instructions (continued)

Name	Mnemonic	Syntax	Operation
Vector Log2 Estimate Floating-Point	**vlogefp**	vD,vB	Place estimates of the base 2 logarithm of each of the 4-word source elements in **v**B in the corresponding 4-word elements in **v**D.
Vector 2 Raised to the Exponent Estimate Floating-Point	**vexptefp**	vD,vB	Place estimates of 2 raised to the power of each of the 4-word source elements in **v**B in the corresponding 4-word elements in **v**D.

4.2.3 Load and Store Instructions

Basic load and store operations are provided in AltiVec. Load and store instructions are provided to help handle memory vectors which are not aligned to a vector register alignment boundary. Also, a powerful set of field manipulation instructions are provided to manipulate data into the desired alignment and arrangement after the data has been brought into the vector registers.

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

Load vector to left indexed (**lvtlx**, **lvtlxl**) and load vector to right indexed (**lvtrx**, **lvtrxl**) instructions transfer a misaligned partial vector from memory and left justify (right justify) the bytes in the vector register. Similarly, store vector from left indexed (**stvflx**, **stvflxl**) and store vector from right indexed (**stvfrx**, **stvfrxl**) instructions transfer bytes from a vector register left justified (right justified) to a partial vector in memory. Load vector with left-right swap (**lvswx**, **lvswxl**) and store vector with left-right swap(**stvswx**, **stvswxl**) instructions perform the functions of both a to/from left and a to/from right operation with a single load. These instructions can be used to traverse down input and output vectors in memory that are misaligned with respect to each other and misaligned with respect to the size of a vector register.

All vector loads and vector stores use the index (**r**A|0 + **r**B) addressing mode to specify the target memory address. AltiVec does not provide any update forms. An **lvebx**, **lvehx**, or **lvewx** instruction transfers a scalar data element from memory into the destination vector register, leaving other elements in the vector with boundedly-undefined values. A **stvebx**, **stvehx**, or **stvewx** instruction transfers a scalar data element from the source vector register to memory leaving other elements in the quad word unchanged. No data alignment occurs, that is, all scalar data elements are transferred directly on their natural memory byte-lanes to or from the corresponding element in the vector register.

An **lvexbx**, **lvexhx**, or **lvexwx** instruction transfers a scalar data element from memory into a specified element in the vector register and sets the rest of the vector register to 0. Similarly a **stvexbx**, **stvexhx**, or **stvexwx** instruction transfers a specified element in the vector register to memory.

Quad-word memory accesses made by any non-element vector load or store instruction is not guaranteed to be atomic.

<Category E.PD>:
Vector load (**lvepx**, **lvepxl**) and vector store (**stvepx**, **stvepxl**) instructions are provided for doing vector length loads and stores to external address spaces.

4.2.3.1 Alignment

All **lvx**, **lvxl**, **stvx**, and **stvxl** instructions memory references must be size aligned. If 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 by software after being brought into the registers.

The **lvtlx**[**l**], **lvtrx**[**l**], and **lvswx**[**l**] instructions provide assistance for loading misaligned vectors from memory and **stvflx**[**l**], **stvfrx**[**l**], and **stvswx**[**l**] instructions provide assistance for storing misaligned vectors to memory. **lvsm** can be used to create a vector register suitable for using as the control register for a **vsel** operation which will merge the left and right parts of a misaligned vector from previous **lvswx** instructions. **stvexbx**, **stvexhx**, and **stvexwx** instructions can be used for tail end processing of vectors to store the last elements of a partial vector to a specified address in memory. Rudimentary assistance is also provided for justifying non-size-aligned vectors. This is provided through the Load Vector for Shift Left (**lvsl**) and Load Vector for Shift Right (**lvsr**) instructions that compute the proper Vector Permute (**vperm**) control vector from the misaligned memory address. For details on how to use these instructions to align data see Section 3.1.5, “Quad-Word Data Alignment.”

The vector load and store instructions can be used to move data. Therefore, because vector loads and stores are size-aligned, care should be taken to align data on even quad-word boundaries for maximum performance.

On some implementations, **lvx**[**l**], **lvtrx**[**l**], **stvflx**[**l**], and **stvfrx**[**l**] may take an alignment interrupt if the EA is in memory that is Caching-Inhibited or Write-Through Required. In such cases, the operating system should emulate the instruction by loading or storing the specified elements of the vector.

4.2.3.2 Load and Store Address Generation

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

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

Table 4-14. Effective Address Alignment

Operand	Effective Address Bit	Setting
Indexed half-word	EA[63]	0b0
Indexed word	EA[62–63]	0b00
Indexed quad word	EA[60–63]	0b0000

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

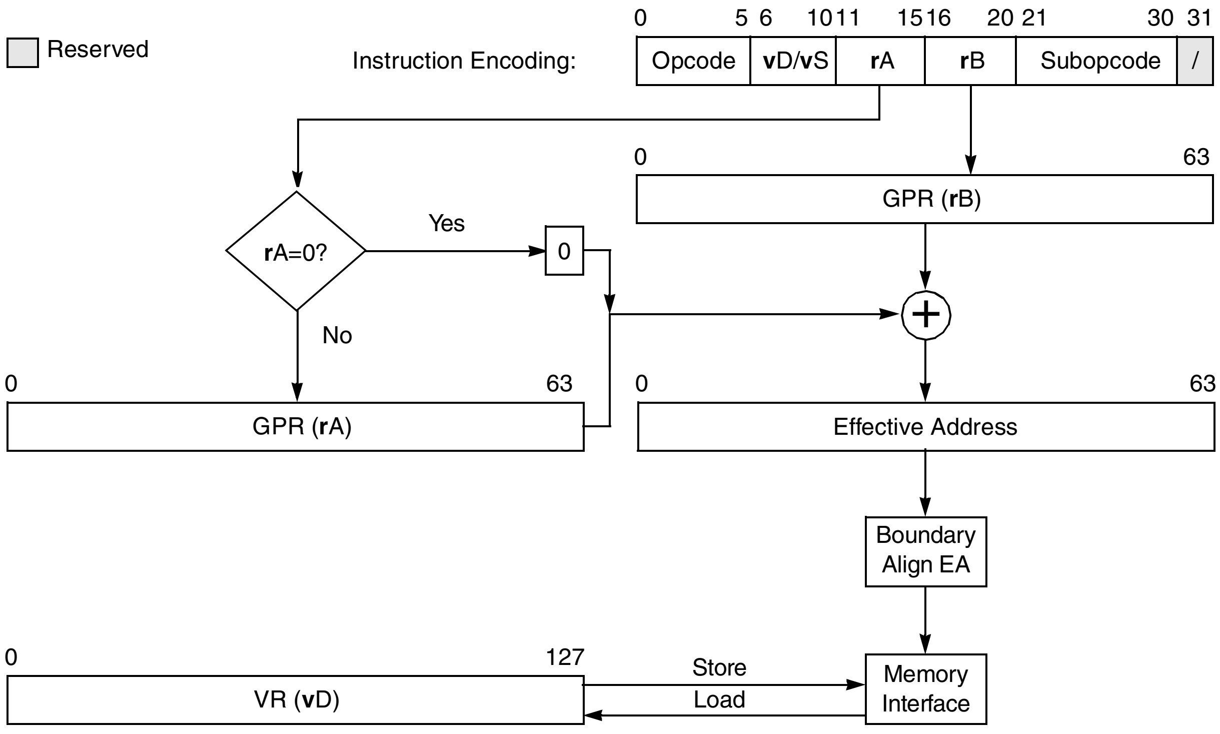


Figure 4-1. Register Indirect with Index Addressing for Loads/Stores

4.2.3.3 Vector Load Instructions

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

Table 4-15 summarizes the vector load instructions.

Table 4-15. Vector Load Instructions

Name	Mnemonic	Syntax	Operation
Load Vector Element Integer Indexed [b,h,w]	**lvebx** **lvehx** **lvewx**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B) truncated to the natural alignment boundary based on the element size (byte, half-word, or word). Load the byte, half-word, or word in memory addressed by EA into the corresponding element of **v**D. The remaining bits in vD are set to boundedly-undefined values. The element of vD is determined by low-order bits of EA based on the element size: • For **b**, byte, 8 bits = 1 byte, element index in vector is EA[60–63] • For **h**, half-word, 16 bits = 2 bytes, element index in vector is EA[60–62] • For **w**, word, 32 bits = 4 bytes, element index in vector is EA[60–61]
Load Vector Indexed	**lvx**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load the quad word in memory addressed by EA & (~0xF) into **v**D.
Load Vector Indexed LRU	**lvxl**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load the quad word in memory addressed by EA & (~0xF) into **v**D. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.

Table 4-15. Vector Load Instructions (continued)

Name	Mnemonic	Syntax	Operation
Load Vector Indexed by External PID <E.PD>	**lvepx**	**v**D,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Load the quad word in memory addressed by EA & (~0xF) into **v**D.
Load Vector Indexed by External PID LRU <E.PD>	**lvepxl**	**v**D,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Load the quad word in memory addressed by EA & (~0xF) into **v**D. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.

The **lvsl** and **lvsr** instructions can be used to create the permute control vector to be used by a subsequent **vperm** instruction. Let X and Y be the contents of **v**A and **v**B specified by **vperm**. The control vector created by **lvsl** causes the **vperm** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes (sh = the value in EA[60–63]). The control vector created by **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. The sh values for the **lvsl** instruction are shown in Table 4-17, and those for the **lvsr** instruction are shown in Table 4-18. For rotating, the vector register to be rotated should be specified as both the **v**A and the **v**B register for **vperm**. For shifting left, the **v**B register for **vperm** should be a register containing all zeros and **v**A should contain the value to be shifted, and vice versa for shifting right.

Table 4-16 summarizes the vector alignment instructions.

Table 4-16. Vector Load Instructions Supporting Alignment

Name	Mnemonic	Syntax	Operation
Load Vector for Shift Left	**lvsl**	**v**D,**r**A, **r**B	The EA is the sum (**r**A|0) + (**r**B). The EA[60–63] = sh, then based on Table 4-17, place the value in **v**D
Load Vector for Shift Right	**lvsr**	**v**D,**r**A, **r**B	The EA is the sum (**r**A|0) + (**r**B). The EA[60–63] = sh, then based on Table 4-18, place the value in **v**D

Table 4-17. Shift Values for lvsl Instruction

Shift (sh)	**v**D**[0–127]**
0x0	0x000102030405060708090A0B0C0D0E0F
0x1	0x0102030405060708090A0B0C0D0E0F10
0x2	0x02030405060708090A0B0C0D0E0F1011
0x3	0x030405060708090A0B0C0D0E0F101112
0x4	0x0405060708090A0B0C0D0E0F10111213
0x5	0x05060708090A0B0C0D0E0F1011121314
0x6	0x060708090A0B0C0D0E0F101112131415
0x7	0x0708090A0B0C0D0E0F10111213141516
0x8	0x08090A0B0C0D0E0F1011121314151617
0x9	0x090A0B0C0D0E0F101112131415161718

Table 4-17. Shift Values for lvsl Instruction (continued)

Shift (sh)	vD[0–127]
0xA	0x0A0B0C0D0E0F10111213141516171819
0xB	0x0B0C0D0E0F101112131415161718191A
0xC	0x0C0D0E0F101112131415161718191A1B
0xD	0x0D0E0F101112131415161718191A1B1C
0xE	0x0E0F101112131415161718191A1B1C1D
0xF	0x0F101112131415161718191A1B1C1D1E

Table 4-18. Shift Values for lvsr Instruction

Shift (sh)	vD[0–127]
0x0	0x101112131415161718191A1B1C1D1E1F
0x1	0x0F101112131415161718191A1B1C1D1E
0x2	0x0E0F101112131415161718191A1B1C1D
0x3	0x0D0E0F101112131415161718191A1B1C
0x4	0x0C0D0E0F101112131415161718191A1B
0x5	0x0B0C0D0E0F101112131415161718191A
0x6	0x0A0B0C0D0E0F10111213141516171819
0x7	0x090A0B0C0D0E0F101112131415161718
0x8	0x08090A0B0C0D0E0F1011121314151617
0x9	0x0708090A0B0C0D0E0F10111213141516
0xA	0x060708090A0B0C0D0E0F101112131415
0xB	0x05060708090A0B0C0D0E0F1011121314
0xC	0x0405060708090A0B0C0D0E0F10111213
0xD	0x030405060708090A0B0C0D0E0F101112
0xE	0x02030405060708090A0B0C0D0E0F1011
0xF	0x0102030405060708090A0B0C0D0E0F10

4.2.3.4 Vector Load Instructions for Misaligned Vector Handling

For vector load instructions that help with misaligned accesses there are three addresses associated with each EA that may be used with these instructions.

1. The first address, EA, is the byte address specified from (**r**A|0) + (**r**B). This is called the pivot point.
2. The second address is the vector aligned address obtained by zeroing the low-order 4 bits of EA. This is called the vector start.
3. The third address is the address of the last byte in the 16 byte vector aligned address (vector start + 15). This is called the vector end.

These vector instructions load bytes from the pivot point to the vector end and left justify them in the vector register (**v**D) and load bytes from the vector start up to, but not including, the pivot point and right justify them in the vector register (**v**D). See Section 4.2.3.7, “Using Left, Right, and Swap Load and Store

Instructions for Misaligned Vector Access" for information on how to use these instructions to access misaligned vectors in memory.

Element load instructions load a byte, half-word, or word from memory at EA and places it in the element in vD specified by the low-order bits of rB. Note that rB is used to both form the EA and provide the element to load. This allows an arbitrary element to be loaded from any element aligned address.

Table 4-19 summarizes the vector load instructions for misaligned vector handling.

Table 4-19. Vector Load (for misaligned vectors) Instructions

Name	Mnemonic	Syntax	Operation
Load Vector Element Indexed [Byte, Half-word, Word] Indexed	**lvexbx** **lvexhx** **lvexwx**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B) truncated to the natural alignment boundary based on the element size (byte, half-word, or word). The element index is **r**B[60–63], **r**B[60–62], or **r**B[60–61] based on the element size (byte, half-word, or word). Load the byte, half-word, or word in memory addressed by EA into the specified element of **v**D. The remaining bits in **v**D are set to 0. The element of **v**D is determined by low-order bits of EA based on the element size: • For **b**, byte, 8 bits = 1 byte, element index in vector is EA[60–63] • For **h**, half-word, 16 bits = 2 bytes, element index in vector is EA[60–62] • For **w**, word, 32 bits = 4 bytes, element index in vector is EA[60–61]
Load Vector to Left Indexed	**lvtlx**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load bytes in memory starting at EA up until vector end, left justifying them into **v**D.
Load Vector to Left Indexed LRU	**lvtlxl**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load bytes in memory starting at EA up until vector end, left justifying them into **v**D. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.
Load Vector to Right Indexed	**lvtrx**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load bytes in memory starting at vector start up until, but not including, EA, right justifying them into **v**D.
Load Vector to Right Indexed LRU	**lvtrxl**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load bytes in memory starting at vector start up until, but not including, EA, right justifying them into **v**D. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.
Load Vector with Left-Right Swap Indexed	**lvswx**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load bytes in memory starting at EA up until vector end, left justifying them into **v**D. Load bytes in memory starting at vector start up to EA-1, right justifying them into **v**D.
Load Vector with Left-Right Swap Indexed LRU	**lvswxl**	vD,rA, rB	EA is the sum (**r**A|0) + (**r**B). Load bytes in memory starting at EA up until vector end, left justifying them into **v**D. Load bytes in memory starting at vector start up to EA-1, right justifying them into **v**D. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.

The **lvsm** instruction can be used to create the control vector to be used by a subsequent **vsel** instruction to combine the left and right sides of a vector loaded with a **lvswx** instruction. lvsm creates the control vector based on the EA supplied such that the bits on the left of the pivot point are all 0 and the bits on the right are all 1. The 0 filled bytes and the 1 filled bytes are then swapped such that the bytes with all 1s are in the leftmost bytes of the vector register. The swapping can be viewed as a rotate operation where the bytes of the vector register are all rotated EA & 0xf bytes to the left. The **lvsm** operation is the same as doing a **lvtlx** where the vector in memory contains all bits set. Thus, when used as the control vector for the **vsel** instruction, load vectors from 2 iterations of the loop from **lvswx** instructions are selected to form the complete vector. See Section 4.2.3.7, "Using Left, Right, and Swap Load and Store Instructions for Misaligned Vector Access" for information on how to use **lvsm** to access misaligned vectors in memory.

Table 4-20 summarizes the vector alignment instructions.

Table 4-20. lvsm Instructions Supporting Alignment

Name	Mnemonic	Syntax	Operation
Load Vector for Swap Merge	**lvsm**	vD,**r**A, **r**B	The EA is the sum (**r**A|0) + (**r**B). Bits to the left of the pivot point (EA) in **v**D are set to 0 and bits to the right of the pivot point (EA) in **v**D are set to 1. vD is then rotated left by (EA & 0xf) bytes.

4.2.3.5 Vector Store Instructions

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

Table 4-21 provides a summary of the vector store instructions.

Table 4-21. Vector Store Instructions

Name	Mnemonic	Syntax	Operation
Store Vector Element Integer Indexed [b,h,w]	**stvebx** **stvehx** **stvewx**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B) truncated to the natural alignment boundary based on the element size (byte, half-word, or word). Store the corresponding element of **v**D into the byte, half-word, or word in memory addressed by EA. The element of vD is determined by low-order bits of EA based on the element size: • For **b**, byte, 8 bits = 1 byte, element index in vector is EA[60–63] • For **h**, half-word, 16 bits = 2 bytes, element index in vector is EA[60–62] • For **w**, word, 32 bits = 4 bytes, element index in vector is EA[60–61]
Store Vector Indexed	**stvx**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the contents of **v**S into the quad word in memory addressed by EA & (~0xF).
Store Vector Indexed LRU	**stvxl**	vD,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the contents of **v**S into the quad word in memory addressed by EA & (~0xF). LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.
Store Vector Indexed by External PID <E.PD>	**stvepx**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the contents of **v**S into the quad word in memory addressed by EA & (~0xF).
Store Vector Indexed LRU <E.PD>	**stvepxl**	vD,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the contents of **v**S into the quad word in memory addressed by EA & (~0xF). LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.

4.2.3.6 Vector Store Instructions for Misaligned Vector Handling

For vector store instructions that help with misaligned accesses there are three addresses associated with each EA that may be used with these instructions.

1. The first address, EA, is the byte address specified from (**r**A|0) + (**r**B). This is called the pivot point.

2. The second address is the vector aligned address obtained by zeroing the low-order 4 bits of EA. This is called the vector start.
3. The third address is the address of the last byte in the 16 byte vector aligned address (vector start + 15). This is called the vector end.

These vector instructions store bytes starting from the left in **v**S to the pivot point up to the vector end and store bytes starting from the right in **v**S to the vector start up to, but not including, the pivot point.

Element store instructions store a byte, half-word, or word to memory at EA from the element in **v**S specified by the low-order bits of **r**B. Note that **r**B is used to both form the EA and provide the element to store. This allows an arbitrary element to be stored to any element aligned address. These element store instructions are useful for handling the tail end of a vector in memory when the remaining bytes are less than 16 bytes.

Table 4-22 summarizes the vector store instructions for misaligned vector handling.

Table 4-22. Vector Store (for misaligned vectors) Instructions

Name	Mnemonic	Syntax	Operation
Store Vector Element Indexed [Byte, Half-word, Word] Indexed	**stvexbx** **stvexhx** **stvexwx**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B) truncated to the natural alignment boundary based on the element size (byte, half-word, or word). The element index is **r**B[60–63], **r**B[60–62], or **r**B[60–61] based on the element size (byte, half-word, or word). Store the byte, half-word, or word to memory addressed by EA from the specified element of **v**S. The element of **v**S is determined by low-order bits of EA based on the element size: • For **b**, byte, 8 bits = 1 byte, element index in vector is EA[60–63] • For **h**, half-word, 16 bits = 2 bytes, element index in vector is EA[60–62] • For **w**, word, 32 bits = 4 bytes, element index in vector is EA[60–61]
Store Vector from Left Indexed	**stvflx**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the leftmost bytes from vS into memory starting at EA up until vector end.
Store Vector from Left Indexed LRU	**stvflxl**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the leftmost bytes from vS into memory starting at EA up until vector end. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.
Store Vector from Right Indexed	**stvfrx**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the rightmost bytes from **v**S into memory starting at vector start up until, but not including, EA.
Store Vector from Right Indexed LRU	**stvfrxl**	vS,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the rightmost bytes from **v**S into memory starting at vector start up until, but not including, EA. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.

Table 4-22. Vector Store (for misaligned vectors) Instructions (continued)

Name	Mnemonic	Syntax	Operation
Store Vector with Left-Right Swap Indexed	**stvswx**	**v**S,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the leftmost bytes from **v**S into memory starting at EA up until vector end. Store the rightmost bytes from **v**S into memory starting at vector start up until EA-1.
Store Vector with Left-Right Swap Indexed LRU	**stvswxl**	**v**S,**r**A, **r**B	EA is the sum (**r**A|0) + (**r**B). Store the leftmost bytes from **v**S into memory starting at EA up until vector end. Store the rightmost bytes from **v**S into memory starting at vector start up until EA-1. LRU = 1, least recently used, hints that the quad word in the memory addressed by EA will probably not be needed again by the program in the near future.

4.2.3.7 Using Left, Right, and Swap Load and Store Instructions for Misaligned Vector Access

The **lvtlx**, **lvswx**, **lvtrx**, and **lvsm** instructions can be used to load an arbitrarily aligned vector from memory to a vector register such that each 16 sequential bytes of memory are correctly positioned for processing. Similarly the **stvflx**, **stvswx**, **stvfrx**, and **lvsm** instructions can be used to store to an arbitrarily aligned location in memory from the data in a vector register. The alignment of the memory vector being loaded and the alignment of the memory vector being stored can be different. This paradigm of loading and storing vectors from memory does not require the **vperm** instruction and accesses each vector register sized quantum of memory only once. Using **vperm** to handle misalignment of arbitrary vectors may be unsuitable in future versions of AltiVec architecture if the width of the vector registers increases.

To illustrate how the arbitrary alignment load/store paradigm works, consider the following example of adding two memory vectors, A and B, with halfword size elements and storing the result in a third memory vector, C. For the purposes of this example assume the following:

- The data elements of the memory vectors are halfwords and the memory vector is halfword-aligned (the example will also work for other data sizes arbitrarily aligned on their natural boundaries - i.e. words or bytes).
- The length of the memory vectors are each a multiple of 16 bytes (that is each 16 bytes will be processed by an iteration of the loop) and there will be three or more (48 + N*16 bytes where N≥0).
- r6 contains the length of the vector in bytes.
- The starting address of memory vector A is in r3, the starting address of memory vector B is in r4, and the starting address of memory vector C is in r5. Further assume that the address of memory vector A is 0x10006, B is 0c2000c, and C is 0x30002.
- The data in memory vectors A and B contain halfword values starting at 0x0102 and incrementing by 0x0202 for each element (to illustrate the contents of the registers through the example).

```
// create the control vectors for the A (v1), B (v2)and C (v3) memory arrays:
    lvsm    v1,0,r3     // create control vector for selecting A
    lvsm    v2,0,r4     // create control vector for selecting B
    lvsm    v3,0,r5     // create control vector for selecting C

    li      r7,16       // running offset the size of a VR to reduce number of addi

// The pivot point for A is 0x6, B is 0xc, and C is 0x2
//
//  v1 = 0x ff ff ff ff ff ff ff ff ff ff 00 00 00 00 00 00
// v2 = 0x ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00
// v3 = 0x ff ff ff ff ff ff ff ff ff ff ff ff ff ff 00 00

// load first set of bytes into the left side of a vector register

   lvtlx   v4,0,r3 // get first 10 bytes of A into left side of v4
// v4 = 0x 01 02 03 04 05 06 07 08 09 0a 00 00 00 00 00 00

   lvtlx   v6,0,r4 // get first 4 bytes of B into left side of v6
// v6 = 0x 01 02 03 04 00 00 00 00 00 00 00 00 00 00 00 00

   lvswx   v5,r7,r3    // get the next 16 bytes of A swapped around the pivot point
// v5 = 0x 11 12 13 14 15 16 17 18 19 1a 0b 0c 0d 0e 0f 10

   vsel    v8,v5,v4,v1 // form the vector from the last left side and current right side
// v8 = 0x 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10

   lvswx   v7,r7,r4    // get the next 16 bytes of B swapped around the pivot point
// v7 = 0x 11 12 13 14 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10

   vsel    v9,v7,v6,v2 // form the vector from the last left side and current right side
// v9 = 0x 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10

// perform operation (add halfwords)
   vadduhm v11,v8,v9
// v11 = 0x 02 04 06 08 0a 0c 0e 10 12 14 16 18 1a 1c 1e 20

   stvflx  v11,0,r5// store start of vector C up to the next alignment boundary
// 0x30000 = 0x XX XX 02 04 06 08 0a 0c 0e 10 12 14 16 18 1a 1c

   srawi   r8,r6,4 // convert length in bytes to length in vector chunks
   addi    r8,r8,-2// subtract for the head and tail operations
   mtctr   r8      // process remaining 1+ in inner loop
// body of loop begins

loop:
   vor     v4,v5,v5    // move most recent loaded vector
   vor     v6,v7,v7    // move most recent loaded vector
   vor     v10,v11,v11 // move most recent store vector

   addi    r5,r5,16    // increment to next part of vector C
   addi    r7,r7,16    // increment to next part of vectors A and B

   lvswx   v5,r7,r3    // get the next 16 bytes of A swapped around the pivot point
// v5 = 0x 21 22 23 24 25 26 27 28 29 2a 1b 1c 1d 1e 1f 20 (on first iteration)

   vsel    v8,v5,v4,v1 // form the vector from the last left side and current right side
// v8 = 0x 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 (on first iteration)

   lvswx   v7,r7,r4    // get the next 16 bytes of B swapped around the pivot point
// v7 = 0x 21 22 23 24 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 (on first iteration)

   vsel    v9,v7,v6,v2 // form the vector from the last left side and current right side
// v9 = 0x 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 (on first iteration)

// perform operation (add halfwords)
   vadduhm v11,v8,v9
// v11 = 0x 22 24 26 28 2a 2c 2e 30 32 34 36 38 3a 3c 3e 40 (on first iteration)

   vsel    v8,v10,v11,v3                        // form vector for storing to C
// v8 = 0x 22 24 26 28 2a 2c 2e 30 32 34 36 38 3a 3c 1e 20 (on first iteration)

   stvswx  v8,0,r5 // store 16 bytes of C
```

```
// 0x30010 = 0x 1e 20 22 24 26 28 2a 2c 2e 30 32 34 36 38 3a 3c (on first iteration)

    bdnz    loop

// do tail end processing. Since we have assumed that the memory vectors are a
// multiple of 16 bytes, we can just do left/right instructions

    vor     v4,v5,v5    // move most recent loaded vector
    vor     v6,v7,v7    // move most recent loaded vector
    vor     v10,v11,v11 // move most recent store vector

    addi    r5,r5,16    // increment to next part of vector C
    addi    r7,r7,16    // increment to next part of vectors A and B

    lvtrx   v5,r7,r3
//  v5 = 0x 00 00 00 00 00 00 00 00 00 00 2b 2c 2d 2e 2f 30

    lvtrx   v7,r7,r4
//  v7 = 0x 00 00 00 00 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30

    vsel    v8,v5,v4,v1
//  v8 = 0x 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30

    vsel    v9,v7,v6,v2
//  v9 = 0x 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30

// perform operation (add halfwords)
    vadduhm v11,v8,v9
//  v11 = 0x 42 44 46 48 4a 4c 4e 50 52 54 56 58 5a 5c 5e 60

    vsel    v8,v10,v11,v3// form vector for storing to C
//  v8 = 0x 42 44 46 48 4a 4c 4e 50 52 54 56 58 5a 5c 3e 40

    stvswx  v8,0,r5 // store 16 bytes of C
//  0x30020 = 0x 3e 40 42 44 46 48 4a 4c 4e 50 52 54 56 58 5a 5c

    addi    r5,r5,16
    stvfrx  v11,0,r5// store last bytes from right side of v11
//  0x30030 = 0x 5e 60 XX XX XX XX XX XX XX XX XX XX XX XX XX XX
```

The example shows that **lvsm** is used to create a control vector for each of the input and output memory vectors that is later used with **vsel** to select the appropriate parts of vector sized chunks which are loaded from memory. Fundamentally, each input or output streams requires 3 vector registers, one for a control vector (from **lvsm**), and two to hold loaded (or computed data ready for storing) data from memory. The two vector registers hold data from 2 consecutive iterations of the loop.

This method only accesses each vector register sized memory element once, reducing load/store bandwidth.

If the size of the memory vectors is not a multiple of vector register length, the tail end of the processing can be modified to change the pivot point for the left and right loads and stores, use **vperm**, or use element loads and stores (**lvex***n***x** and **stvex***n***x**) instructions.

4.2.4 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.

4.2.4.1 Vector Pack Instructions

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

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

Table 4-23 describes the vector pack instructions.

Table 4-23. Vector Pack Instructions

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

Table 4-23. Vector Pack Instructions (continued)

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

4.2.4.2 Vector Unpack Instructions

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

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

Table 4-24 describes the unpack instructions.

Table 4-24. Vector Unpack Instructions

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

Table 4-24. Vector Unpack Instructions (continued)

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

4.2.4.3 Vector Merge Instructions

Byte vector merge instructions interleave the eight low bytes (or eight high bytes) from two source operands producing a result of 16 bytes. Similarly, half-word vector merge instructions interleave the four low half-words (or four high half-words) of two source operands producing a result of eight half-words, and word vector merge instructions interleave the two low words (or two 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 4-25 describes the merge instructions.

Table 4-25. Vector Merge Instructions

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

4.2.4.4 Vector Splat Instructions

When a program needs to perform vector arithmetic, 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 must be 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 4-26 describes the vector splat instructions.

Table 4-26. Vector Splat Instructions

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

4.2.4.5 Vector Permute Instruction

Permute instructions allow any byte in any two source vector registers to be directed to any byte in the destination vector. The fields in a third source operand specify from which field in the source operands the corresponding destination field will be taken. The Vector Permute (**vperm**) instruction is a very powerful one that provides many useful functions. For example, it provides a good way to perform table-lookups and data alignment operations. An example of how to use the command in aligning data is shown in Section 3.1.5, "Quad-Word Data Alignment."

Table 4-27 describes the vector permute instruction.

Table 4-27. Vector Permute Instruction

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

4.2.4.6 Vector Select Instruction

Data flow 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 4-28 describes the **vsel** instruction.

Table 4-28. Vector Select Instruction

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

4.2.4.7 Vector Shift Instructions

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

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

Table 4-29 describes the various vector shift instructions.

Table 4-29. Vector Shift Instructions

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

4.2.4.7.1 Immediate inter-element Shifts/Rotates

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

Table 4-30 below enumerates how various shift functions can be achieved using the **vsidoi** instruction.

Table 4-30. 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

Table 4-30. Coding Various Shifts and Rotates with the vsidoi Instruction (continued)

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

4.2.4.7.2 Computed inter-element Shifts/Rotates

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

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

For an example on how the **lvsl** instruction is used to align a vector in big-endian mode, see Section 3.1.5.1, "Accessing a Misaligned Quad Word in Big-Endian Mode."

4.2.4.7.3 Variable inter-element Shifts

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

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

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

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

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

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

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

4.2.4.8 AltiVec Status and Control Register Instructions

Table 4-31 summarizes the instructions for reading from or writing to the Vector Status and Control Register (VSCR). For more information on VSCR see Section 2.2.2, "Vector Status and Control Register (VSCR)."

Table 4-31. Move To/From Vector Status and Control Register Instructions

Name	Mnemonic	Syntax	Operation
Move to Vector Status and Control Register	**mtvscr**	**v**B	Place the contents of **v**B into VSCR.
Move from Vector Status and Control Register	**mfvscr**	**v**D	Place the contents of VSCR into **v**D.

4.2.4.9 GPR to AltiVec Move Instructions

Table 4-32 summarizes the instructions for moving data from the GPRs to a vector register.

Table 4-32. Move To Vector Register from GPR Instructions

Name	Mnemonic	Syntax	Operation
Move to Vector from Integer Double Word and Splat <64>	**mvidsplt**	**v**D,**r**A,**r**B	Place the contents of **r**A into high-order 64 bits of **v**D and place the contents of **r**B into the low-order 64 bits of **v**D.
Move to Vector from Integer Word and Splat	**mviwsplt**	**v**D,**r**A,**r**B	Place the contents of the low-order 32 bits of **r**A concatenated with low-order 32 bits of **r**B into the low-order and high-order 64 bits of **v**D.

Chapter 5
Cache, Interrupts, and Memory Management

This chapter summarizes details of AltiVec technology that pertain to cache and memory management models. Note that AltiVec technology defines most of its instructions at the user level. Because most AltiVec instructions are computational, there is little effect on the supervisor portions of Power ISA™.

Because AltiVec uses 128-bit operands, additional instructions are provided to optimize cache and memory bus use.

5.1 Prioritizing Cache Block Replacement

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

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

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

5.2 Partially Executed AltiVec Instructions

Power ISA and EIS permit certain instructions to be partially executed when an alignment or data storage interrupt occurs. In the same way that the target register may be altered when floating-point load instructions cause a data storage interrupt, if the AltiVec facility is implemented, the target register (**v**D) may be altered when **lvx**, **lvxl**, **lvepx** <E.PD> , or **lvepxl** <E.PD> is executed and the TLB entry is invalidated before the access completes.

5.3 AltiVec Unavailable Interrupt

The AltiVec facility includes an instruction-caused, precise synchronous interrupt to notify the operating system when an AltiVec instruction attempts execution. This allows the operating system to only save and restore vector registers for processes that are using them. The interrupt is more fully described in Chapter

7, “Interrupts and Exceptions,” in the *EREF*. An AltiVec unavailable interrupt occurs when no higher priority interrupt exists, an attempt is made to execute an AltiVec instruction, and MSR[SPV] = 0.

5.4 AltiVec Assist Interrupt

The AltiVec facility also includes an AltiVec assist interrupt. The AltiVec assist interrupt occurs when a denormalized number is an operand to an AltiVec floating-point instruction and the processor requires that software perform the operation by emulating the instruction to provide correct results with the denormalized input. If an implementation can perform the operation correctly using the denormalized operand, no AltiVec assist interrupt will occur. The interrupt is more fully described in Chapter 7, “Interrupts and Exceptions,” in the *EREF*.

Chapter 6
AltiVec Instructions

This chapter lists the AltiVec instruction set in alphabetical order by mnemonic. Note that each entry includes the instruction format and a graphical representation of the instruction. All the instructions are 32 bits and a description of the instruction fields and pseudocode conventions are also provided. For more information on the AltiVec instruction set, refer to Chapter 4, "Addressing Modes and Instruction Set Summary." For more information on the Power ISA™ instruction set, refer to Chapter 5, "Instruction Set," in the *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture Processors*.

6.1 Instruction Formats

AltiVec instructions are 4 bytes (32 bits) long and are word-aligned. AltiVec instruction set architecture (ISA) has four operands, three source vectors, and one result vector. Bits 0–5 always specify the primary opcode for AltiVec instructions. AltiVec ALU-type instructions specify the primary opcode 04 (0b00_01_00). AltiVec load and store use secondary opcodes in primary opcode 31 (0b01_11_11).

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

- Byte elements, each byte = 8 bits; in the pseudocode, n = 8 with a total of 16 elements
- Half-word elements, each byte = 16 bits; in the pseudocode, n = 16 with a total of 8 elements
- Word elements, each byte = 32 bits; in the pseudocode, n = 32 with 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

Field	Description
OPCD (0–5)	Primary opcode field
rA, A (11–15)	Specifies a GPR to be used as a source or destination
rB, B (16–20)	Specifies a GPR to be used as a source
Rc (31)	Record bit 0 Does not update the condition register (CR). 1 For the AltiVec facility, set CR field 6 to control program flow.
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

Table 6-1. Instruction Syntax Conventions (continued)

Field	Description
vD (6–10)	Specifies a vector register to be used as a destination
vS (6–10)	Specifies a vector register to be used as a source
SHB (22–25)	Specifies a shift amount in bytes
SIMM (11–15)	This immediate field is used to specify a (5-bit) signed integer
UIMM (11–15)	This immediate field is used to specify a 4-, 8-, 12-, or 16-bit unsigned integer

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 chapter.

Table 6-2. Notation and Conventions

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

Table 6-2. Notation and Conventions (continued)

Notation/Convention	Meaning
$<_{fp}$, $\leq_{fp}$, $>_{fp}$, $\geq_{fp}$	Single precision floating-point comparison relations
$\neq$	Not equal
$=_{int}$	Integer equal to
$=_{ui}$	Unsigned integer equal to
$=_{si}$	Signed integer equal to
$=_{fp}$	Floating-point equal to
$X >>_{ui} Y$	Shift X right by Y bits extending Xs vacated bits with zeros
$X >>_{si} Y$	Shift X right by Y bits extending Xs vacated bits with the sign bit of X
$X <<_{ui} Y$	Shift X left by Y bits inserting Xs vacated bits with zeros
||	Used to describe the concatenation of two values (that is, 010 || 111 is the same as 010111)
&	AND logical operator
|	OR logical operator
$\oplus$, $\equiv$	Exclusive-OR, equivalence logical operators (for example, $(a \equiv b) = (a \oplus \neg b)$)
0b*nnnn*	A number expressed in binary format
0x*nnnn*	A number expressed in hexadecimal format
?	Unordered comparison relation
${}^{x}0$	X zeros
${}^{x}1$	X ones
${}^{x}Y$	X copies of Y
X_Y	Bit Y of X
$X_{Y:Z}$	Bits Y through Z, inclusive, of X
LENGTH(x)	Length of x, in bits. If x is the word 'element,' LENGTH(x) is the length, in bits, of the element implied by the instruction mnemonic.
ROTL(x,y)	Result of rotating x left by y bits
ROTR(x,y)	Result of rotating x right by y bits
UItoUImod(X,Y)	Chop unsigned integer X- to Y-bit unsigned integer
UItoUIsat(X,Y)	Result of converting the unsigned-integer x to a y-bit unsigned-integer with unsigned-integer saturation
SItoUIsat(X,Y)	Result of converting the signed-integer x to a y-bit unsigned-integer with unsigned-integer saturation
SItoSImod(X,Y)	Chop integer X- to Y-bit integer
SItoSIsat(X,Y)	Result of converting the signed-integer x to a y-bit signed-integer with signed-integer saturation

Table 6-2. Notation and Conventions (continued)

Notation/Convention	Meaning
RndToNearFP32	The single-precision floating-point number that is nearest in value to the infinitely-precise floating-point intermediate result x (in case of a tie, the even single-precision floating-point value is used).
RndToFPInt32Near	The value x if x is a single-precision floating-point integer; otherwise the single-precision floating-point integer that is nearest in value to x (in case of a tie, the even single-precision floating-point integer is used).
RndToFPInt32Trunc	The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x if $x > 0$, or the smallest single-precision floating-point integer that is greater than x if $x < 0$.
RndToFPInt32Ceil	The value x if x is a single-precision floating-point integer; otherwise the smallest single-precision floating-point integer that is greater than x.
RndToFPInt32Floor	The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x.
CnvtFP32ToUI32Sat(x)	Result of converting the single-precision floating-point value x to a 32-bit unsigned-integer with unsigned-integer saturation
CnvtFP32ToSI32Sat(x)	Result of converting the single-precision floating-point value x to a 32-bit signed-integer with signed-integer saturation
CnvtUI32ToFP32(x)	Result of converting the 32-bit unsigned-integer x to floating-point single format
CnvtSI32ToFP32(x)	Result of converting the 32-bit signed-integer x to floating-point single format
MEM(X,Y)	Value at memory location X of size Y bytes
SwapDouble	Swap the doublewords in a quad-word vector
ZeroExtend(X,Y)	Zero-extend X on the left with zeros to produce Y-bit value
SignExtend(X,Y)	Sign-extend X on the left with sign bits (that is, with copies of bit 0 of x) to produce Y-bit value
RotateLeft(X,Y)	Rotate X left by Y bits
mod(X,Y)	Remainder of X/Y
UImaximum(X,Y)	Maximum of 2 unsigned integer values, X and Y
SImaximum(X,Y)	Maximum of 2 unsigned integer values, X and Y
FPmaximum(X,Y)	Maximum of 2 floating-point values, X and Y
UIminimum(X,Y)	Minimum of 2 unsigned integer values, X and Y
SIminimum(X,Y)	Minimum of 2 unsigned integer values, X and Y
FPminimum(X,Y)	Minimum of 2 floating-point values, X and Y
FPReciprocalEstimate12(X)	12-bit-accurate floating-point estimate of 1/X
FPReciprocalSQRTEstimate12(X)	12-bit-accurate floating-point estimate of 1/(sqrt(X))
FPLog$_2$Estimate3(X)	3-bit-accurate floating-point estimate of log2(X)
FPPower2Estimate3(X)	3-bit-accurate floating-point estimate of 2**X
CarryOut(X + Y)	Carry out of the sum of X and Y

Table 6-2. Notation and Conventions (continued)

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

Table 6-2. Notation and Conventions (continued)

Notation/Convention	Meaning
Insert	Select a field of *n* bits in the source register, insert this field starting at bit position *b* of the target register, and leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a field when operating on double words; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)
Leave	Leave innermost do loop, or the do loop described in leave statement
MASK(x, y)	Mask having ones in positions x through y (wrapping if x > y) and zeros elsewhere
MEM(x, y)	Contents of y bytes of memory starting at address x
NIA	Next instruction address, which is the 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.
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.

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

Table 6-3. Instruction Field Conventions

Power ISA Architecture Specification	Equivalent in AltiVec Technology PEM as:
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S
SI	SIMM
U	IMM
UI	UIMM
VA, VB, VC, VT, VS	**v**A, **v**B, **v**C, **v**D, **v**S
/, //, ///	0...0 (shaded)

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

Table 6-4. Precedence Rules

Operators	Associativity
x[*n*], function evaluation	Left to right
(*n*)x or replication, x(*n*) or exponentiation	Right to left
unary –, ¬	Right to left
∗, ÷	Left to right
+, –	Left to right
||	Left to right
=, ≠, <, ≤, >, ≥, <U, >U, ?	Left to right
&, ⊕, ≡	Left to right
|	Left to right
– (range), : (range)	None
$\leftarrow$, $\leftarrow_{\text{iea}}$	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 in the Associativity column. For example, '–' (unary minus) associates from left to right, so a – b – c = (a – b) – c. Parentheses are used to override the evaluation order implied by Table 6-4, or to increase clarity; parenthesized expressions are evaluated before serving as operands.

6.2 AltiVec Instruction Set

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

Instruction name

vaddsbs **vaddsbs**

Vector Add Signed Byte Saturate

Instruction syntax and form

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

Instruction encoding in decimal

0 5	6 10	11 15	16 20	21 31
04	**v**D	**v**A	**v**B	768

Pseudocode description of instruction operation

```
do i=0 to 127 by 8
        aop$_{0:8}$ ← SignExtend((vA)$_{i:i+7}$,9)
        bop$_{0:8}$ ← SignExtend((vB)$_{i:i+7}$,9)
        temp$_{0:8}$← aop$_{0:8}$ +$_{int}$ bop$_{0:8}$
        vD$_{i:i+7}$ ← SItoSIsat(temp$_{0:8}$,8)
end
```

Text description of instruction operation

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

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

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

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

Other registers altered:

Vector status and control register (VSCR):
Affected: SAT

Figure 6-11 shows the usage of the **vaddsbs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

Figure showing instruction usage

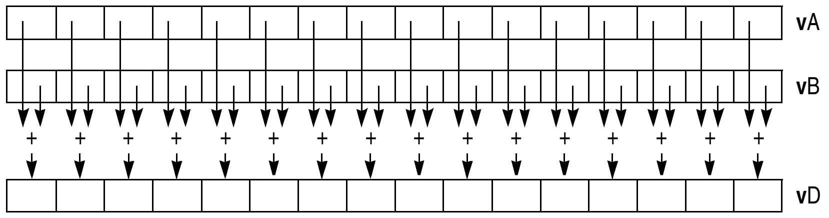


Figure 6-11. vaddsbs—Add Saturating Sixteen Signed Integer Elements (8-Bit)

dss dss

Data Stream Stop

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

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

```
no operation
```

For processors that comply with versions of Power ISA, this instruction is not defined and is treated as a no-op.

Software Note

dss, **dssall**, **dst**, **dstt**, **dstst**, and **dststt** instructions were present in the first definition of AltiVec for PowerPC® processors. These instructions provided software initiated streaming prefetch controls. In Power ISA these instructions are no longer defined and streaming is performed by variants of the **dcbt** instruction or by hardware prefetchers. For Freescale EIS, these instructions are treated as no-ops since they may be present in older code and do not change architectural state.

dst dst

Data Stream Touch

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

0 5	6	7 8	9 10	11 15	16 20	21 30	31
31	T	//	STRM	rA	rB	342	/

```
no operation
```

For processors that comply with versions of Power ISA, this instruction is not defined and is treated as a no-op.

Software Note

dss, **dssall**, **dst**, **dstt**, **dstst**, and **dststt** instructions were present in the first definition of AltiVec for PowerPC processors. These instructions provided software initiated streaming prefetch controls. In Power ISA these instructions are no longer defined and streaming is performed by variants of the **dcbt** instruction or by hardware prefetchers. For Freescale EIS, these instructions are treated as no-ops since they may be present in older code and do not change architectural state.

dstst dstst

Data Stream Touch for Store

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

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

```
no operation
```

For processors that comply with versions of Power ISA, this instruction is not defined and is treated as a no-op.

Software Note

dss, **dssall**, **dst**, **dstt**, **dstst**, and **dststt** instructions were present in the first definition of AltiVec for PowerPC processors. These instructions provided software initiated streaming prefetch controls. In Power ISA these instructions are no longer defined and streaming is performed by variants of the **dcbt** instruction or by hardware prefetchers. For Freescale EIS, these instructions are treated as no-ops since they may be present in older code and do not change architectural state.

lvebx lvebx

Load Vector Element Byte Indexed

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

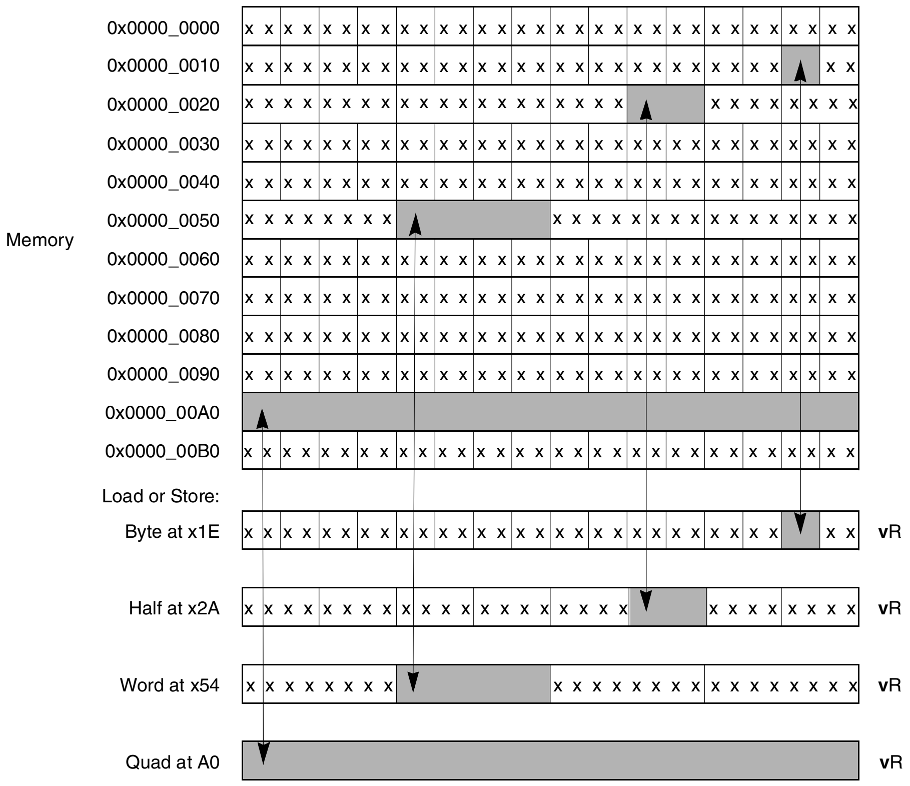
0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	7	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← $EA_{28:31}$
vD ← undefined
$vD_{eb*8:(eb*8)+7}$ ← MEM(EA,1)
```

Let m be the byte element of the vector indexed by EA[60–63]. The byte addressed by EA is loaded into the byte element indexed by m of **v**D. Remaining bytes in **v**D are undefined.

Other registers altered:

- None



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

Figure 6-1. Effects of Example Load/Store Instructions

lvehx lvehx

Load Vector Element Half Word Indexed

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

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

```
if rA=0 then    b ← 0
else            b ← (rA)
EA ← (b + (rB)) & (~1)
eb ← EA$_{60:63}$
vD ← undefined
vD$_{(eb*8):(eb*8)+15}$ ← MEM(EA,2)
```

Let m be the half word element of the vector indexed by EA[60–62]. The half word addressed by EA is loaded into half word element indexed by m of **v**D. The remaining half words in **v**D are set to undefined values. Figure 6-1 shows this instruction.

Other registers altered:

- None

lvepx

E.PD, V	Supervisor

lvepx

Load Vector by External PID Indexed

lvepx **v**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	vD	rA	rB	0 1 0 0 1 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
vD ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16)
```

The quadword addressed by EA truncated to the nearest quadword boundary is loaded into **v**D. The data is loaded in big-endian form regardless of the setting of the E storage attribute.

This instruction is guest supervisor privileged.

An attempt to execute **lvepx** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **lvepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lvepxl

E.PD, V	Supervisor

lvepxl

Load Vector by External PID Indexed LRU

lvepxl **v**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	vD	rA	rB	0 1 0 0 0 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
vD ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16)
```

The quadword addressed by EA truncated to the nearest quadword boundary is loaded into **v**D. The data is loaded in big-endian form regardless of the setting of the E storage attribute.

The **lvepxl** instruction provides a hint that the quad word addressed by EA will probably not be needed again by the program in the near future.

Note that on some implementations, the hint provided by the **lvepxl** instruction and the corresponding hint provided by the **stvepxl** instruction 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.

This instruction is guest supervisor privileged.

An attempt to execute **lvepxl** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **lvepxl**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lvewx lvewx

Load Vector Element Word Indexed

lvewx vD,**r**A,**r**B Form X

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

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~3)
eb ← $EA_{60:63}$
vD ← undefined
$vD_{eb*8:(eb*8)+31}$ ← MEM(EA,4)
```

Let m be the word element of the vector indexed by EA[60–61]. The word addressed by EA is loaded into word m of **v**D. The remaining words in **v**D are set to undefined values. Figure 6-1 shows this instruction.

Other registers altered:

- None

lvexbx lvexbx

Load Vector Element Indexed Byte Indexed

lvexbx vD,**r**A,**r**B Form X

0 … 5	6 … 10	11 … 15	16 … 20	21 … 30	31
31	vD	rA	rB	261	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
e ← (rB)$_{60:63}$
vD ← 0
vD$_{e*8:(e*8)+7}$ ← MEM(EA,1)
```

Let E be the byte element of vector register **v**D indexed by **r**B[60–63]. The byte addressed by EA is loaded into byte E of **v**D. The remaining bytes in **v**D are set to 0.

Other registers altered:

- None

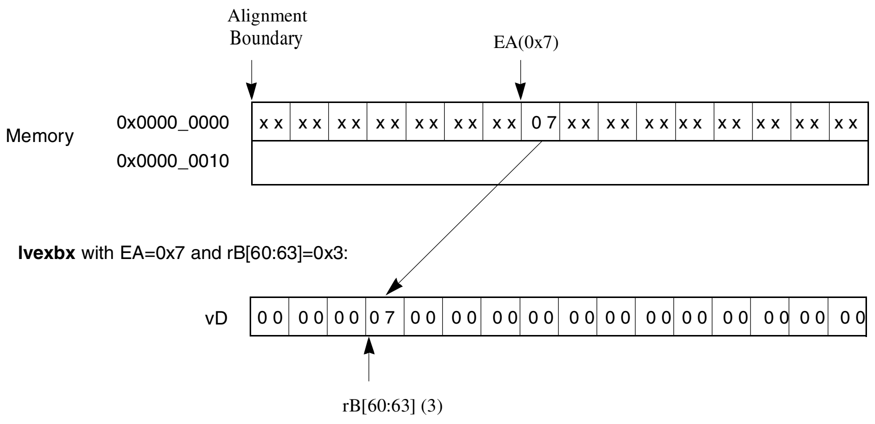


Figure 6-2. Example lvexbx Instruction

lvexhx lvexhx

Load Vector Element Indexed Half Word Indexed

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

0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	293	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & ~0x1
e ← (rB)60:62
vD ← 0
vDe*16:(e*16)+15 ← MEM(EA,2)
```

Let E be the half word element index of vector register **v**D indexed by **r**B[60–62]. The half word addressed by EA is loaded into half word E of **v**D. The remaining bytes in **v**D are set to 0.

Other registers altered:

- None

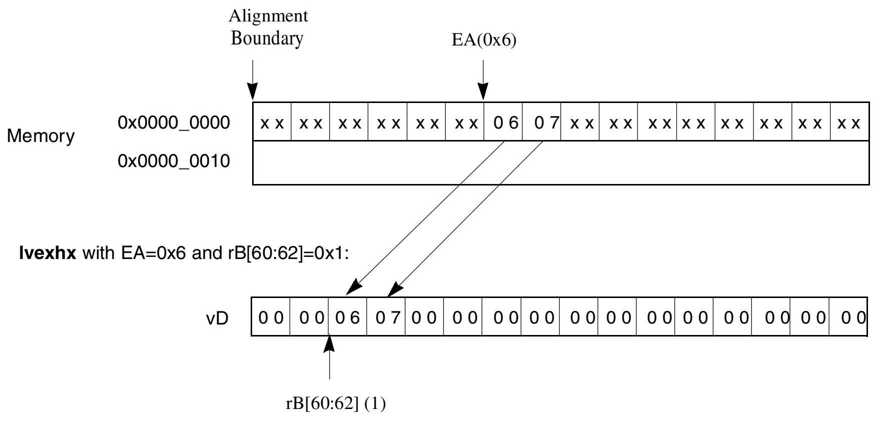


Figure 6-3. Example lvexhx Instruction

lvexwx lvexwx

Load Vector Element Indexed Word Indexed

lvexwx vD,**r**A,**r**B Form X

0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	325	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & ~0x3
e ← (rB)60:61
vD ← 0
vDe*32:(e*32)+31 ← MEM(EA,4)
```

Let E be the word element index of vector register **v**D indexed by **r**B[60–61]. The word addressed by EA is loaded into word E of **v**D. The remaining bytes in **v**D are set to 0.

Other registers altered:

- None

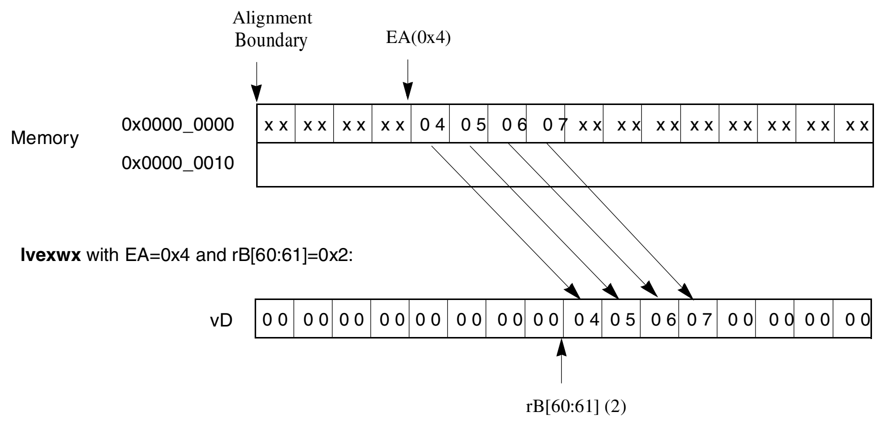


Figure 6-4. Example lvexwx Instruction

lvtlx lvtlx

Load Vector to Left Indexed

lvtlx vD,**r**A,**r**B (LRU=0) Form X

0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	581	/

```
if rA=0 then b ← 0
else          b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
vt0:127 ← MEM(sa,16)
vD ← vt <<ui (pivot * 8)
```

Bytes addressed by EA up to, but not including the next quad word alignment boundary are loaded from memory into the most significant bytes in **v**D. Other bytes in **v**D are set to 0.

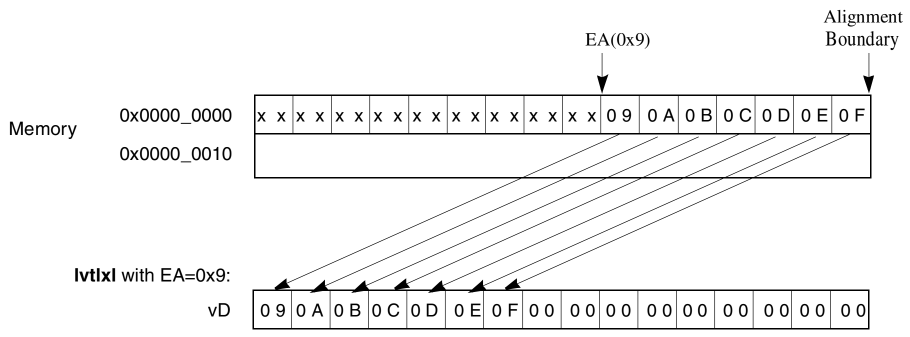
lvtlx performs a "load to left" operation, where the bytes starting at EA up to the next quad word alignment boundary are loaded from memory left justified into **v**D.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-5 shows an example of **lvtlx**.

Other registers altered:

- None



Alignment Boundary
EA(0x9)
Memory
0x0000_0000
0x0000_0010
x x x x x x x x x x x x x x x x x x 0 9 0 A 0 B 0 C 0 D 0 E 0 F
lvtlxl with EA=0x9:
vD
0 9 0 A 0 B 0 C 0 D 0 E 0 F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figure 6-5. Example lvtlx Instruction

lvtlxl lvtlxl

Load Vector to Left Indexed LRU

lvtlxl vD,**r**A,**r**B (LRU=1) Form X

0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	837	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
$vt_{0:127}$ ← MEM(sa,16)
vD ← vt $<<_{ui}$ (pivot * 8)
```

Bytes addressed by EA up to, but not including the next quad word alignment boundary are loaded from memory into the most significant bytes in **v**D. Other bytes in **v**D are set to 0.

lvtlxl performs a “load to left” operation, where the bytes starting at EA up to the next quad word alignment boundary are loaded from memory left justified into **v**D.

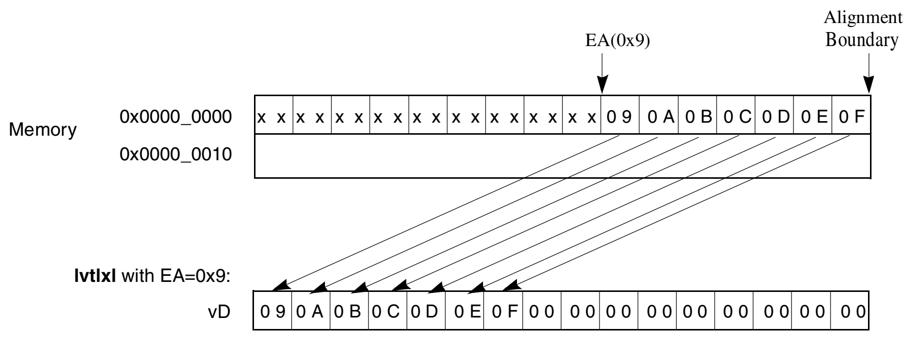
lvtlxl provides a hint that the program may not need the cache line addressed by EA again soon.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-6 shows an example of **lvtlxl**.

Other registers altered:

- None



Alignment Boundary
EA(0x9)
Memory
0x0000_0000
x x x x x x x x x x x x x x x x x x 0 9 0 A 0 B 0 C 0 D 0 E 0 F
0x0000_0010
lvtlxl with EA=0x9:
vD
0 9 0 A 0 B 0 C 0 D 0 E 0 F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figure 6-6. Example lvtlxl Instruction

lvtrx lvtrx

Load Vector to Right Indexed

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

0 5	6 10	11 15	16 20	21 30	31
31	vD	rA	rB	549	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
if pivot = 0 then
    vD ← 0
else
    $vt_{0:127}$ ← MEM(sa,16)
    vD ← vt $>>_{ui}$ ((16 - pivot) * 8)
```

Bytes addressed by EA truncated to the quad word alignment boundary up to, but not including, the byte addressed by EA are loaded from memory into the least significant bytes in **v**D. Other bytes in **v**D are set to 0.

lvtrx performs a “load to right” operation, where the bytes starting at EA truncated to a quad word alignment boundary up to EA are loaded from memory right justified into **v**D. EA points to the byte after the last byte to be loaded. If EA points to the first byte in a quad word, no memory access is performed but protection and debug events are still performed for address EA, and **v**D is set to 0.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-7 shows an example of **lvtrx**.

Other registers altered:

- None

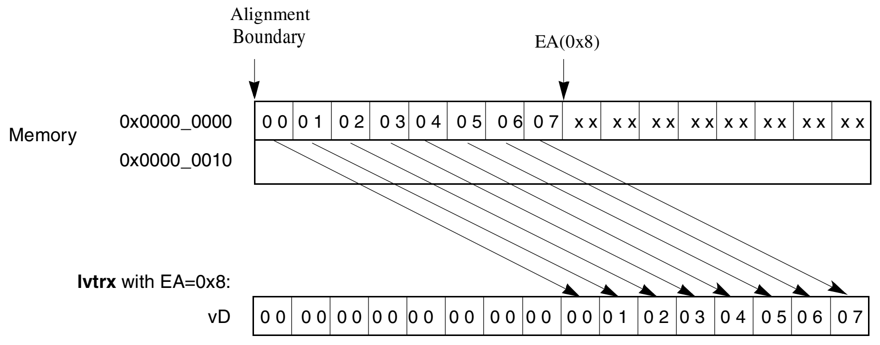


Figure 6-7. Example lvtrx Instruction

lvtrxl lvtrxl

Load Vector to Right Indexed LRU

lvtrxl vD,**r**A,**r**B (LRU=1) Form X

0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	805	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
if pivot = 0 then
    vD ← 0
else
    $vt_{0:127}$ ← MEM(sa,16)
    vD ← vt $>>_{ui}$ ((16 - pivot) * 8)
```

Bytes addressed by EA truncated to the quad word alignment boundary up to, but not including, the byte addressed by EA are loaded from memory into the least significant bytes in **v**D. Other bytes in **v**D are set to 0.

lvtrxl performs a "load to right" operation, where the bytes starting at EA truncated to a quad word alignment boundary up to EA are loaded from memory right justified into **v**D. EA points to the byte after the last byte to be loaded. If EA points to the first byte in a quad word, no memory access is performed but protection and debug events are still performed for address EA, and **v**D is set to 0.

lvtrxl provides a hint that the program may not need the cache line addressed by EA again soon.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-8 shows an example of **lvtrxl**.

Other registers altered:

- None

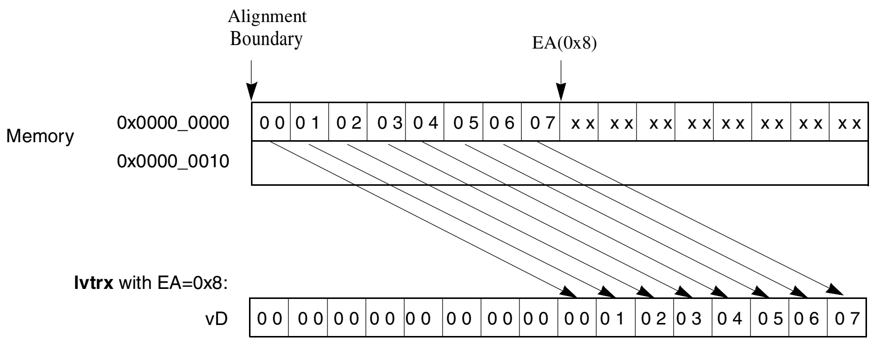


Figure 6-8. Example lvtrxl Instruction

lvsl lvsl

Load Vector for Shift Left

lvsl vD,**r**A,**r**B Form X

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

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

Let X be the 32-byte value 0x00 || 0x01 || 0x02 || ... || 0x1E || 0x1F. Bytes sh:sh+15 of X are placed into **v**D. Figure 6-9 shows how this instruction works.

Other registers altered:

- None

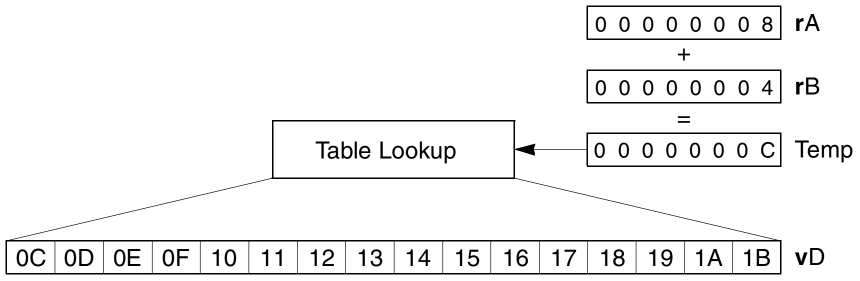


Figure 6-9. Load Vector for Shift Left

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

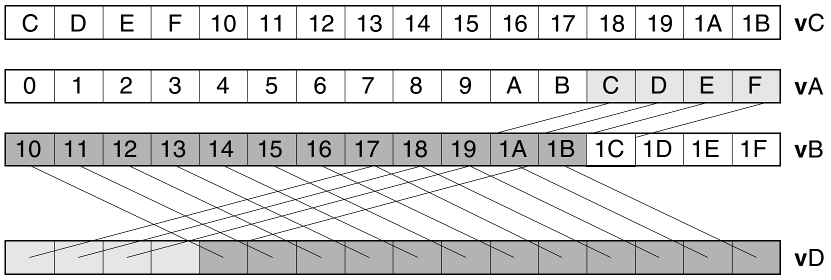


Figure 6-10. Instruction vperm Used in Aligning Data

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

lvsm lvsm

Load Vector for Swap Merge

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

0 5	6 10	11 15	16 20	21 30	31
31	vD	rA	rB	773	/

```
if rA = 0 then b ← 0
else           b ← (rA)
EA ← b + (rB)

$vt_{0:127}$ ← 0
na ← 0x10
i ← (EA & 0xF) * 8
addr ← EA & 0xF
pivot ← addr
do while (addr < na)
    $vt_{i:i+7}$ ← 0b11111111
    i ← i + 8
    addr ← addr + 1
end
vD ← ROTL(vt, pivot * 8)
```

Let X be the low-order 4 bits of EA and vt be a temporary vector. Byte elements 0 through X-1 of vt are set to 0. Byte elements X through 15 of vt are set to 0b11111111. The bytes of vt are then rotated to place the bytes with all bits set in the leftmost elements of the vector and the results are placed in vD.

The vector created by **lvsm** is used as a control vector to select (using **vsel**) the left and right elements of a vector from 2 iterations of a loop that uses **lvswx** to load (or **stvswx** to store) left and right portions of a vector. The EA for **lvsm** corresponds to the address that is used to divide the left and right portions of the load operation.

lvsm will generally be performed at the beginning of a loop with EA pointing to the address of the start of a vector in memory, using the resulting vector register as a control register for selecting out the left and right portions of **lvswx** loaded (or **stvswx** stored) vectors using **vsel**.

Other registers altered:

- None

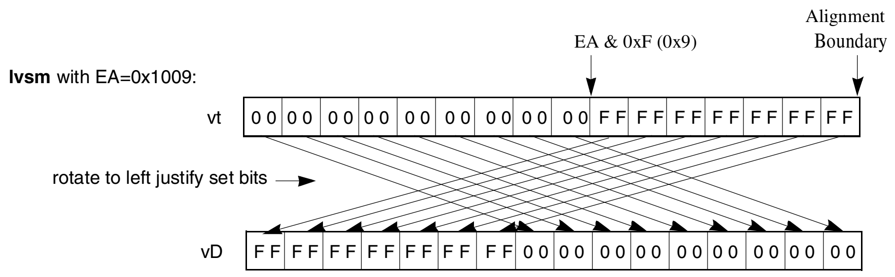

Alignment Boundary
EA & 0xF (0x9)
lvsm with EA=0x1009:
vt
0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | F F | F F | F F | F F | F F | F F | F F
rotate to left justify set bits
vD
F F | F F | F F | F F | F F | F F | F F | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0

Figure 6-11. Example lvsm Instruction

lvsr lvsr

Load Vector for Shift Right

lvsr vD,**r**A,**r**B Form X

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

```
if rA = 0 then b ← 0
else           b ← (rA)
addr ← b + (rB)
sh ← addr$_{60:63}$
if sh=0x0 then vD ← 0x101112131415161718191A1B1C1D1E1F
if sh=0x1 then vD ← 0x0F101112131415161718191A1B1C1D1E
if sh=0x2 then vD ← 0x0E0F101112131415161718191A1B1C1D
if sh=0x3 then vD ← 0x0D0E0F101112131415161718191A1B1C
if sh=0x4 then vD ← 0x0C0D0E0F101112131415161718191A1B
if sh=0x5 then vD ← 0x0B0C0D0E0F101112131415161718191A
if sh=0x6 then vD ← 0x0A0B0C0D0E0F10111213141516171819
if sh=0x7 then vD ← 0x090A0B0C0D0E0F101112131415161718
if sh=0x8 then vD ← 0x08090A0B0C0D0E0F1011121314151617
if sh=0x9 then vD ← 0x0708090A0B0C0D0E0F10111213141516
if sh=0xA then vD ← 0x060708090A0B0C0D0E0F101112131415
if sh=0xB then vD ← 0x05060708090A0B0C0D0E0F1011121314
if sh=0xC then vD ← 0x0405060708090A0B0C0D0E0F10111213
if sh=0xD then vD ← 0x030405060708090A0B0C0D0E0F101112
if sh=0xE then vD ← 0x02030405060708090A0B0C0D0E0F1011
if sh=0xF then vD ← 0x0102030405060708090A0B0C0D0E0F10
```

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

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

These instructions can also be used to rotate or shift the contents of a vector register by sh bytes. For rotating, the vector register to be rotated should be specified as both **v**A and **v**B for **vperm**. For shifting left, the **v**B register for **vperm** should contain all zeros and **v**A should contain the value to be shifted, and vice versa for shifting right. Figure 6-9 shows a similar instruction only in that figure the shift is to the left.

Other registers altered:

- None

lvswx lvswx

Load Vector with Left-Right Swap Indexed

lvswx vD,**r**A,**r**B (LRU=0) Form X

0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	613	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
vt0:127 ← MEM(sa,16)
vD ← ROTL(vt, pivot * 8)
```

Bytes addressed by EA up to, but not including the next quad word alignment boundary are loaded from memory into the most significant bytes in **v**D. Bytes addressed by EA truncated to the quad word alignment boundary up to but not including the byte addressed by EA are loaded from memory into the least significant bytes in **v**D.

lvswx performs both a “load to left” and “load to right” operation using EA as a pivot point within the aligned quad word boundary where EA points. Bytes starting at EA up to the next quad word alignment boundary are loaded from memory left justified into **v**D and bytes starting at EA truncated to a quad word alignment are loaded right justified into **v**D. If EA is already quad word aligned, the vector is loaded in the same as if it was loaded with a **lvx** instruction.

Figure 6-12 shows an example of **lvswx**.

Other registers altered:

- None

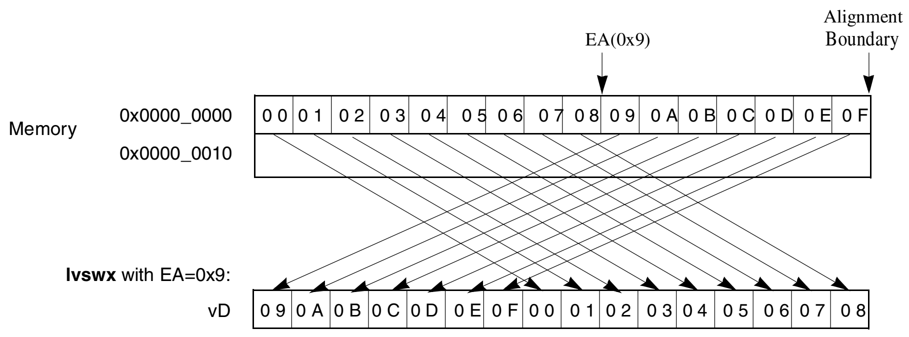

Alignment Boundary
EA(0x9)
Memory
0x0000_0000
0x0000_0010
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
lvswx with EA=0x9:
vD
09 0A 0B 0C 0D 0E 0F 00 01 02 03 04 05 06 07 08

Figure 6-12. Example lvswx Instruction

Software Note

The **lvswx** instruction performs the functions of both the **lvtlx** and **lvtrx** instructions putting the result in a single register (and thus performing only one memory access). The results of a **lvswx** instruction normally contains data from 2 iterations of a loop. The data on the right side of the register contains information for the current iteration and the data on the left side of the register contains information for the next iteration of the loop. The **lvsm** instruction is used to create a select vector for the **vsel** instruction to select the left and right parts to form a full vector for each iteration of the loop.

Implementation Note

The left/right swap is merely a byte rotation of the vector in memory truncated to a vector alignment. The number of bytes that are rotated is the offset from the aligned memory address that is specified by EA. For example, if EA = 0x116, the vector at 0x110 would be loaded, then rotated left by 6 bytes.

lvswxl lvswxl

Load Vector with Left-Right Swap Indexed LRU

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

0 5	6 10	11 15	16 20	21 30	31
31	vD	rA	rB	869	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
vt_{0:127} ← MEM(sa,16)
vD ← ROTL(vt, pivot * 8)
```

Bytes addressed by EA up to, but not including the next quad word alignment boundary are loaded from memory into the most significant bytes in **v**D. Bytes addressed by EA truncated to the quad word alignment boundary up to but not including the byte addressed by EA are loaded from memory into the least significant bytes in **v**D.

lvswxl performs both a “load to left” and “load to right” operation using EA as a pivot point within the aligned quad word boundary where EA points. Bytes starting at EA up to the next quad word alignment boundary are loaded from memory left justified into **v**D and bytes starting at EA truncated to a quad word alignment are loaded right justified into **v**D. If EA is already quad word aligned, the vector is loaded in the same as if it was loaded with a **lvxl** instruction.

lvswxl provides a hint that the program may not need the cache line addressed by EA again soon.

Figure 6-13 shows an example of **lvswxl**.

Other registers altered:

- None

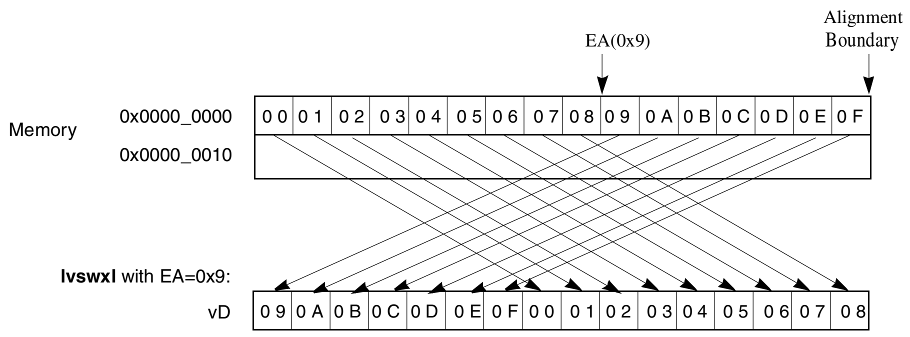

Alignment Boundary
EA(0x9)
Memory
0x0000_0000
0x0000_0010
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
lvswxl with EA=0x9:
vD
09 0A 0B 0C 0D 0E 0F 00 01 02 03 04 05 06 07 08

Figure 6-13. Example lvswxl Instruction

lvx lvx

Load Vector Indexed

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

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

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

The quad word in memory addressed by EA is loaded into **v**D.

Figure 6-9 shows this instruction.

Other registers altered:

- None

lvxl lvxl

Load Vector Indexed LRU

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

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

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

The quad word addressed by EA is loaded into **v**D.

lvxl provides a hint that the program may not need the quad word addressed by EA again soon.

Note that on some implementations, the hint provided by the **lvxl** instruction and the corresponding hint provided by the Store Vector Indexed LRU (**stvxl**) instruction are applied to the entire cache block containing the specified quad word. 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 quad word 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 quad word if the subsequent references are likely to occur sufficiently soon that the cache block containing the quad word is not likely to be displaced from the cache before the last reference. Figure 6-9 shows this instruction.

Other registers altered:

- None

mfvscr mfvscr

Move from Vector Status and Control Register

mfvscr **vD** Form VX

0 5	6 10	11 20	21 31
04	vD	///	1540

```
vD ← $^{96}0$ || (VSCR)
```

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

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

Other registers altered:

- None

mtvscr mtvscr

Move to Vector Status and Control Register

mtvscr **v**B Form VX

0 5	6 15	16 20	21 31
04	///	vB	1604

```
VSCR ← (vB)$_{96:127}$
```

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

Other registers altered:

- None

mvidsplt

64, V	User

Move to Vector from Integer Double Word and Splat

mvidsplt **v**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	110	/

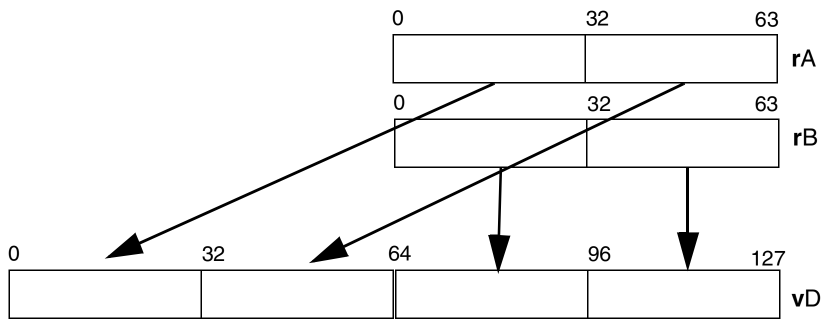
```
vD_0:127 ← (rA)_0:63 || (rB)_0:63
```

The contents of **r**A are copied to the most significant 64 bits of Vector register **v**D and the contents of **r**B are copied to the least significant 64 bits of Vector register **v**D.

Other registers altered:

- None

This instruction is defined for 64-bit implementations only.



mviwsplt mviwsplt

Move to Vector from Integer Word and Splat

mviwsplt vD,**r**A,**r**B

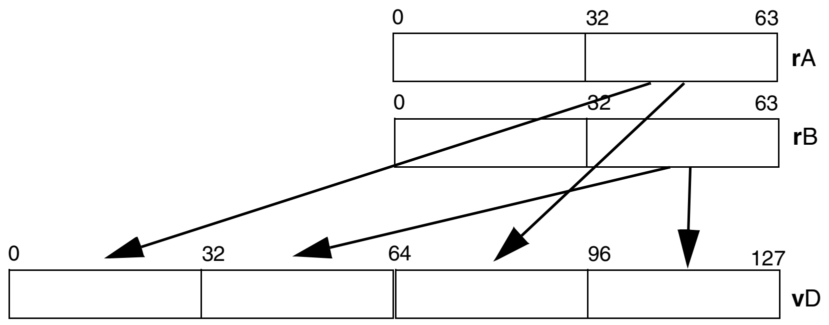
0–5	6–10	11–15	16–20	21–30	31
31	vD	rA	rB	46	/

```
vD0:63 ← (rA)32:63 || (rB)32:63
vD64:127 ← (rA)32:63 || (rB)32:63
```

The least significant 32 bits of **r**A are concatenated with the least significant 32 bits of **r**B and are copied to the most significant 64 bits of Vector register **v**D and the least significant 64 bits of Vector register **v**D.

Other registers altered:

- None



stvebx stvebx

Store Vector Element Byte Indexed

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

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

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← b + (rB)
eb ← $EA_{60:63}$
MEM(EA,1) ← $(vS)_{eb*8:(eb*8)+7}$
```

Let m be the byte element of the vector indexed by EA[60–63]. Byte m of **v**S is stored into the byte in memory addressed by EA. Figure 6-1 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvehx stvehx

Store Vector Element Half Word Indexed

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

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

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & (~0x1)
eb ← $EA_{60:63}$
MEM(EA,2) ← $(vS)_{eb*8:(eb*8)+15}$
```

Let m be the half word element of the vector indexed by EA[60–62]. Half word m of **v**S is stored into the half word addressed by EA. Figure 6-1 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvepx

E.PD, V	Supervisor

stvepx

Store Vector by External PID Indexed

stvepx vS,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	vS	rA	rB	1 1 0 0 1 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16) ← (vS)
```

The contents of **v**S are stored into the quadword addressed by EA truncated to the nearest quadword boundary. The data is stored in big-endian form regardless of the setting of the E storage attribute.

This instruction is guest supervisor privileged.

An attempt to execute **stvepx** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **stvepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value.
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID,ST,SPV] (GESR[EPID,ST,SPV]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stvepxl

E.PD, V	Supervisor

stvepxl

Store Vector by External PID Indexed

stvepxl **v**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	vS	rA	rB	1 1 0 0 0 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16)← (vS)
```

The contents of **v**S are stored into the quadword addressed by EA truncated to the nearest quadword boundary. The data is stored in big-endian form regardless of the setting of the E storage attribute.

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

This instruction is guest supervisor privileged.

An attempt to execute **stvepxl** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **stvepxl**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value.
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID,ST,SPV] (GESR[EPID,ST,SPV]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stvewx stvewx

Store Vector Element Word Indexed

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

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

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & ~0x3
eb ← $EA_{60:63}$
MEM(EA,4) ← $(vS)_{eb*8:(eb*8)+31}$
```

Let m be the word element of the vector indexed by EA[60–61]. Word m of **v**S is stored into the word addressed by EA. Figure 6-1 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvexbx stvexbx

Store Vector Element Indexed Byte Indexed

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

0–5	6–10	11–15	16–20	21–30	31
31	vS	rA	rB	389	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
e ← (rB)$_{60:63}$
MEM(EA,1) ← (vS)$_{e*8:(e*8)+7}$
```

Let E be the byte element of vector register **v**S indexed by **r**B[60–63]. Byte E of **v**S is stored into the byte addressed by EA.

Other registers altered:

- None

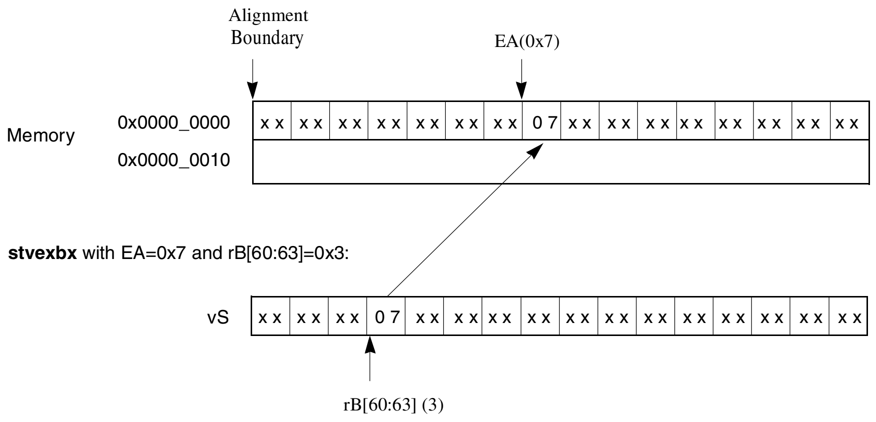

Alignment Boundary
EA(0x7)
Memory
0x0000_0000
0x0000_0010
xx xx xx xx xx xx xx 07 xx xx xx xx xx xx xx xx
stvexbx with EA=0x7 and rB[60:63]=0x3:
vS
xx xx xx 07 xx xx xx xx xx xx xx xx xx xx xx xx
rB[60:63] (3)

Figure 6-14. Example stvexbx Instruction

stvexhx stvexhx

Store Vector Element Indexed Half Word Indexed

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

0 5	6 10	11 15	16 20	21 30	31
31	vS	rA	rB	421	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & ~0x1
e ← (rB)60:62
MEM(EA,2) ← (vS)e*16:(e*16)+15
```

Let E be the half word element of vector register **v**S indexed by **r**B[60–62]. Half word E of **v**S is stored into the half word addressed by EA.

Other registers altered:

- None

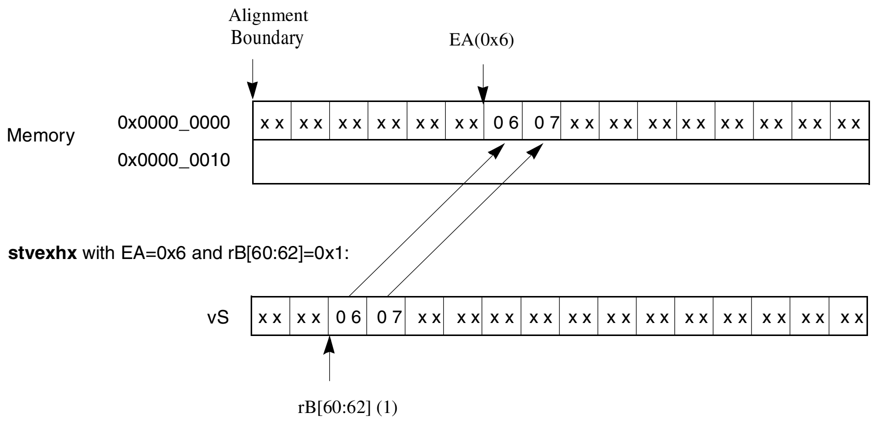

Alignment Boundary
EA(0x6)
Memory
0x0000_0000
0x0000_0010
x x x x x x x x x x x x 0 6 0 7 x x x x x x x x x x x x x x x x
stvexhx with EA=0x6 and rB[60:62]=0x1:
vS
x x x x 0 6 0 7 x x x x x x x x x x x x x x x x x x x x x x x x
rB[60:62] (1)

Figure 6-15. Example stvexhx Instruction

stvexwx stvexwx

Store Vector Element Indexed Word Indexed

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

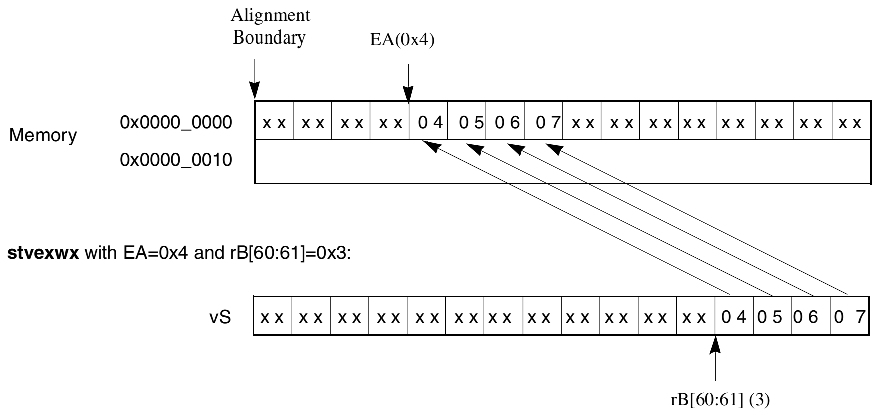
0–5	6–10	11–15	16–20	21–30	31
31	vS	rA	rB	453	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB)) & ~0x3
e ← (rB)$_{60:61}$
MEM(EA,4) ← (vS)$_{e*32:(e*32)+31}$
```

Let E be the word element of vector register **v**S indexed by **r**B[60–61]. Word E of **v**S is stored into the word addressed by EA.

Other registers altered:

- None


Alignment Boundary
EA(0x4)
Memory
0x0000_0000
0x0000_0010
xx xx xx xx 04 05 06 07 xx xx xx xx xx xx xx xx
stvexwx with EA=0x4 and rB[60:61]=0x3:
vS
xx xx xx xx xx xx xx xx xx xx xx xx 04 05 06 07
rB[60:61] (3)

stvflx stvflx

Store Vector from Left Indexed

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

0–5	6–10	11–15	16–20	21–30	31
31	vS	rA	rB	709	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
n ← 0x10 - (EA & 0xF)
i ← 0
addr ← EA
do while (n > 0)
    MEM(addr,1) ← (vS)i:i+7
    addr ← addr + 1
    i ← i + 8
    n ← n - 1
end
```

Bytes of vS, starting with the most significant byte, are stored to memory starting at EA up to, but not including, the byte at the next largest quad word alignment boundary .

stvflx performs a “store from left” operation, where the bytes starting at EA up to the next quad word alignment boundary are stored from the most significant bytes of **v**S.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-16 shows an example of **stvflx**.

Other registers altered:

- None

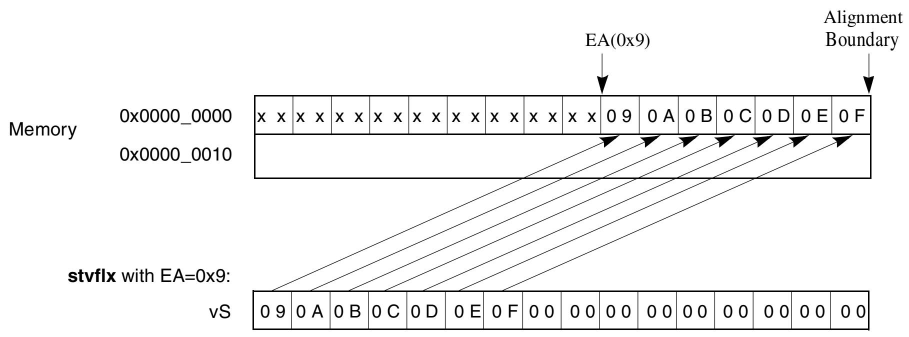

Alignment Boundary
EA(0x9)
Memory
0x0000_0000
0x0000_0010
x x x x x x x x x x x x x x x x x x 0 9 0 A 0 B 0 C 0 D 0 E 0 F
stvflx with EA=0x9:
vS
0 9 0 A 0 B 0 C 0 D 0 E 0 F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figure 6-16. Example stvflx Instruction

stvflxl stvflxl

Store Vector from Left Indexed LRU

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

0 5	6 10	11 15	16 20	21 30	31
31	vS	rA	rB	965	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
n ← 0x10 - (EA & 0xF)
i ← 0
addr ← EA
do while (n > 0)
    MEM(addr,1) ← (vS)i:i+7
    addr ← addr + 1
    i ← i + 8
    n ← n - 1
end
```

Bytes of vS, starting with the most significant byte, are stored to memory starting at EA up to, but not including, the byte at the next largest quad word alignment boundary .

stvflxl performs a "store from left" operation, where the bytes starting at EA up to the next quad word alignment boundary are stored from the most significant bytes of **v**S.

stvflxl provides a hint that the program may not need the cache line addressed by EA again soon.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-17 shows an example of **stvflxl**.

Other registers altered:

- None

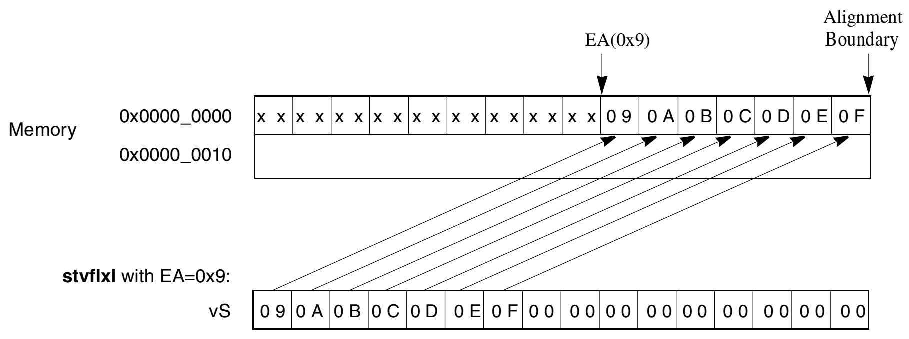

Alignment Boundary
EA(0x9)
Memory
0x0000_0000
0x0000_0010
x x x x x x x x x x x x x x x x x x 0 9 0 A 0 B 0 C 0 D 0 E 0 F
stvflxl with EA=0x9:
vS
0 9 0 A 0 B 0 C 0 D 0 E 0 F 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figure 6-17. Example stvflxl Instruction

stvfrx stvfrx

Store Vector from Right Indexed

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

0 5	6 10	11 15	16 20	21 30	31
31	vS	rA	rB	677	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
addr ← EA & ~(0xF)
n ← EA & 0xF
i ← 128 - (n * 8)
do while (n > 0)
    MEM(addr,1) ← (vS)$_{i:i+7}$
    addr ← addr + 1
    i ← i + 8
    n ← n - 1
end
```

Bytes of vS, starting with the least significant byte, are stored to memory starting at the byte prior to EA down to, and including, the byte at EA truncated to a quad word alignment boundary.

stvfrx performs a “store from right” operation, where the bytes starting at EA truncated to a quad word boundary, up to but not including, EA are stored from the least significant bytes of **v**S.

If EA points to the first byte in a quad word, no memory access is performed but protection and debug events are still performed for address EA.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-18 shows an example of **stvfrx**.

Other registers altered:

- None

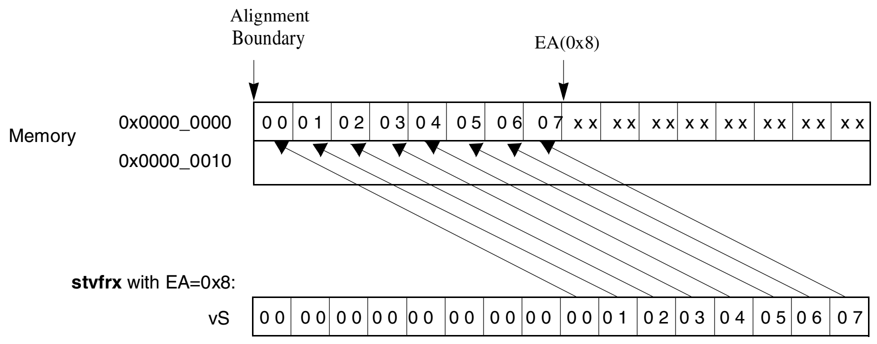


Figure 6-18. Example stvfrx Instruction

stvfrxl stvfrxl

Store Vector from Right Indexed LRU

stvfrxl vS,**r**A,**r**B (LRU=1) Form X

0–5	6–10	11–15	16–20	21–30	31
31	vS	rA	rB	933	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
addr ← EA & ~(0xF)
n ← EA & 0xF
i ← 128 - (n * 8)
do while (n > 0)
    MEM(addr,1) ← $(vS)_{i:i+7}$
    addr ← addr + 1
    i ← i + 8
    n ← n - 1
end
```

Bytes of vS, starting with the least significant byte, are stored to memory starting at the byte prior to EA down to, and including, the byte at EA truncated to a quad word alignment boundary.

stvfrxl performs a “store from right” operation, where the bytes starting at EA truncated to a quad word boundary, up to but not including, EA are stored from the least significant bytes of **v**S.

If EA points to the first byte in a quad word, no memory access is performed but protection and debug events are still performed for address EA.

stvfrxl provides a hint that the program may not need the cache line addressed by EA again soon.

On some implementations, if EA is in memory that is caching-inhibited or write through required, an alignment interrupt may be taken.

Figure 6-19 shows an example of **stvfrxl**.

Other registers altered:

- None

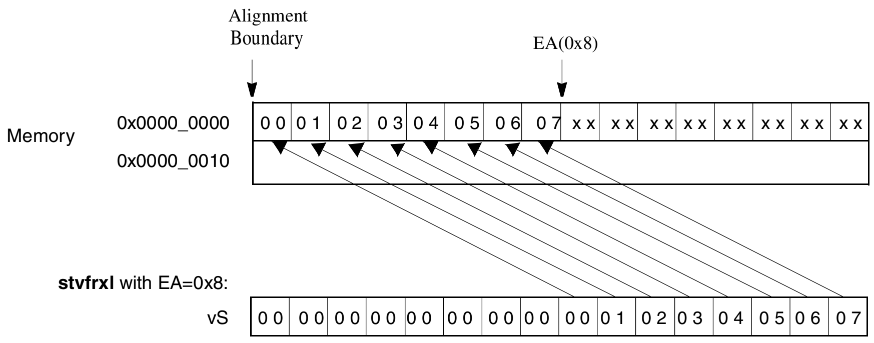


Figure 6-19. Example stvfrxl Instruction

stvswx stvswx

Store Vector with Left-Right Swap Indexed

stvswx vS,**r**A,**r**B (LRU=0) Form X

0–5	6–10	11–15	16–20	21–30	31
31	vS	rA	rB	741	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
vt0:127 ← ROTR(vD, pivot * 8)
MEM(sa,16) ←vt0:127
```

Bytes of **v**S, starting with the most significant byte, are stored to memory starting at EA up to, but not including, the byte at the next largest quad word alignment boundary. Bytes of **v**S, starting with the least significant byte, are stored to memory starting at EA-1 down to, and including, the byte at EA truncated to a quad word alignment boundary.

stvswx performs both a "store from left" and "store from right" operation using EA as a pivot point within the aligned quad word boundary where EA points. Bytes starting at the most significant byte (left most) of **v**S are stored starting at EA up to, but not including the next quad word alignment boundary. The remaining bytes in **v**S are stored starting at EA truncated to a quad word alignment boundary. If EA is already quad word aligned, the vector is stored the same as if it was stored with a **stvx** instruction.

Figure 6-20 shows an example of **stvswx**.

Other registers altered:

- None

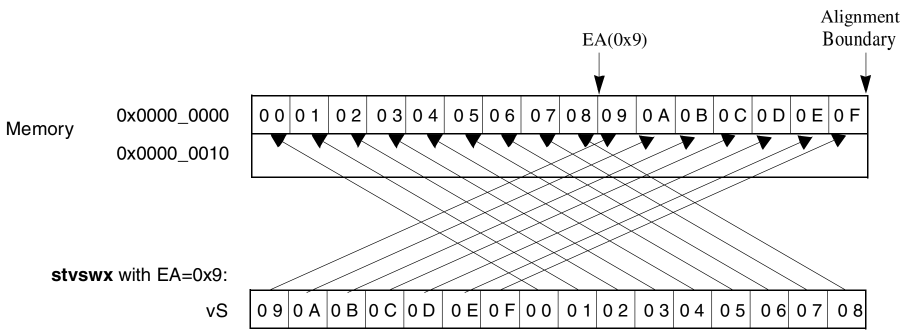

Alignment Boundary
EA(0x9)
Memory
0x0000_0000
0x0000_0010
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
stvswx with EA=0x9:
vS
09 0A 0B 0C 0D 0E 0F 00 01 02 03 04 05 06 07 08

Figure 6-20. Example stvswx Instruction

Software Note

The **stvswx** instruction performs the functions of both the **stvflx** and **stvfrx** instructions storing from a single register (and thus performing only one memory access). The results of a **stvswx** instruction normally store data from 2 iterations of a loop. The data on the right side of the register contains information for the current iteration and the data on the left side of the register contains information for the next iteration of the loop. The **lvsm** instruction is used to create a select vector for the **vsel** instruction to select the left and right parts to form a full vector for each iteration of the loop.

stvswxl stvswxl

Store Vector with Left-Right Swap Indexed LRU

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

0–5	6–10	11–15	16–20	21–30	31
31	vS	rA	rB	997	/

```
if rA=0 then b ← 0
else         b ← (rA)
EA ← (b + (rB))
sa ← EA & ~(0xF)
pivot ← EA & 0xF
vt0:127 ← ROTR(vD, pivot * 8)
MEM(sa,16) ←vt0:127
```

Bytes of **v**S, starting with the most significant byte, are stored to memory starting at EA up to, but not including, the byte at the next largest quad word alignment boundary. Bytes of **v**S, starting with the least significant byte, are stored to memory starting at EA-1 down to, and including, the byte at EA truncated to a quad word alignment boundary.

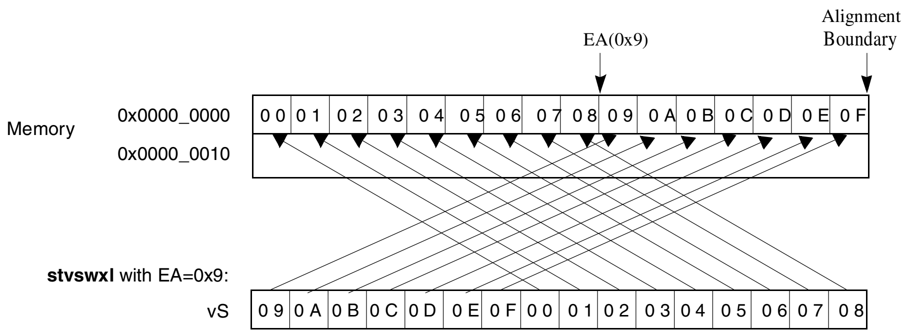
stvswxl performs both a “store from left” and “store from right” operation using EA as a pivot point within the aligned quad word boundary where EA points. Bytes starting at the most significant byte (left most) of **v**S are stored starting at EA up to, but not including the next quad word alignment boundary. The remaining bytes in **v**S are stored starting at EA truncated to a quad word alignment boundary. If EA is already quad word aligned, the vector is stored the same as if it was stored with a **stvxl** instruction.

stvswxl provides a hint that the program may not need the cache line addressed by EA again soon.

Figure 6-21 shows an example of **stvswxl**.

Other registers altered:

- None



Alignment Boundary
EA(0x9)
Memory
0x0000_0000
0x0000_0010
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
stvswxl with EA=0x9:
vS
09 0A 0B 0C 0D 0E 0F 00 01 02 03 04 05 06 07 08

Figure 6-21. Example stvswxl Instruction

stvx stvx

Store Vector Indexed

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

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

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

The contents of **v**S are stored into the quad word addressed by EA.

Figure 6-1 shows how a store instruction is performed for a vector register.

Other registers altered:

- None

stvxl stvxl

Store Vector Indexed LRU

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

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

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

The contents of **v**S are stored into the quad word addressed by EA.

The **stvxl** instruction provides a hint that the quad word addressed by EA will probably not be needed again by the program in the near future.

Note that on some implementations, the hint provided by the **stvxl** instruction (see Section 5.1, "Prioritizing Cache Block Replacement") is applied to the entire cache block containing the specified quad word. 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 quad word 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 quad word if the subsequent references are likely to occur sufficiently soon that the cache block containing the quad word is not likely to be displaced from the cache before the last reference. Figure 6-1 shows how a store instruction is performed on the vector registers.

Other registers altered:

- None

vabsdub vabsdub

Vector Absolute Difference Unsigned Byte

vabsdub vD,vA,vB Form VX

0–5	6–10	11–15	16–20	21–31
04	vD	vA	vB	1027

```
do i = 0 to 127 by 8
    aop$_{0:8}$ ← ZeroExtend((vA)$_{i:i+7}$,9)
    bop$_{0:8}$ ← ZeroExtend((vB)$_{i:i+7}$,9)
    temp$_{0:8}$ ← aop$_{0:8}$ $-_{int}$ bop$_{0:8}$
    vD$_{i:i+7}$ ← ABS(temp$_{0:8}$)
end
```

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

Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The elements of **v**A and **v**B are treated as unsigned integers. The absolute value of the result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-22 shows the **vabsdub** instruction usage. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

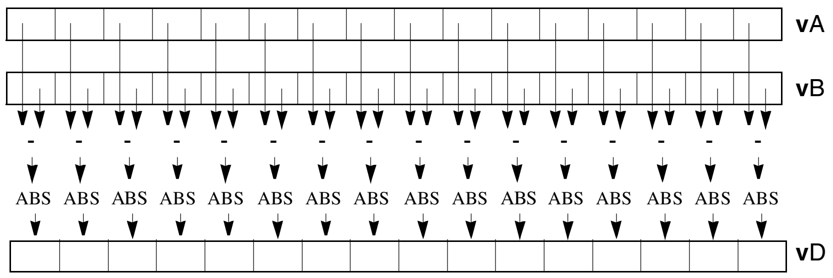


Figure 6-22. vabsdub—Absolute Differences Sixteen Integer Elements (8-Bit)

vabsduh vabsduh

Vector Absolute Difference Unsigned Half Word

vabsduh vD,vA,vB Form VX

0–5	6–10	11–15	16–20	21–31
04	vD	vA	vB	1091

```
do i = 0 to 127 by 16
    aop$_{0:16}$ ← ZeroExtend((vA)$_{i:i+15}$,16)
    bop$_{0:16}$ ← ZeroExtend((vB)$_{i:i+15}$,16)
    temp$_{0:16}$ ← aop$_{0:16}$ $-_{int}$ bop$_{0:16}$
    vD$_{i:i+15}$ ← ABS(temp$_{0:16}$)
end
```

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

Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The elements of **v**A and **v**B are treated as unsigned integers. The absolute value of the result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-23 shows the **vabsduh** instruction usage. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

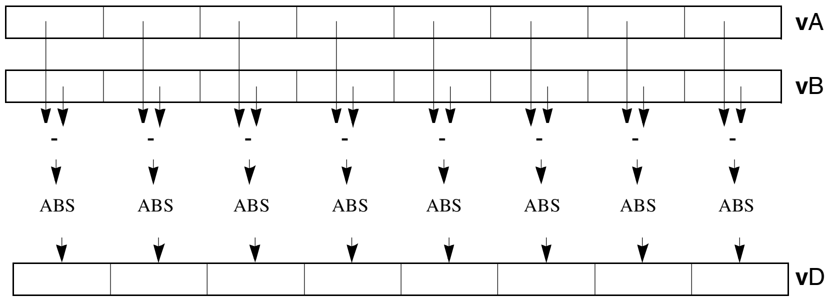


Figure 6-23. vabsduh—Absolute Differences Eight Integer Elements (16-Bit)

vabsduw vabsduw

Vector Absolute Difference Unsigned Word

vabsduw vD,vA,vB Form VX

0–5	6–10	11–15	16–20	21–31
04	vD	vA	vB	1155

```
do i = 0 to 127 by 32
    $aop_{0:32}$ ← ZeroExtend((vA)$_{i:i+31}$,32)
    $bop_{0:32}$ ← ZeroExtend((vB)$_{i:i+31}$,32)
    $temp_{0:32}$ ← $aop_{0:32}$ $-_{int}$ $bop_{0:32}$
    $vD_{i:i+31}$ ← ABS($temp_{0:32}$)
end
```

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

Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The elements of **v**A and **v**B are treated as unsigned integers. The absolute value of the result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-24 shows the **vabsduw** instruction usage. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

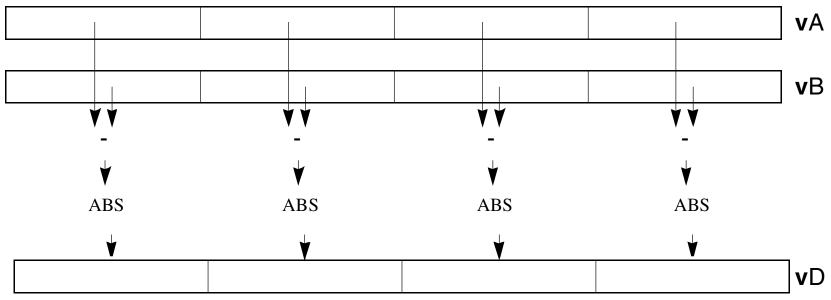


Figure 6-24. vabsduw—Absolute Differences Four Integer Elements (32-Bit)

vaddcuw vaddcuw

Vector Add Carryout Unsigned Word

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

04	vD	vA	vB	384
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $aop_{0:32}$ ← ZeroExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32}$ ← ZeroExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32}$ $+_{int}$ $bop_{0:32}$
    $vD_{i:i+31}$ ← ZeroExtend($temp_0$,32)
end
```

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

Other registers altered:

- None

Figure 6-25 shows the usage of the **vaddcuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

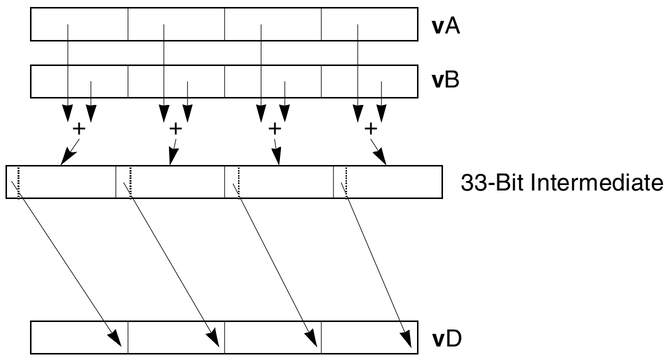


Figure 6-25. vaddcuw—Determine Carries of Four Unsigned Integer Adds (32-Bit)

vaddfp vaddfp

Vector Add Floating-Point

vaddfp vD,vA,vB Form VX

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

```
do i = 0 to 127 by 32
    $(vD)_{i:i+31}$ ← RndToNearFP32($(vA)_{i:i+31}$ $+_{fp}$ $(vB)_{i:i+31}$)
end
```

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

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

Other registers altered:

- None

Figure 6-26 shows the usage of the **vaddfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

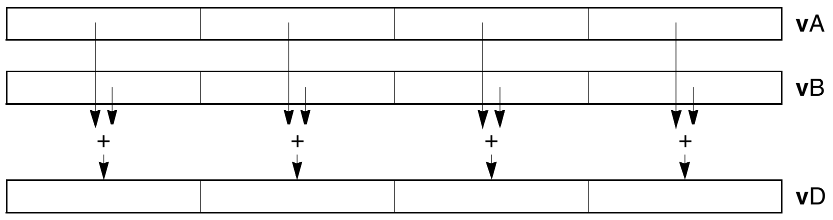


Figure 6-26. vaddfp—Add Four Floating-Point Elements (32-Bit)

vaddsbs vaddsbs

Vector Add Signed Byte Saturate

vaddsbs vD,vA,vB Form VX

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

```
do i = 0 to 127 by 8
    $aop_{0:8}$ ← SignExtend((vA)$_{i:i+7}$,9)
    $bop_{0:8}$ ← SignExtend((vB)$_{i:i+7}$,9)
    $temp_{0:8}$ ← $aop_{0:8}$ $+_{int}$ $bop_{0:8}$
    $vD_{i:i+7}$ ← SItoSIsat($temp_{0:8}$,8)
end
```

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

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

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

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

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-27 shows the usage of the **vaddsbs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

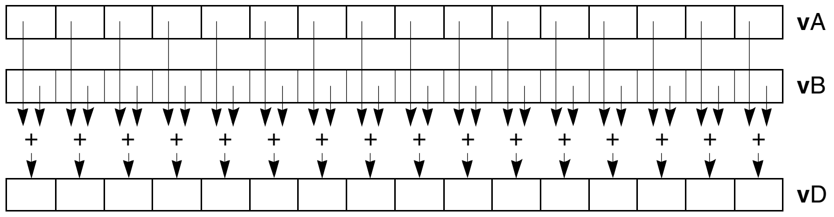


Figure 6-27. vaddsbs—Add Saturating Sixteen Signed Integer Elements (8-Bit)

vaddshs vaddshs

Vector Add Signed Half Word Saturate

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

04	vD	vA	vB	832
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 16
    $aop_{0:16} \leftarrow$ SignExtend((vA)$_{i:i+15}$,16)
    $bop_{0:16} \leftarrow$ SignExtend((vB)$_{i:i+15}$,16)
    $temp_{0:16} \leftarrow aop_{0:16} +_{int} bop_{0:16}$
    $vD_{i:i+15} \leftarrow$ SItoSIsat($temp_{0:16}$,16)
end
```

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

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

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

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

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-28 shows the usage of the **vaddshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

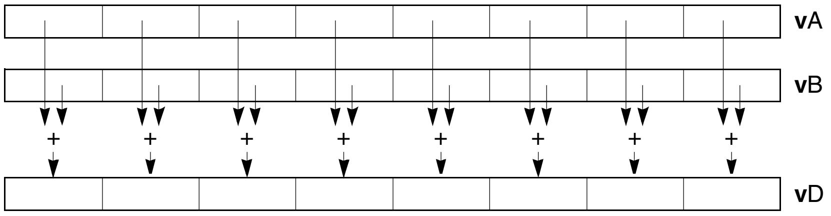


Figure 6-28. vaddshs—Add Saturating Eight Signed Integer Elements (16-Bit)

vaddsws vaddsws

Vector Add Signed Word Saturate

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

04	vD	vA	vB	896
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 32
    $aop_{0:32} \leftarrow$ SignExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32} \leftarrow$ SignExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32} \leftarrow aop_{0:32} +_{int} bop_{0:32}$
    $vD_{i:i+31} \leftarrow$ SItoSIsat($temp_{0:32}$,32)
end
```

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

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

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

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

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-29 shows the usage of the **vaddsws** instruction. Each of the 4 elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

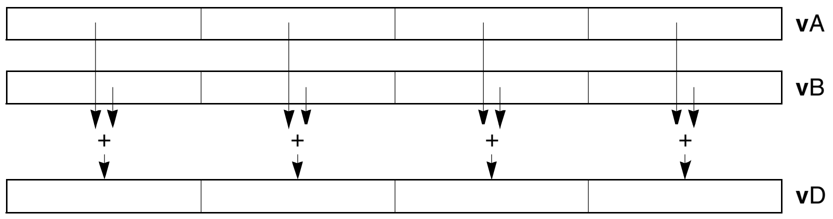


Figure 6-29. vaddsws—Add Saturating Four Signed Integer Elements (32-Bit)

vaddubm vaddubm

Vector Add Unsigned Byte Modulo

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
    $vD_{i:i+7} \leftarrow (vA)_{i:i+7} +_{int} (vB)_{i:i+7}$
end
```

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

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

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

Other registers altered:

- None

Figure 6-30 shows the **vaddubm** instruction usage. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

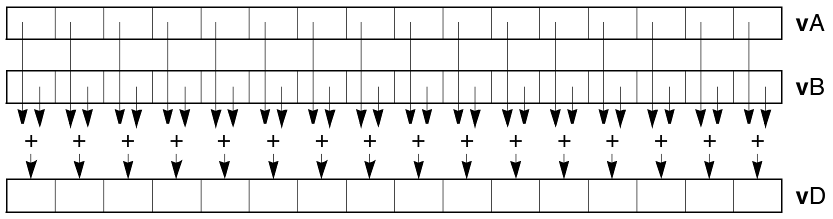


Figure 6-30. vaddubm—Add Sixteen Integer Elements (8-Bit)

vaddubs vaddubs

Vector Add Unsigned Byte Saturate

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

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

```
do i = 0 to 127 by 8
    $aop_{0:8}$ ← ZeroExtend((vA)$_{i:i+7}$,9)
    $bop_{0:8}$ ← ZeroExtend((vB)$_{i:i+7}$,9)
    $temp_{0:8}$ ← $aop_{0:8}$ $+_{int}$ $bop_{0:8}$
    $vD_{i:i+7}$ ← UItoUIsat($temp_{0:8}$,8)
end
```

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

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

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

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

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-31 shows the usage of the **vaddubs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

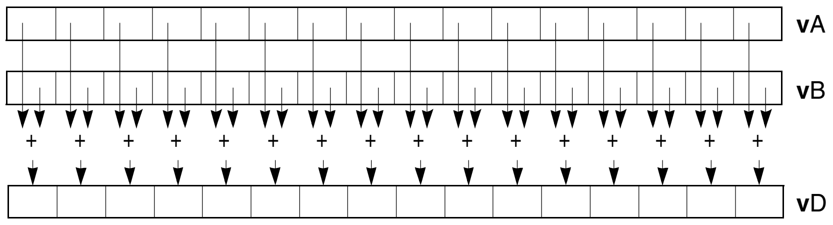


Figure 6-31. vaddubs—Add Saturating Sixteen Unsigned Integer Elements (8-Bit)

vadduhm

Vector Add Unsigned Half Word Modulo

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
    $vD_{i:i+15} \leftarrow (vA)_{i:i+15} +_{int} (vB)_{i:i+15}$
end
```

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

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

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

Other registers altered:

- None

Figure 6-32 shows the usage of the **vadduhm** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

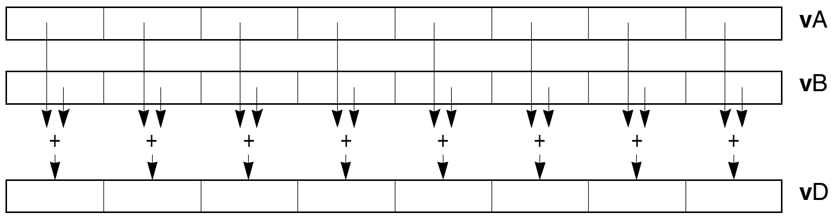


Figure 6-32. vadduhm—Add Eight Integer Elements (16-Bit)

vadduhs   vadduhs

Vector Add Unsigned Half Word Saturate

vadduhs vD,vA,vB Form VX

04	vD	vA	vB	576
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 16
    $aop_{0:16}$ ← ZeroExtend($(vA)_{i:i+15}$,17)
    $bop_{0:16}$ ← ZeroExtend($(vB)_{i:i+15}$,17)
    $temp_{0:16}$ ← $aop_{0:16}$ $+_{int}$ $bop_{0:16}$
    $vD_{i:i+15}$ ← UItoUIsat($temp_{0:16}$,16)
end
```

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

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

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

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

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-33 shows the usage of the **vadduhs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

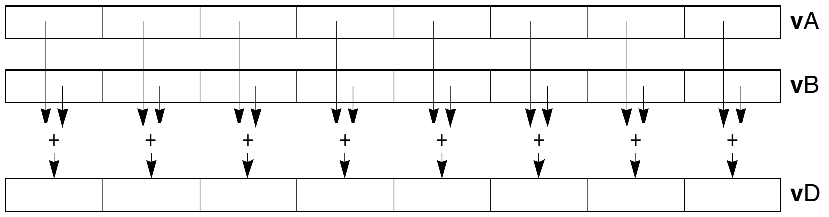


Figure 6-33. vadduhs—Add Saturating Eight Unsigned Integer Elements (16-Bit)

vadduwm

Vector Add Unsigned Word Modulo

vadduwm vD,vA,vB Form: VX

04	vD	vA	vB	128
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 32
    $vD_{i:i+31} \leftarrow (vA)_{i:i+31} +_{int} (vB)_{i:i+31}$
end
```

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

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

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

Other registers altered:

- None

Form:

- VX

Figure 6-34 shows the usage of the **vadduwm** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

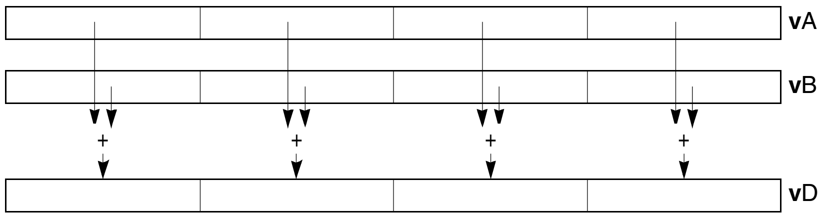


Figure 6-34. vadduwm—Add Four Integer Elements (32-Bit)

vadduws vadduws

Vector Add Unsigned Word Saturate

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

04	vD	vA	vB	640
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 3
    $aop_{0:32}$ ← ZeroExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32}$ ← ZeroExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32}$ $+_{int}$ $bop_{0:32}$
    $vD_{i:i+31}$ ← UItoUIsat($temp_{0:32}$,32)
end
```

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

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

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

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

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-35 shows the usage of the **vadduws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

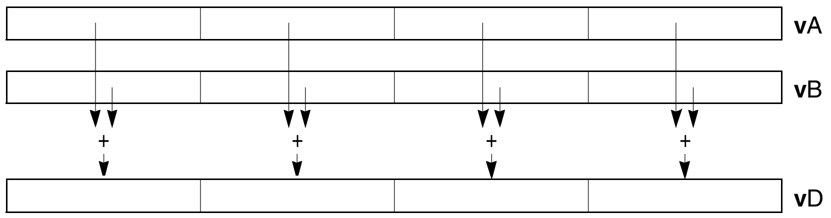


Figure 6-35. vadduws—Add Saturating Four Unsigned Integer Elements (32-Bit)

vand vand

Vector Logical AND

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

04	vD	vA	vB	1028
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) & (vB)
```

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

Other registers altered:

- None

Figure 6-36 shows usage of the **vand** instruction.

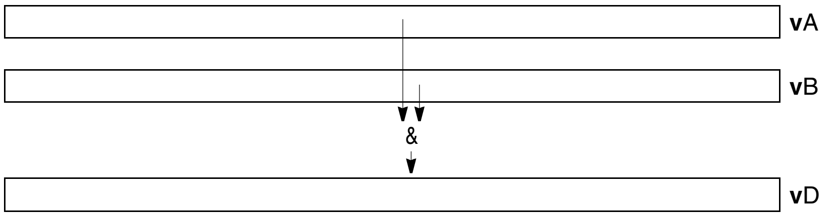


Figure 6-36. vand—Logical Bitwise AND

vandc vandc

Vector Logical AND with Complement

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

04	vD	vA	vB	1092
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) & ¬(vB)
```

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

Other registers altered:

- None

Figure 6-36 shows usage of the **vandc** instruction.

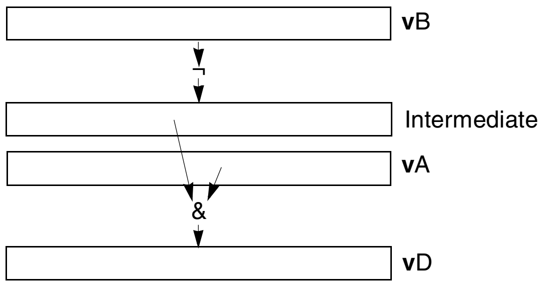


Figure 6-37. vand—Logical Bitwise AND with Complement

vavgsb vavgsb

Vector Average Signed Byte

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

04	vD	vA	vB	1282
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 8
    $aop_{0:8}$ ← SignExtend($(vA)_{i:i+7}$,9)
    $bop_{0:8}$ ← SignExtend($(vB)_{i:i+7}$,9)
    $temp_{0:8}$ ← $aop_{0:8}$ $+_{int}$ $bop_{0:8}$ $+_{int}$ 1
    $vD_{i:i+7}$ ← $temp_{0:7}$
end
```

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

Each signed-integer byte element in **v**A is added to the corresponding signed-integer byte element in **v**B, producing 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 element of **v**D.

Other registers altered:

- None

Figure 6-38 shows the usage of the **vavgsb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

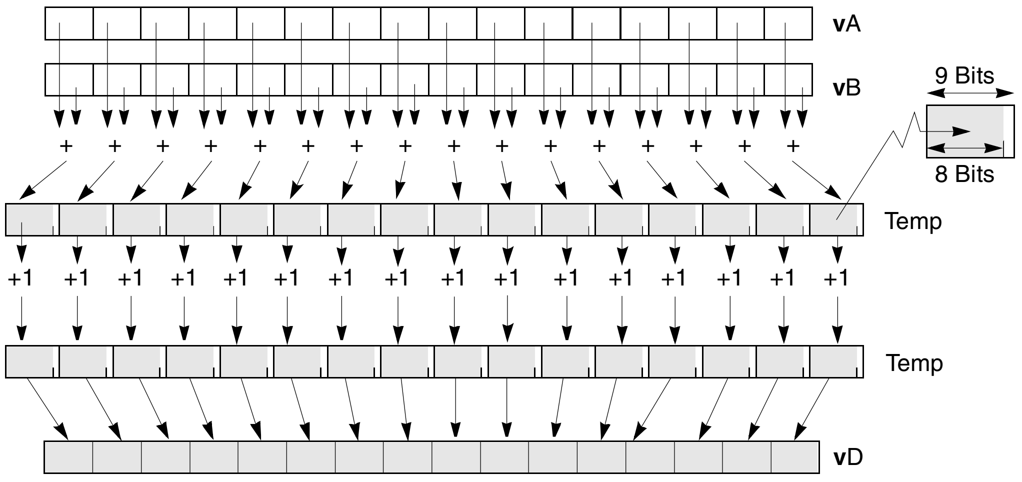


Figure 6-38. vavgsb—Average Sixteen Signed Integer Elements (8-Bit)

vavgsh vavgsh

Vector Average Signed Half Word

vavgsh vD,vA,vB Form: VX

04	vD	vA	vB	1346
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 16
    $aop_{0:16}$ ← SignExtend($(vA)_{i:i+15}$,17)
    $bop_{0:16}$ ← SignExtend($(vB)_{i:i+15}$,17)
    $temp_{0:16}$ ← $aop_{0:15}$ $+_{int}$ $bop_{0:15}$ $+_{int}$ 1
    $vD_{i:i+15}$ ← $temp_{0:15}$
end
```

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

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B, producing an 17-bit signed-integer sum. The sum is incremented by 1. The high-order 16 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-39 shows the usage of the **vavgsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

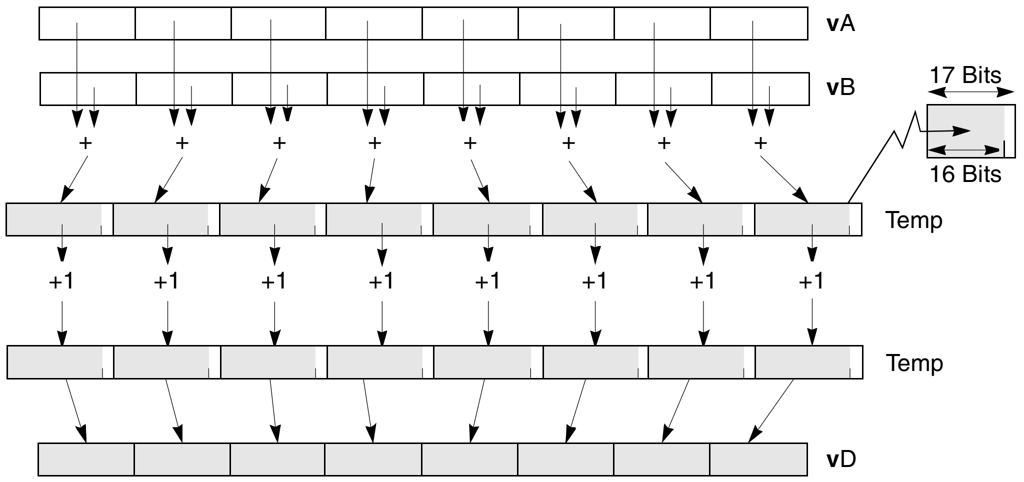


Figure 6-39. vavgsh—Average Eight Signed Integer Elements (16-Bit)

vavgsw vavgsw

Vector Average Signed Word

vavgsw vD,vA,vB Form: VX

04	vD	vA	vB	1410
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 32
    $aop_{0:32}$ ← SignExtend((vA)$_{i:i+31}$,33)
    $bop_{0:3}$ ← SignExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32}$ $+_{int}$ $bop_{0:32}$ $+_{int}$ 1
    $vD_{i:i+31}$ ← $temp_{0:31}$
end
```

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

Each signed-integer element in **v**A is added to the corresponding signed-integer element in **v**B, producing an 33-bit signed-integer sum. The sum is incremented by 1. The high-order 32 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-40 shows the usage of the **vavgsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

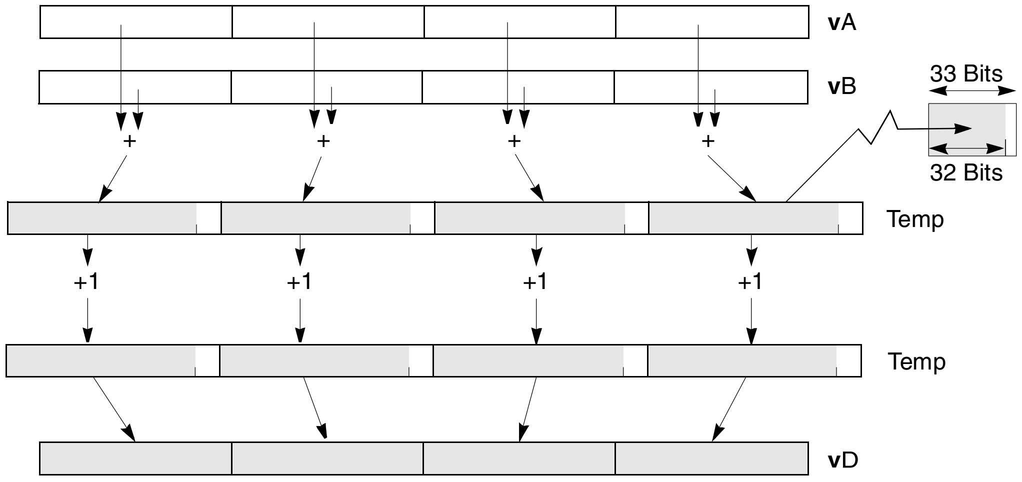


Figure 6-40. vavgsw—Average Four Signed Integer Elements (32-Bit)

vavgub vavgub

Vector Average Unsigned Byte

vavgub vD,vA,vB Form: VX

04	vD	vA	vB	1026
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 8
    aop$_{0:8}$ ← ZeroExtend((vA)$_{i:i+7}$,9)
    bop$_{0:n}$ ← ZeroExtend((vB)$_{i:i+71}$,9)
    temp$_{0:n}$ ← aop$_{0:8}$ +$_{int}$ bop$_{0:8}$ +$_{int}$ 1
    vD$_{i:i+7}$ ← temp$_{0:7}$
end
```

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

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

Other registers altered:

- None

Figure 6-41 shows the usage of the **vavgub** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

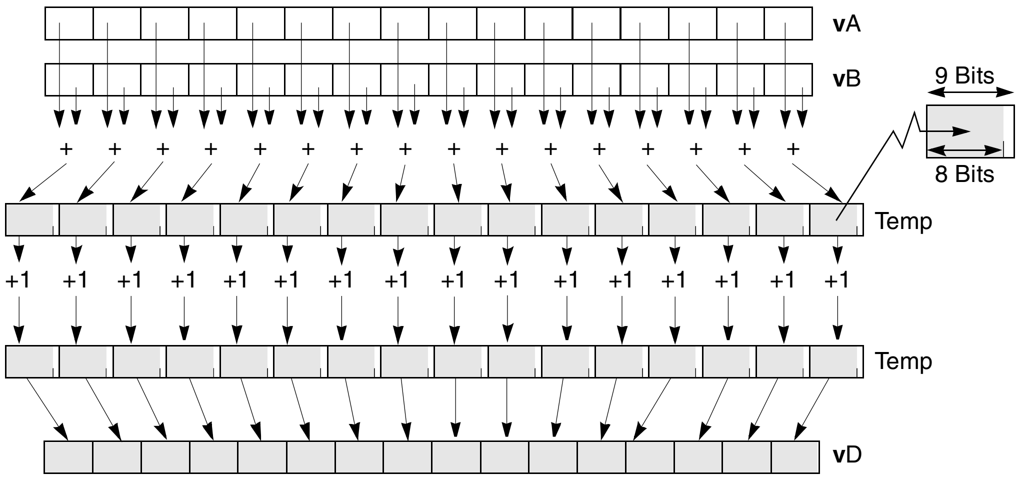


Figure 6-41. vavgub—Average Sixteen Unsigned Integer Elements (8-Bit)

vavguh vavguh

Vector Average Unsigned Half Word

vavguh vD,vA,vB Form: VX

04	vD	vA	vB	1090
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 16
    $aop_{0:16} \leftarrow$ ZeroExtend((vA)$_{i:i+15}$,17)
    $bop_{0:16} \leftarrow$ ZeroExtend((vB)$_{i:i+15}$,17)
    $temp_{0:16} \leftarrow aop_{0:16} +_{int} bop_{0:16} +_{int} 1$
    $vD_{i:i+15} \leftarrow temp_{0:15}$
end
```

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

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing a 17-bit unsigned-integer. The sum is incremented by 1. The high-order 16 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-42 shows the usage of the **vavguh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

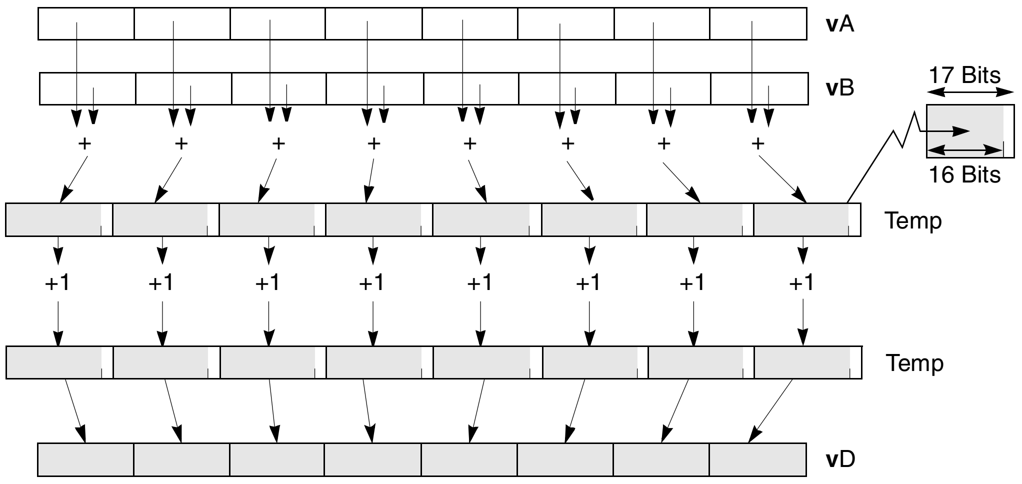


Figure 6-42. vavguh—Average Eight Signed Integer Elements (16-Bit)

vavguw vavguw

Vector Average Unsigned Word

vavguw **vD**,**vA**,**vB** Form: VX

04	vD	vA	vB	1154
0 5	6 10	11 15	16 20	21 31

```
do i = 0 to 127 by 32
    $aop_{0:32}$ ← ZeroExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32}$ ← ZeroExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32}$ $+_{int}$ $bop_{0:32}$ $+_{int}$ 1
    $vD_{i:i+31}$ ← $temp_{0:31}$
end
```

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

Each unsigned-integer element in **v**A is added to the corresponding unsigned-integer element in **v**B, producing a 33-bit unsigned-integer sum. The sum is incremented by 1. The high-order 32 bits of the result are placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-43 shows the usage of the **vavguw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

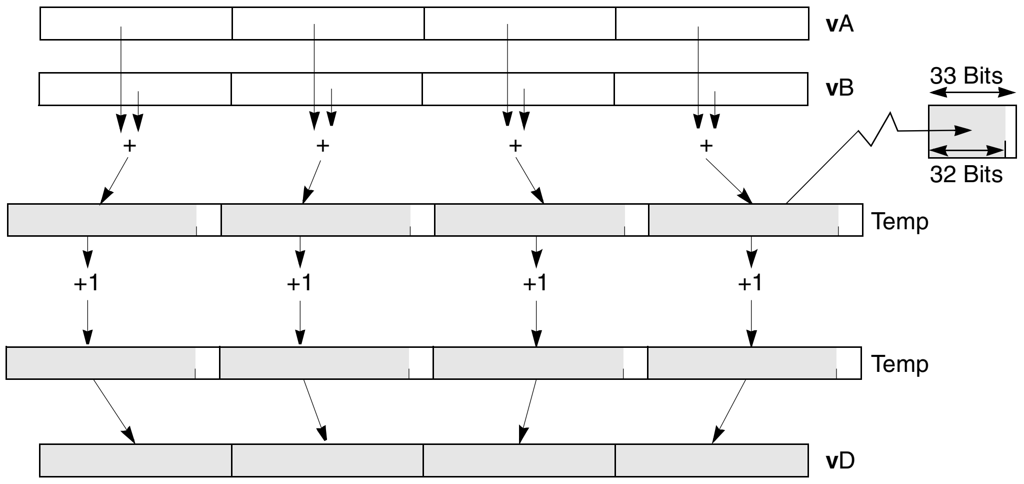


Figure 6-43. vavguw—Average Four Unsigned Integer Elements (32-Bit)

vcfsx vcfsx

Vector Convert from Signed Fixed-Point Word

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
    $vD_{i:i+31} \leftarrow$ CnvtSI32ToFP32($(vB)_{i:i+31}$) $\div_{fp}$ $2^{UIMM}$
end
```

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

Other registers altered:

- None

Figure 6-44 shows the usage of the **vcfsx** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

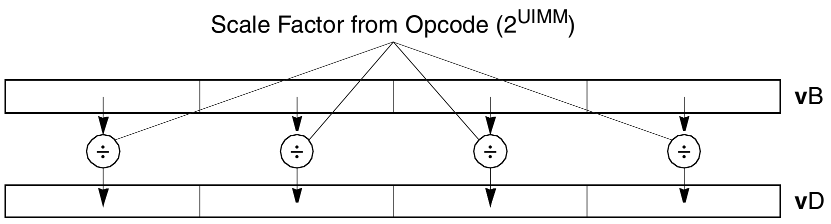


Figure 6-44. vcfsx—Convert Four Signed Integer Elements to Four Floating-Point Elements (32-Bit)

vcfux vcfux

Vector Convert from Unsigned Fixed-Point Word

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

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

Other registers altered:

- None

Figure 6-45 shows the usage of the **vcfux** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

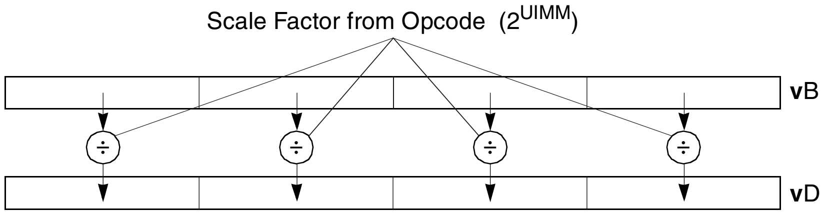


Figure 6-45. vcfux—Convert Four Unsigned Integer Elements to Four Floating-Point Elements (32-Bit)

vcmpbfp*x* vcmpbfp*x*

Vector Compare Bounds Floating-Point

vcmpbfp	**v**D,**v**A,**v**B	(Rc=0)	Form: VXR
vcmpbfp.	**v**D,**v**A,**v**B	(Rc=1)	

04	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)$_{i:i+31}$ $\le_{fp}$ (vB)$_{i:i+31}$)
    ge ← ((vA)$_{i:i+31}$ $\ge_{fp}$ -(vB)$_{i:i+31}$)
    vD$_{i:i+31}$ ← ¬le || ¬ge || $^{30}$0
end
if Rc=1 then do
    ib ← (vD = $^{128}$0)
    CR6 ← 0b00 || ib || 0b0
end
```

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

Bit 0 of the 2-bit value is zero if the element in **v**A is less than or equal to the element in **v**B, and is one otherwise. Bit 1 of the 2-bit value is zero if the element in **v**A is greater than or equal to the negative of the element in **v**B, and is one otherwise.

The 2-bit value is placed into the high-order two bits of the corresponding word element (bits 0–1 for word element 0, bits 32–33 for word element 1, bits 64–65 for word element 2, bits 96–97 for word element 3) of **v**D and the remaining bits of the element are cleared.

If Rc=1, CR Field 6 is set to indicate whether all four elements in **v**A are within the bounds specified by the corresponding element in **v**B, as follows.

- CR6=0b00 || all_within_bounds || 0

Note that if any single-precision floating-point word element in **v**B is negative; the corresponding element in **v**A is out of bounds. Note that if a **v**A or a **v**B element is a NaN, the two high order bits of the corresponding result will both have the value 1.

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

Other registers altered:

- Condition register (CR6):
  Affected: Bit 2 (if Rc=1)

Figure 6-46 shows the usage of the **vcmpbfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

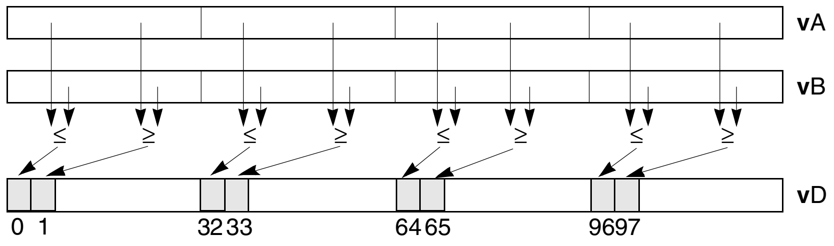


Figure 6-46. vcmpbfp—Compare Bounds of Four Floating-Point Elements (32-Bit)

vcmpeqfp*x* vcmpeqfp*x*

Vector Compare Equal-to-Floating Point

vcmpeqfp vD,vA,vB Form: VXR
vcmpeqfp. vD,vA,vB

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 $vD_{i:i+31} \leftarrow$ 0xFFFF_FFFF
    else $vD_{i:i+31} \leftarrow$ 0x0000_0000
end

if Rc=1 then do
    t ← (vD = $^{128}1$)
    f ← (vD = $^{128}0$)
    CR6 ← t || 0b0 || f || 0b0
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

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

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

Note that if a **v**A or **v**B element is a NaN, the corresponding result will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):
  Affected: Bits 0–3 (if Rc=1)

Figure 6-47 shows the usage of the **vcmpeqfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

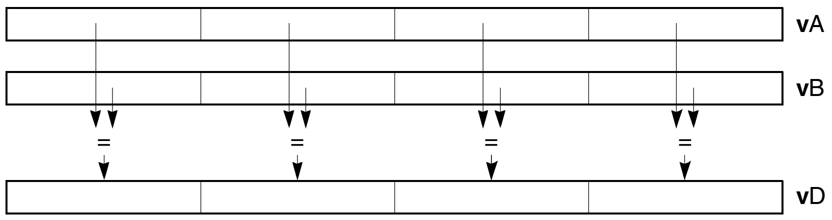


Figure 6-47. vcmpeqfp—Compare Equal of Four Floating-Point Elements (32-Bit)

vcmpequb*x* vcmpequb*x*

Vector Compare Equal-to Unsigned Byte

vcmpequb	vD,vA,vB	Form: VXR
vcmpequb.	vD,vA,vB	

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
        $vD_{i:i+7} \leftarrow {}^{8}1$
    else
        $vD_{i:i+7} \leftarrow {}^{8}0$
end

if Rc=1 then do
    t ← (vD = $^{128}1$)
    f ← (vD = $^{128}0$)
    CR6 ← t || 0b0 || f || 0b0
end
```

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

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

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

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

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

Other registers altered:

- Condition register (CR6):
  Affected: Bits 0–3 (if Rc=1)

Figure 6-48 shows the usage of the **vcmpequb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

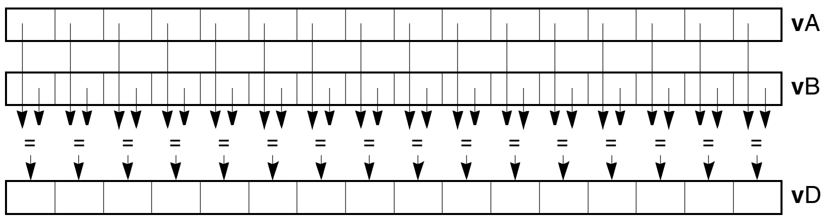


Figure 6-48. vcmpequb—Compare Equal of Sixteen Integer Elements (8-Bit)

vcmpequh*x* vcmpequh*x*

Vector Compare Equal-to Unsigned Half Word

vcmpequh	**v**D,**v**A,**v**B	Form: VXR
vcmpequh.	**v**D,**v**A,**v**B	

04	vD	vA	vB	Rc	70
0 5	6 10	11 15	16 20	21	22 31

```
do i = 0 to 127 by 16
    if $(vA)_{i:i+15} =_{int} (vB)_{i:i+15}$
    then
        $vD_{i:i+15} \leftarrow {}^{16}1$
    else
        $vD_{i:i+15} \leftarrow {}^{16}0$
end

if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \| 0b0 \| f \| 0b0$
end
```

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

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

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

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

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

Other registers altered:

- Condition register (CR6):
  Affected: Bits 0–3 (if Rc=1)

Figure 6-49 shows the usage of the **vcmpequh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

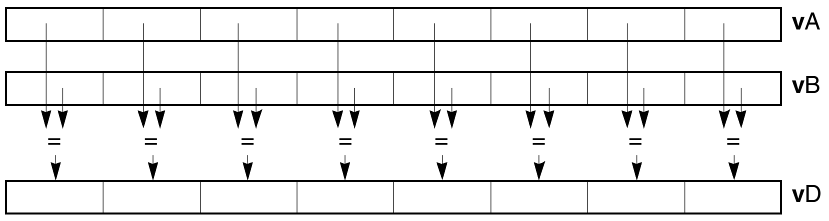


Figure 6-49. vcmpequh—Compare Equal of Eight Integer Elements (16-Bit)

vcmpequw*x* vcmpequw*x*

Vector Compare Equal-to Unsigned Word

vcmpequw	vD,vA,vB	Form: VXR
vcmpequw.	vD,vA,vB	

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 $(vA)_{i:i+311} =_{int} (vB)_{i:i+31}$
    then
        $vD_{i:i+31} \leftarrow {}^{n}1$
    else
        $vD_{i:i+31} \leftarrow {}^{n}0$
end
if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \,||\, 0b0 \,||\, f \,||\, 0b0$
end
```

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

Each integer element in **v**A is compared to the corresponding integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is equal to the element in **v**B, and is cleared to all 0s otherwise.

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

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

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

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-50 shows the usage of the **vcmpequw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

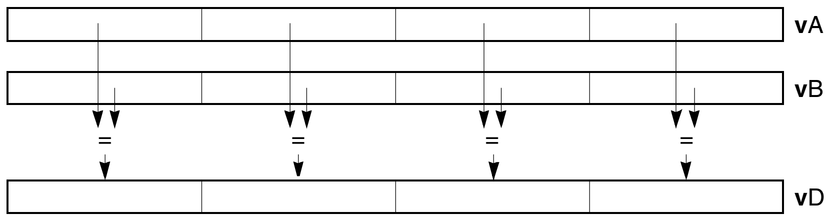


Figure 6-50. vcmpequw—Compare Equal of Four Integer Elements (32-Bit)

vcmpgefp*x* vcmpgefp*x*

Vector Compare Greater-Than-or-Equal-to Floating-Point

vcmpgefp	vD,vA,vB	(Rc=0)	Form: VXR
vcmpgefp.	vD,vA,vB	(Rc=1)	

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 $(vA)_{i:i+31} \geq_{fp} (vB)_{i:i+31}$
    then
        $vD_{i:i+31} \leftarrow$ 0xFFFF_FFFF
    else
        $vD_{i:i+31} \leftarrow$ 0x0000_0000
end

if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \Vert 0b0 \Vert f \Vert 0b0$
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is greater than or equal to the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_or_equal || some_greater_or_equal || none_great_or_equal.

CR6=all_greater_or_equal || 0b0 || none greater_or_equal || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-51 shows the usage of the **vcmpgefp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long

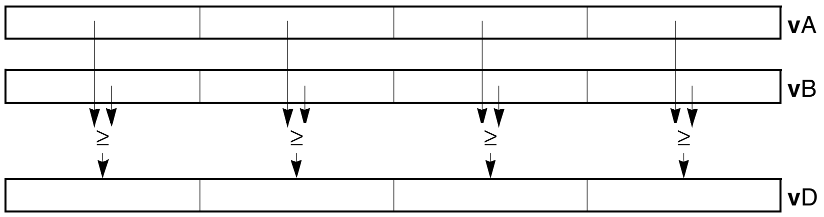


Figure 6-51. vcmpgefp—Compare Greater-Than-or-Equal of Four Floating-Point Elements (32-Bit)

vcmpgtfp*x* vcmpgtfp*x*

Vector Compare Greater-Than Floating-Point

vcmpgtfp	**v**D,**v**A,**v**B	Form: VXR
vcmpgtfp.	**v**D,**v**A,**v**B	

0–5	6–10	11–15	16–20	21	22–31
04	vD	vA	vB	Rc	710

```
do i = 0 to 127 by 32
    if $(vA)_{i:i+31} >_{fp} (vB)_{i:i+31}$
    then
        $vD_{i:i+31} \leftarrow$ 0xFFFF_FFFF
    else
        $vD_{i:i+31} \leftarrow$ 0x0000_0000
end
if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \parallel 0b0 \parallel f \parallel 0b0$
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The corresponding word element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_than || some_greater_than || none_greater_than.

CR6=all_greater_than || 0b0 || none greater_than || 0b0.

Note that if a **v**A or **v**B element is a NaN, the corresponding results will be 0x0000_0000.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-52 shows the usage of the **vcmpgtfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

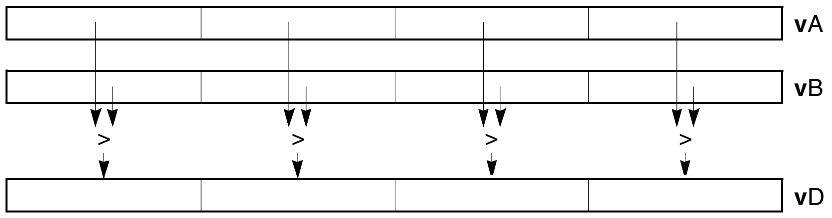


Figure 6-52. vcmpgtfp—Compare Greater-Than of Four Floating-Point Elements (32-Bit)

vcmpgtsb*x* vcmpgtsb*x*

Vector Compare Greater-Than Signed Byte

vcmpgtsb	**v**D,**v**A,**v**B	Form: VXR
vcmpgtsb.	**v**D,**v**A,**v**B	

0–5	6–10	11–15	16–20	21	22–31
04	vD	vA	vB	Rc	774

```
do i = 0 to 127 by 8
    if $(vA)_{i:i+7} >_{si} (vB)_{i:i+7}$
    then
        $vD_{i:i+7} \leftarrow {}^{8}1$
    else
        $vD_{i:i+7} \leftarrow {}^{8}0$
end

if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \| 0b0 \| f \| 0b0$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6=all_greater_than || 0b0 || none greater_than || 0b0.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-53 shows the usage of the **vcmpgtsb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

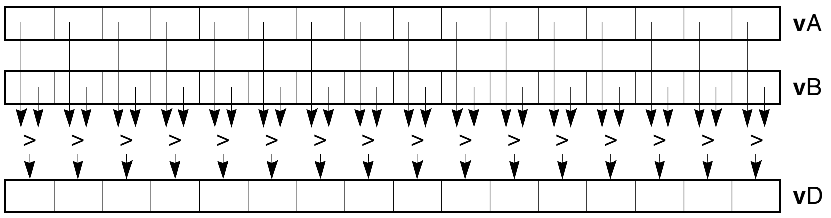


Figure 6-53. vcmpgtsb—Compare Greater-Than of Sixteen Signed Integer Elements (8-Bit)

vcmpgtsh*x* vcmpgtsh*x*

Vector Compare Greater-Than Signed Half Word

vcmpgtsh	**v**D,**v**A,**v**B	Form: VXR
vcmpgtsh.	**v**D,**v**A,**v**B	

0–5	6–10	11–15	16–20	21	22–31
04	vD	vA	vB	Rc	838

```
do i = 0 to 127 by 16
    if $(vA)_{i:i+15} >_{si} (vB)_{i:i+15}$
    then
        $vD_{i:i+15} \leftarrow {}^{16}1$
    else
        $vD_{i:i+15} \leftarrow {}^{16}0$
end

if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \parallel 0b0 \parallel f \parallel 0b0$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6=all_greater_than || 0b0 || none greater_than || 0b0.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-54 shows the usage of the **vcmpgtsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

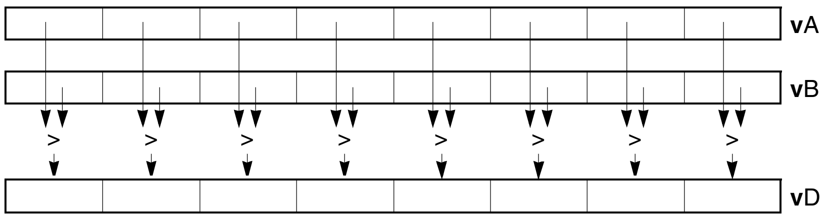


Figure 6-54. vcmpgtsh—Compare Greater-Than of Eight Signed Integer Elements (16-Bit)

vcmpgtsw*x* vcmpgtsw*x*

Vector Compare Greater-Than Signed Word

vcmpgtsw	**v**D,**v**A,**v**B	Form: VXR
vcmpgtsw.	**v**D,**v**A,**v**B	

0–5	6–10	11–15	16–20	21	22–31
04	vD	vA	vB	Rc	902

```
do i = 0 to 127 by 32
    if $(vA)_{i:i+31} >_{si} (vB)_{i:i+31}$
    then
        $vD_{i:i+31} \leftarrow {}^{32}1$
    else
        $vD_{i:i+31} \leftarrow {}^{32}0$
end

if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \,||\, 0b0 \,||\, f \,||\, 0b0$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6=all_greater_than || 0b0 || none greater_than || 0b0.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-55 shows the usage of the **vcmpgtsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

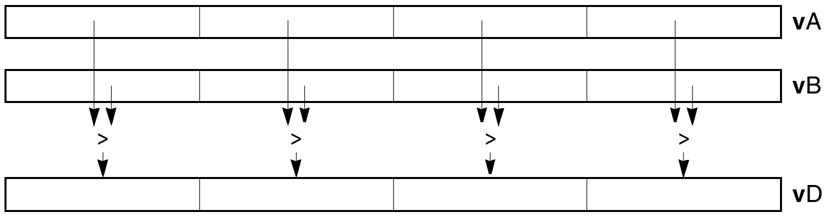


Figure 6-55. vcmpgtsw—Compare Greater-Than of Four Signed Integer Elements (32-Bit)

vcmpgtub*x* vcmpgtub*x*

Vector Compare Greater-Than Unsigned Byte

vcmpgtub	**v**D,**v**A,**v**B	Form: VXR
vcmpgtub.	**v**D,**v**A,**v**B	

0–5	6–10	11–15	16–20	21	22–31
04	vD	vA	vB	Rc	518

```
do i = 0 to 127 by 8
    if $(vA)_{i:i+7} >_{ui} (vB)_{i:i+7}$
    then
        $vD_{i:i+7} \leftarrow {}^{8}1$
    else
        $vD_{i:i+7} \leftarrow {}^{8}0$
end

if Rc=1 then do
    $t \leftarrow (vD = {}^{128}1)$
    $f \leftarrow (vD = {}^{128}0)$
    $CR6 \leftarrow t \,||\, 0b0 \,||\, f \,||\, 0b0$
end
```

Each element of **vcmpgtub** is a byte. Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6=all_greater_than || 0b0 || none greater_than || 0b0.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-56 shows the usage of the **vcmpgtub** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

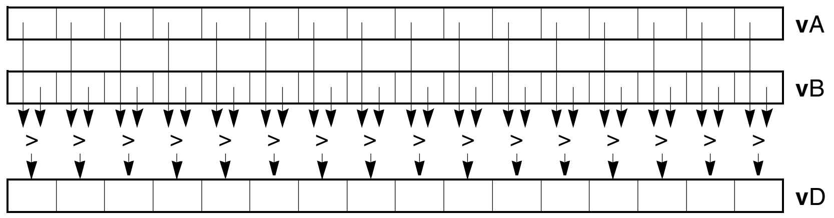


Figure 6-56. vcmpgtub—Compare Greater-Than of Sixteen Unsigned Integer Elements (8-Bit)

vcmpgtuh*x* vcmpgtuh*x*

Vector Compare Greater-Than Unsigned Half Word

vcmpgtuh	**v**D,**v**A,**v**B	Form: VXR
vcmpgtuh.	**v**D,**v**A,**v**B	

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

```
do i = 0 to 127 by 16
    if $(vA)_{i:i+151}$ $>_{ui}$ $(vB)_{i:i+15}$
    then
        $vD_{i:i+15} \leftarrow {}^{16}1$
    else
        $vD_{i:i+15} \leftarrow {}^{16}0$
end

if Rc=1 then do
    t $\leftarrow$ (vD = $^{128}1$)
    f $\leftarrow$ (vD = $^{128}0$)
    CR6 $\leftarrow$ t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtuh** is a half word. Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

CR6=all_greater_than || 0b0 || none greater_than || 0b0.

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-57 shows the usage of the **vcmpgtuh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

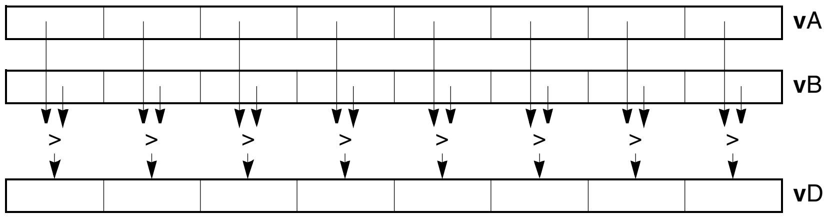


Figure 6-57. vcmpgtuh—Compare Greater-Than of Eight Unsigned Integer Elements (16-Bit)

vcmpgtuw*x* vcmpgtuw*x*

Vector Compare Greater-Than Unsigned Word

vcmpgtuw vD,vA,vB Form: VXR
vcmpgtuw. vD,vA,vB

0–5	6–10	11–15	16–20	21	22–31
04	vD	vA	vB	Rc	646

```
do i = 0 to 127 by 32
    if $(vA)_{i:i+31} >_{ui} (vB)_{i:i+31}$
    then
        $vD_{i:i+31} \leftarrow {}^{32}1$
    else
        $vD_{i:i+31} \leftarrow {}^{32}0$
end

if Rc=1 then do
    t ← (vD = $^{128}1$)
    f ← (vD = $^{128}0$)
    CR6 ← t || 0b0 || f || 0b0
end
```

Each element of **vcmpgtuw** is a word. Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The corresponding element in **v**D is set to all 1s if the element in **v**A is greater than the element in **v**B, and is cleared to all 0s otherwise.

If Rc=1, CR6 is set according to all_greater_than || some_greater_than || none_great_than.

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

Other registers altered:

- Condition register (CR6):

  Affected: Bits 0–3 (if Rc=1)

Figure 6-58 shows the usage of the **vcmpgtuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

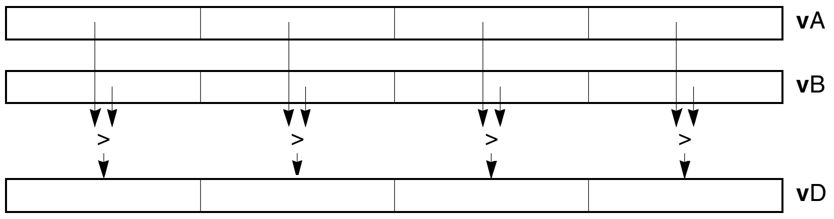


Figure 6-58. vcmpgtuw—Compare Greater-Than of Four Unsigned Integer Elements (32-Bit)

vctsxs

Vector Convert to Signed Fixed-Point Word Saturate

vctsxs vD,vB,UIMM Form: VX

04	vD	UIMM	vB	970
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if (vB)_{i+1:i+8}=255 | (vB)_{i+1:i+8} + UIMM ≤ 254 then
        vD_{i:i+31} ← CnvtFP32ToSI32Sat((vB)_{i:i+31} *_fp 2^UIMM)
     else
        do
        if (vB)_i=0 then vD_{i:i+31} ← 0x7FFF_FFFF
          else vD_{i:i+31} ← 0x8000_0000
          VSCR_SAT ← 1
     end
end
```

Each single-precision word element in **v**B is multiplied by 2^{UIMM}. The product is converted to a signed integer using the rounding mode, Round toward Zero. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1); if it is less than -2^{31} it saturates to -2^{31}. A signed-integer result is placed into the corresponding word element of **v**D.

Fixed-point integers used by the vector convert instructions can be interpreted as consisting of 32-UIMM integer bits followed by UIMM fraction bits. The vector convert to fixed-point word instructions support only the rounding mode, Round toward Zero. A single-precision number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate vector round to floating-point integer instruction before the vector convert to fixed-point word instruction.

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-59 shows the usage of the **vctsxs** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

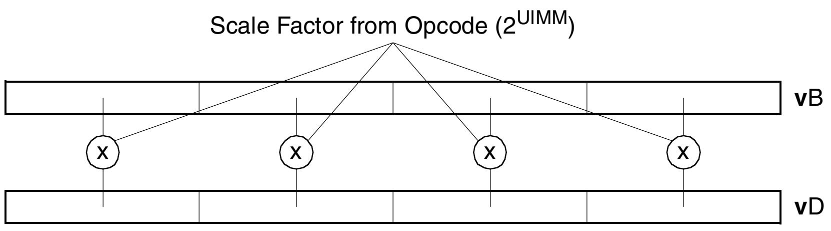


Figure 6-59. vctsxs—Convert Four Floating-Point Elements to Four Signed Integer Elements (32-Bit)

vctuxs vctuxs

Vector Convert to Unsigned Fixed-Point Word Saturate

vctuxs vD,vB,UIMM Form: VX

04	vD	UIMM	vB	906
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
     if (vB)$_{i+1:i+8}$=255 | (vB)$_{i+1:i+8}$ + UIMM ≤ 254 then
         vD$_{i:i+31}$ ← CnvtFP32ToUI32Sat((vB)$_{i:i+31}$ *$_{fp}$ 2$^{UIM}$)
      else
         do
           if (vB)$_i$=0 thenvD$_{i:i+31}$ ← 0xFFFF_FFFF
           elsevD$_{i:i+31}$ ← 0x0000_0000
           VSCR$_{SAT}$ ← 1
      end
end
```

Each single-precision floating-point word element in **v**B is multiplied by 2^{UIM}. The product is converted to an unsigned fixed-point integer using the rounding mode Round toward Zero.

If the intermediate result is greater than $(2^{32}-1)$ it saturates to $(2^{32}-1)$ and if it is less than 0 it saturates to 0.

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

Other registers altered:

- Vector status and control register (VSCR):
  Affected: SAT

Figure 6-60 shows the usage of the **vctuxs** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

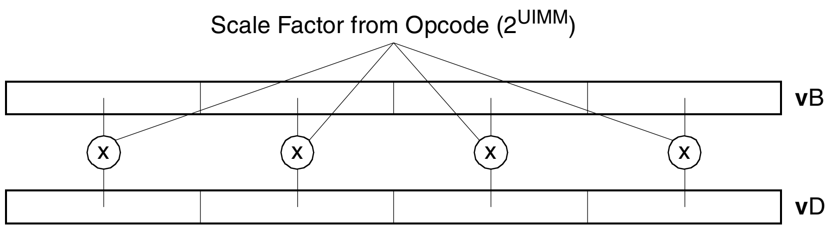


Figure 6-60. vctuxs—Convert Four Floating-Point Elements to Four Unsigned Integer Elements (32-Bit)

vexptefp vexptefp

Vector 2 Raised to the Exponent Estimate Floating-Point

vexptefp **v**D,**v**B Form: VX

04	vD	0_0000	vB	394
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    x ← (vB)$_{i:i+31}$
    vD$_{i:i+31}$ ← $2^x$
end
```

The single-precision floating-point estimate of 2 raised to the power of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

The estimate has a relative error in precision no greater than one part in 16, that is,

$$\left|\frac{\text{estimate} - 2^x}{2^x}\right| \le \frac{1}{16}$$

where *x* is the value of the element in **v**B. The most significant 12 bits of the estimate's significant are monotonic. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

If an operation has an integral value and the resulting value is not 0 or +∞, the result is exact.

Operation with various special values of the element in **v**B is summarized in Table 6-5 below.

Table 6-5. Special Values of the Element in vB

Value of Element in vB	Result
–∞	+0
–0	+1
+0	+1
+∞	+∞
NaN	QNaN

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

Other registers altered:

- None

Figure 6-61 shows the usage of the **vexptefp** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

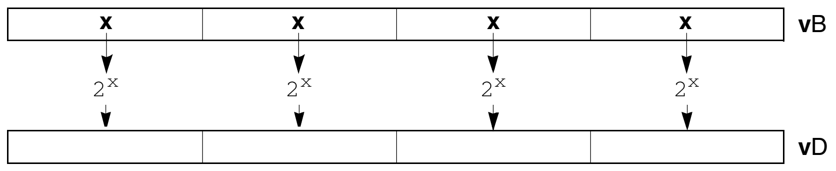


Figure 6-61. vexptefp—Two Raised to the Exponent Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vlogefp vlogefp

Vector Log_2 Estimate Floating-Point

vlogefp **v**D,**v**B Form: VX

04	vD	0_0000	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} ← log_2(x)
end
```

The single-precision floating-point estimate of the base 2 logarithm of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

The estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than 2^{-5}. The estimate has a relative error in precision no greater than one part in 8, as described below:

$$\left(|\text{estimate} - \log_2(x)| \le \frac{1}{32}\right) \quad \text{and} \quad \frac{|\text{estimate} - \log_2(x)|}{\log_2(x)} \le \frac{1}{8}$$

where *x* is the value of the element in **v**B. The most significant 12 bits of the estimate's significant are monotonic. Note that the value placed into the element of **v**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the element in **v**B is summarized below in Table 6-6.

Table 6-6. Special Values of the Element in vB

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 6-62 shows the usage of the **vlogefp** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

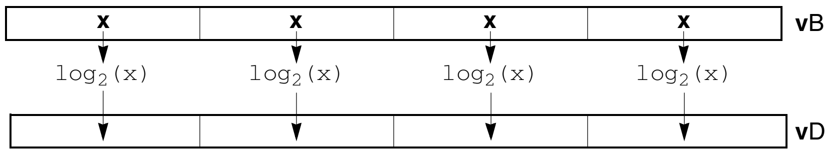


Figure 6-62. vlogefp—Log$_2$ Estimate Floating-Point for Four Floating-Point Elements (32-Bit)

vmaddfp vmaddfp

Vector Multiply Add Floating-Point

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
    $\mathbf{v}D_{i:i+31} \leftarrow$ RndToNearFP32(($\mathbf{v}A)_{i:i+31} *_{fp} (\mathbf{v}C)_{i:i+31}) +_{fp} (\mathbf{v}B)_{i:i+31}$)
end
```

Each single-precision floating-point word element in **v**A is multiplied by the corresponding single-precision floating-point word element in **v**C. The corresponding single-precision floating-point word element in **v**B is added to the product. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

Note that a vector multiply floating-point instruction is not provided. The effect of such an instruction can be obtained by using **vmaddfp** with **v**B containing the value -0.0 (0x8000_0000) in each of its four single-precision floating-point word elements. (The value must be -0.0, not +0.0, in order to obtain the IEEE-conforming result of -0.0 when the result of the multiplication is -0.)

Other registers altered:

- None

If VSCR[NJ]=1, every denormalized operand element is truncated to a 0 of the same sign before the operation is carried out, and each denormalized result element truncates to a 0 of the same sign. Figure 6-63 shows the usage of the **vmaddfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

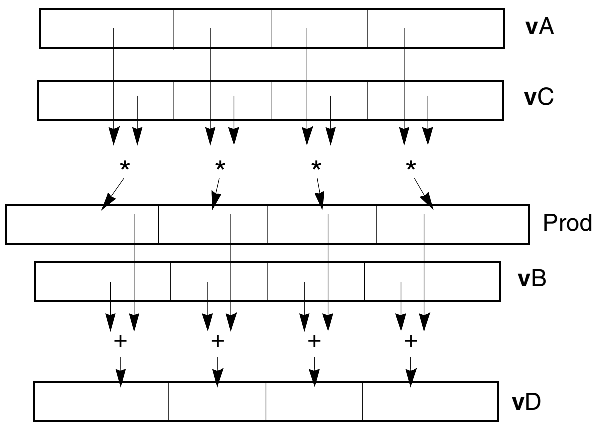


Figure 6-63. vmaddfp—Multiply-Add Four Floating-Point Elements (32-Bit)

vmaxfp vmaxfp

Vector Maximum Floating-Point

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

04	vD	vA	vB	1034
0–5	6–10	11–15	16–20	21–31

```
do i=0 to 127 by 32
    if $(\mathbf{v}A)_{i:i+31} \geq_{fp} (\mathbf{v}B)_{i:i+31}$
        then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
        else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

Each single-precision floating-point word element in **v**A is compared to the corresponding single-precision floating-point word element in **v**B. The larger of the two single-precision floating-point values is placed into the corresponding word element of **v**D.

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

Other registers altered:

- None

Figure 6-64 shows the usage of the **vmaxfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

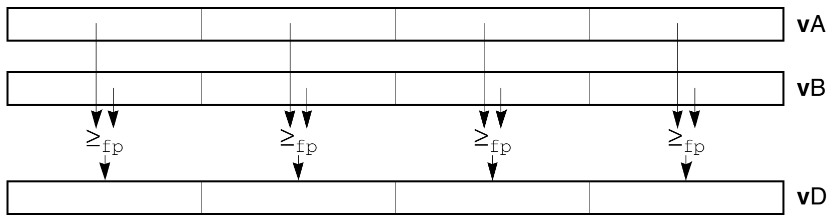


Figure 6-64. vmaxfp—Maximum of Four Floating-Point Elements (32-Bit)

vmaxsb vmaxsb

Vector Maximum Signed Byte

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)$_{i:i+7}$ $\geq_{si}$ (vB)$_{i:i+7}$
        then vD$_{i:i+7}$ $\leftarrow$ (vA)$_{i:i+7}$
        else vD$_{i:i+7}$ $\leftarrow$ (vB)$_{i:i+7}$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-65 shows the usage of the **vmaxsb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

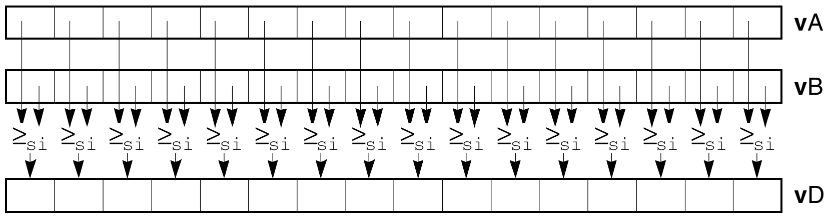


Figure 6-65. vmaxsb—Maximum of Sixteen Signed Integer Elements (8-Bit)

vmaxsh vmaxsh

Vector Maximum Signed Half Word

vmaxsh vD,vA,vB Form: VX

04	vD	vA	vB	322
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    if $(\mathbf{v}A)_{i:i+7} \geq_{si} (\mathbf{v}B)_{i:i+15}$
        then $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{i:i+15}$
        else $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}B)_{i:i+15}$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-66 shows the usage of the **vmaxsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

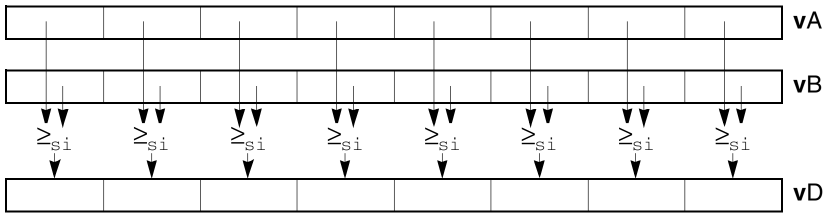


Figure 6-66. vmaxsh—Maximum of Eight Signed Integer Elements (16-Bit)

vmaxsw vmaxsw

Vector Maximum Signed Word

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

04	vD	vA	vB	386
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    if $(\mathbf{v}A)_{i:i+31} \geq_{si} (\mathbf{v}B)_{i:i+31}$
        then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
        else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-67 shows the usage of the **vmaxsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

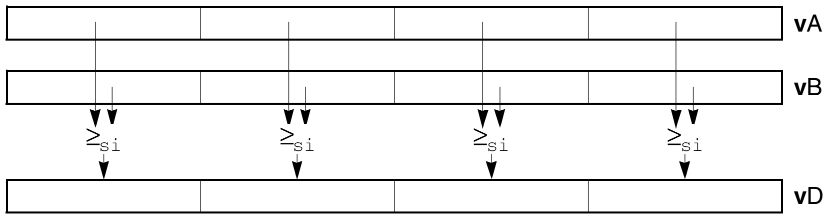


Figure 6-67. vmaxsw—Maximum of Four Signed Integer Elements (32-Bit)

vmaxub vmaxub

Vector Maximum Unsigned Byte

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 $(\mathbf{v}A)_{i:i+7} \geq_{ui} (\mathbf{v}B)_{i:i+7}$
        then $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}A)_{i:i+7}$
        else $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}B)_{i:i+7}$
end
```

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

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-68 shows the usage of the **vmaxub** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

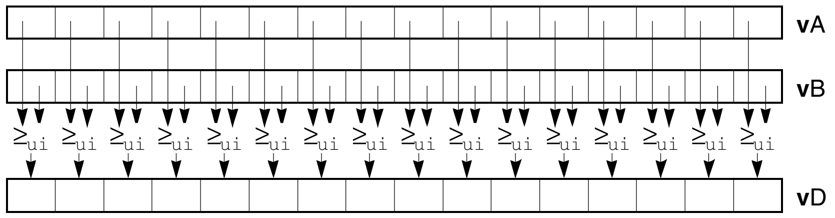


Figure 6-68. vmaxub—Maximum of Sixteen Unsigned Integer Elements (8-Bit)

vmaxuh vmaxuh

Vector Maximum Unsigned Half Word

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 $(\mathbf{v}A)_{i:i+15} \geq_{ui} (\mathbf{v}B)_{i:i+15}$
        then $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{i:i+15}$
        else $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}B)_{i:i+15}$
end
```

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

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-69 shows the usage of the **vmaxuh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

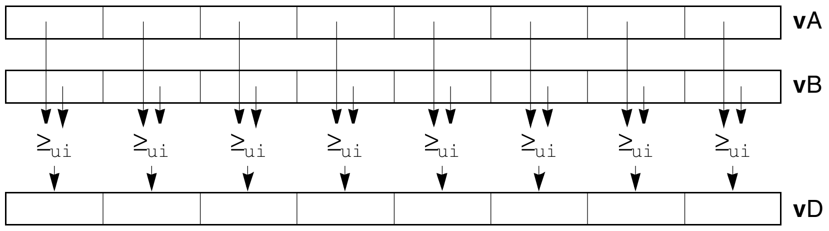


Figure 6-69. vmaxuh—Maximum of Eight Unsigned Integer Elements (16-Bit)

vmaxuw vmaxuw

Vector Maximum Unsigned Word

vmaxuw vD,vA,vB Form: VX

04	vD	vA	vB	130
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
     if $(\mathbf{v}A)_{i:i+31} \geq_{ui} (\mathbf{v}B)_{i:i+31}$
          then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
          else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

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

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-70 shows the usage of the **vmaxuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

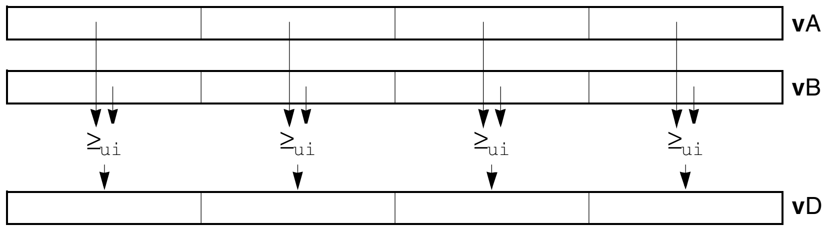


Figure 6-70. vmaxuw—Maximum of Four Unsigned Integer Elements (32-Bit)

vmhaddshs vmhaddshs

Vector Multiply High and Add Signed Half Word Saturate

vmhaddshs vD,vA,vB,vC Form: VA

04	vD	vA	vB	vC	32
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 16
    $prod_{0:31} \leftarrow (\mathbf{v}A)_{i:i+15} *_{si} (\mathbf{v}B)_{i:i+15}$
    $temp_{0:16} \leftarrow prod_{0:16} +_{int} SignExtend((\mathbf{v}C)_{i:i+15},17)$
    $\mathbf{v}D_{i:i+15} \leftarrow SItoSIsat(temp_{0:16},16)$
end
```

Each signed-integer half-word element in **v**A is multiplied by the corresponding signed-integer half-word element in **v**B, producing a 32-bit signed-integer product. Bits 0–16 of the intermediate product are added to the corresponding signed-integer half-word element in **v**C after they have been sign extended to 17 bits. The 16-bit saturated result from each of the eight 17-bit sums is placed in register **v**D.

If the intermediate result is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than (-2^{15}) it saturates to (-2^{15}).

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

Other registers altered:

- Vector status and control register (VSCR):

  Affected: SAT

Figure 6-71 shows the usage of the **vmhaddshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 16 bits long.

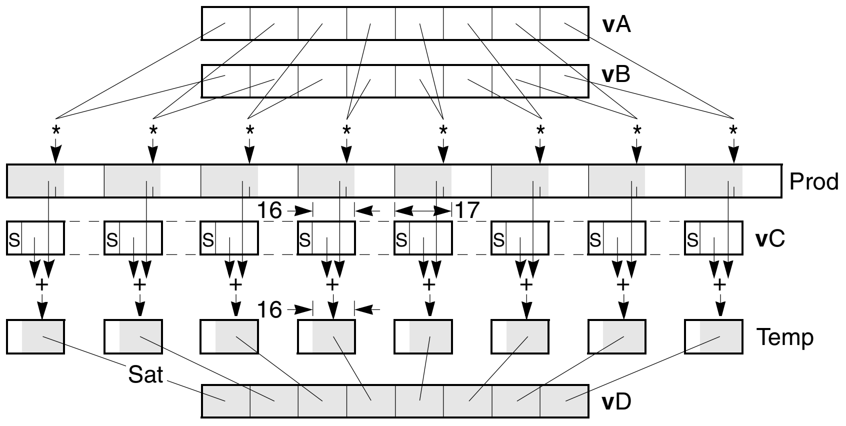


Figure 6-71. vmhaddshs—Multiply-High and Add Eight Signed Integer Elements (16-Bit)

vmhraddshs vmhraddshs

Vector Multiply High Round and Add Signed Half Word Saturate

vmhraddshs **v**D,**v**A,**v**B,**v**C Form: VA

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

```
do i=0 to 127 by 16
    $prod_{0:31} \leftarrow (vA)_{i:i+15} *_{si} (vB)_{i:i+15}$
    $prod_{0:31} \leftarrow prod_{0:31} +_{int}$ 0x0000_4000
    $temp_{0:16} \leftarrow prod_{0:16} +_{int}$ SignExtend($(vC)_{i:i+15}$,17)
    $(vD)_{i:i+15} \leftarrow$ SItoSIsat($temp_{0:16}$,16)
end
```

Each signed integer half-word element in register **v**A is multiplied by the corresponding signed integer half-word element in register **v**B, producing a 32-bit signed integer product. The value 0x0000_4000 is added to the product, producing a 32-bit signed integer sum. Bits 0–16 of the sum are added to the corresponding signed integer half-word element in register **v**D.

If the intermediate result is greater than $(2^{15}-1)$ it saturates to $(2^{15}-1)$ and if it is less than (-2^{15}) it saturates to (-2^{15}).

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

Figure 6-72 shows the usage of the **vmhraddshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 16 bits long.

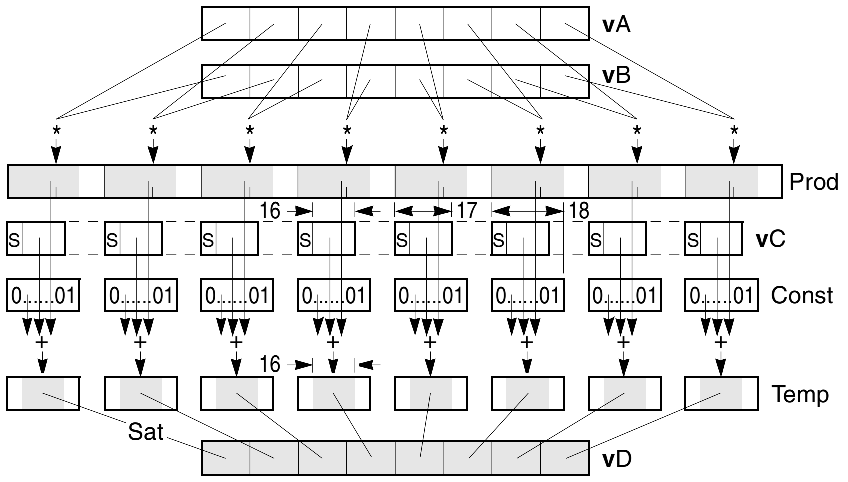


Figure 6-72. vmhraddshs—Multiply-High Round and Add Eight Signed Integer Elements (16-Bit)

vminfp vminfp

Vector Minimum Floating-Point

vminfp vD,vA,vB Form: VX

04	vD	vA	vB	1098
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
     if $(\mathbf{v}A)_{i:i+31} <_{fp} (\mathbf{v}B)_{i:i+31}$
         then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
         else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

Each single-precision floating-point word element in register **v**A is compared to the corresponding single-precision floating-point word element in register **v**B. The smaller of the two single-precision floating-point values is placed into the corresponding word element of register **v**D.

The minimum of +0.0 and -0.0 is -0.0. The minimum of any value and a NaN is a QNaN.

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

Figure 6-73 shows the usage of the **vminfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

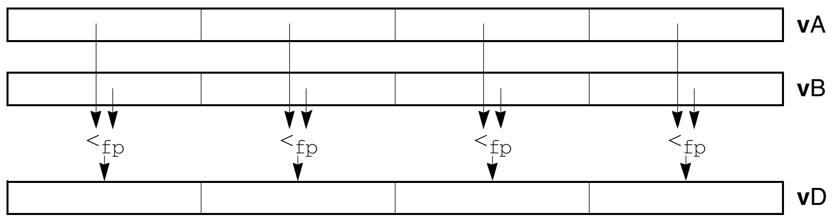


Figure 6-73. vminfp—Minimum of Four Floating-Point Elements (32-Bit)

vminsb vminsb

Vector Minimum Signed Byte

vminsb vD,vA,vB Form: VX

04	vD	vA	vB	770
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
     if $(\mathbf{v}A)_{i:i+7} <_{si} (\mathbf{v}B)_{i:i+7}$
          then $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}A)_{i:i+7}$
          else $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}B)_{i:i+7}$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-74 shows the usage of the **vminsb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

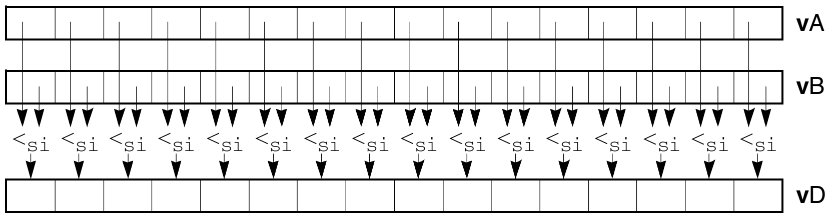


Figure 6-74. vminsb—Minimum of Sixteen Signed Integer Elements (8-Bit)

vminsh vminsh

Vector Minimum Signed Half Word

vminsh vD,vA,vB 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 $(\mathbf{v}A)_{i:i+15} <_{si} (\mathbf{v}B)_{i:i+15}$
          then $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{i:i+15}$
          else $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}B)_{i:i+15}$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-75 shows the usage of the **vminsh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

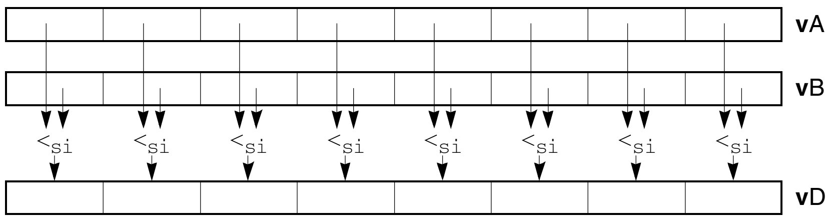


Figure 6-75. vminsh—Minimum of Eight Signed Integer Elements (16-Bit)

vminsw vminsw

Vector Minimum Signed Word

vminsw 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 $(\mathbf{v}A)_{i:i+31} <_{si} (\mathbf{v}B)_{i:i+31}$
        then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
        else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

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

Each signed-integer element in **v**A is compared to the corresponding signed-integer element in **v**B. The larger of the two signed-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-76 shows the usage of the **vminsw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

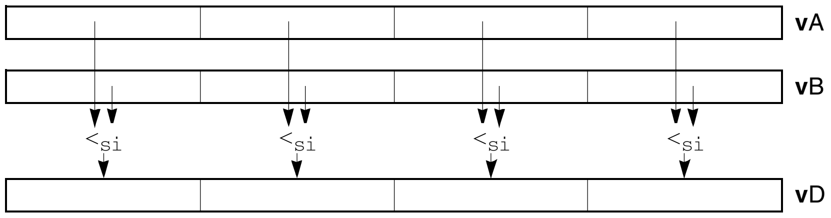


Figure 6-76. vminsw—Minimum of Four Signed Integer Elements (32-Bit)

vminub vminub

Vector Minimum Unsigned Byte

vminub vD,vA,vB Form: VX

04	vD	vA	vB	514
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    if (vA)$_{i:i+7}$ <ui (vB)$_{i:i+7}$
        then vD$_{i:i+7}$ ← (vA)$_{i:i+7}$
        else vD$_{i:i+7}$ ← (vB)$_{i:i+7}$
end
```

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

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-77 shows the usage of the **vminub** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

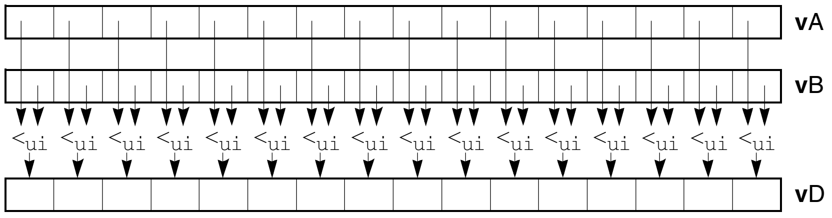

vA
vB
<ui <ui <ui <ui <ui <ui <ui <ui <ui <ui <ui <ui <ui <ui <ui <ui
vD

Figure 6-77. vminub—Minimum of Sixteen Unsigned Integer Elements (8-Bit)

vminuh vminuh

Vector Minimum Unsigned Half Word

vminuh vD,vA,vB Form: VX

04	vD	vA	vB	578
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    if (vA)$_{i:i+15}$ $<_{ui}$ (vB)$_{i:i+15}$
        then vD$_{i:i+15}$ $\leftarrow$ (vA)$_{i:i+15}$
        else vD$_{i:i+15}$ $\leftarrow$ (vB)$_{i:i+15}$
end
```

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

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-78 shows the usage of the **vminuh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

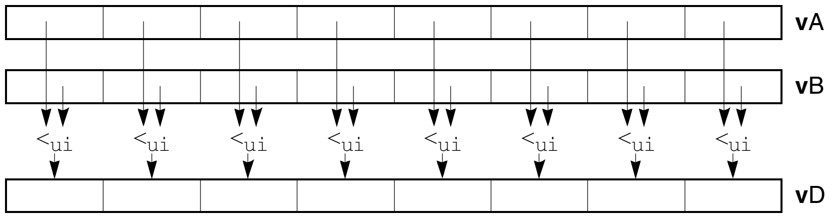


Figure 6-78. vminuh—Minimum of Eight Unsigned Integer Elements (16-Bit)

vminuw   vminuw

Vector Minimum Unsigned Word

vminuw vD,vA,vB Form: VX

04	vD	vA	vB	642
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
     if $(\mathbf{v}A)_{i:i+31} <_{ui} (\mathbf{v}B)_{i:i+31}$
          then $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}A)_{i:i+31}$
          else $\mathbf{v}D_{i:i+31} \leftarrow (\mathbf{v}B)_{i:i+31}$
end
```

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

Each unsigned-integer element in **v**A is compared to the corresponding unsigned-integer element in **v**B. The larger of the two unsigned-integer values is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-79 shows the usage of the **vminuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

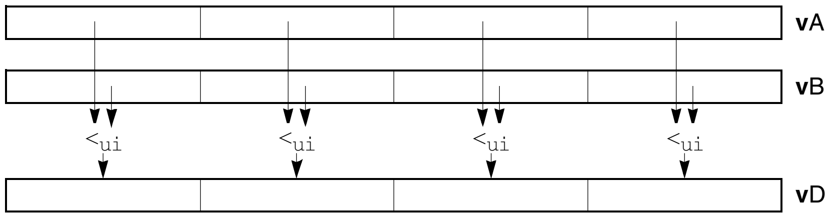


Figure 6-79. vminuw—Minimum of Four Unsigned Integer Elements (32-Bit)

vmladduhm vmladduhm

Vector Multiply Low and Add Unsigned Half Word Modulo

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
    $prod_{0:31}$ ← $(\mathbf{v}A)_{i:i+15}$ $*_{ui}$ $(\mathbf{v}B)_{i:i+15}$
    $\mathbf{v}D_{i:i+15}$ ← $prod_{0:31}$ $+_{int}$ $(\mathbf{v}C)_{i:i+15}$
end
```

Each integer half-word element in **v**A is multiplied by the corresponding integer half-word element in **v**B, producing a 32-bit integer product. The product is added to the corresponding integer half-word element in **v**C. The integer result is placed into the corresponding half-word element of **v**D.

Note that **vmladduhm** can be used for unsigned or signed integers.

Other registers altered:

- None

Figure 6-80 shows the usage of the **vmladduhm** instruction. Each of the eight elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 16 bits long.

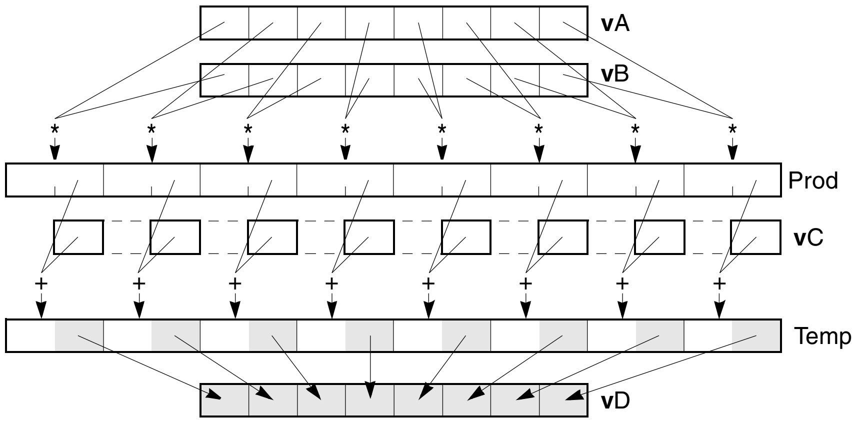


Figure 6-80. vmladduhm—Multiply-Add of Eight Integer Elements (16-Bit)

vmrghb vmrghb

Vector Merge High Byte

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
```

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

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-81 shows the usage of the **vmrghb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

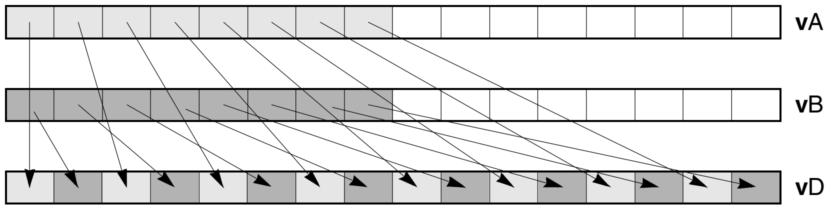


Figure 6-81. vmrghb—Merge Eight High-Order Elements (8-Bit)

vmrghh vmrghh

Vector Merge High Half word

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
    $vD_{i*2:(i*2)+31} \leftarrow (vA)_{i:i+15} \| (vB)_{i:i+15}$
end
```

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

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-82 shows the usage of the **vmrghh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

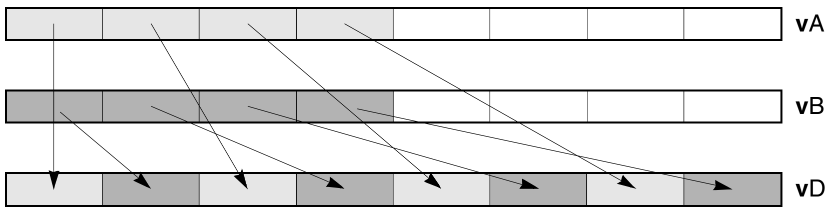


Figure 6-82. vmrghh—Merge Four High-Order Elements (16-Bit)

vmrghw　　vmrghw

Vector Merge High Word

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
    $\mathbf{v}D_{i*2:(i*2)+63} \leftarrow (\mathbf{v}A)_{i:i+31} \parallel (\mathbf{v}B)_{i:i+31}$
end
```

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

The elements in the high-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the high-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-83 shows the usage of the **vmrghw** instruction. Each of the two elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

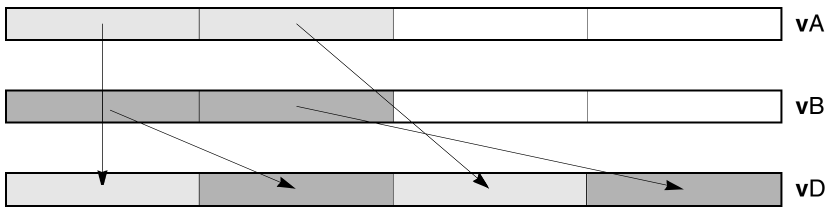


Figure 6-83. vmrghw—Merge Two High-Order Elements (32-Bit)

vmrglb vmrglb

Vector Merge Low Byte

vmrglb vD,vA,vB Form: VX

04	vD	vA	vB	268
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i*2:(i*2)+15} ← (vA)_{i+64:i+71} || (vB)_{i+64:i+71}
end
```

Each element offer **vmrglb** is a byte.

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-84 shows the usage of the **vmrglb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

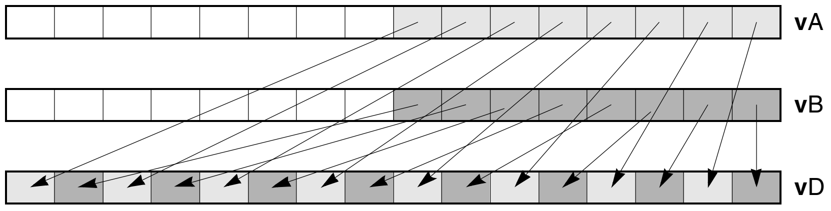


Figure 6-84. vmrglb—Merge Eight Low-Order Elements (8-Bit)

vmrglh vmrglh

Vector Merge Low Half Word

vmrglh vD,vA,vB Form: VX

04	vD	vA	vB	332
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+64:i+79} || (vB)_{i+64:i+79}
end
```

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

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-85 shows the usage of the **vmrglh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

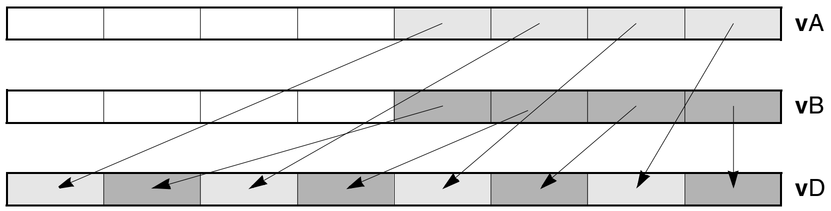


Figure 6-85. vmrglh—Merge Four Low-Order Elements (16-Bit)

vmrglw vmrglw

Vector Merge Low Word

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

04	vD	vA	vB	396
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 32
    $\mathbf{v}D_{i*2:(i*2)+63} \leftarrow (\mathbf{v}A)_{i+64:i+95} \parallel (\mathbf{v}B)_{i+64:i+95}$
end
```

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

The elements in the low-order half of **v**A are placed, in the same order, into the even-numbered elements of **v**D. The elements in the low-order half of **v**B are placed, in the same order, into the odd-numbered elements of **v**D.

Other registers altered:

- None

Figure 6-86 shows the usage of the **vmrglw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

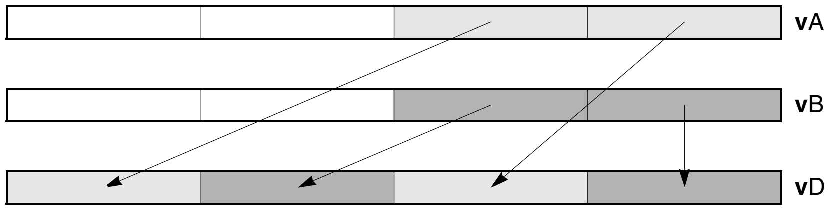


Figure 6-86. vmrglw—Merge Four Low-Order Elements (32-Bit)

vmsummbm vmsummbm

Vector Multiply Sum Mixed-Sign Byte Modulo

vmsummbm **v**D,**v**A,**v**B,**v**C Form: VA

04	vD	vA	vB	vC	37
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
     $temp_{0:31}$ ← $(vC)_{i:i+31}$
      do j=0 to 31 by 8
          $prod_{0:15}$ ← $(vA)_{i+j:i+j+7}$ $*_{sui}$ $(vB)_{i+j:i+j+7}$
          $temp_{0:31}$ ← $temp_{0:31}$ $+_{int}$ SignExtend($prod_{0:15}$,32)
          end
     $vD_{i:i+31}$ ← $temp_{0:31}$
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the four signed-integer byte elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B, producing a signed-integer 16-bit product.
- The signed-integer modulo sum of these four products is added to the signed-integer word element in **v**C.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-87 shows the usage of the **vmsummbm** instruction. Each of the sixteen elements in the vectors, **v**A and **v**B, are 8 bits long. Each of the four elements in the vectors, **v**C and **v**D, are 32 bits long.

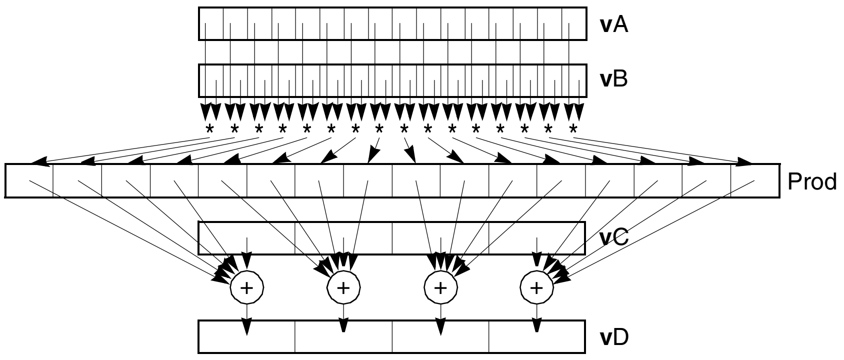


Figure 6-87. vmsummbm—Multiply-Sum of Integer Elements (8- to 32-Bit)

vmsumshm vmsumshm

Vector Multiply Sum Signed Half Word Modulo

vmsumshm **v**D,**v**A,**v**B,**v**C 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
    $temp_{0:31} \leftarrow (\mathbf{v}C)_{i:i+31}$
     do j=0 to 31 by 16
        $prod_{0:31} \leftarrow (\mathbf{v}A)_{i+j:i+j+15} *_{si} (\mathbf{v}B)_{i+j:i+j+15}$
        $temp_{0:31} \leftarrow temp_{0:31} +_{int} prod_{0:31}$
        $\mathbf{v}D_{i:i+31} \leftarrow temp_{0:31}$
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two signed-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding signed-integer half-word element in **v**B, producing a signed-integer 32-bit product.
- The signed-integer modulo sum of these two products is added to the signed-integer word element in **v**C.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-88 shows the usage of the **vmsumshm** instruction. Each of the eight elements in the vectors, **v**A and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D, are 32 bits long.

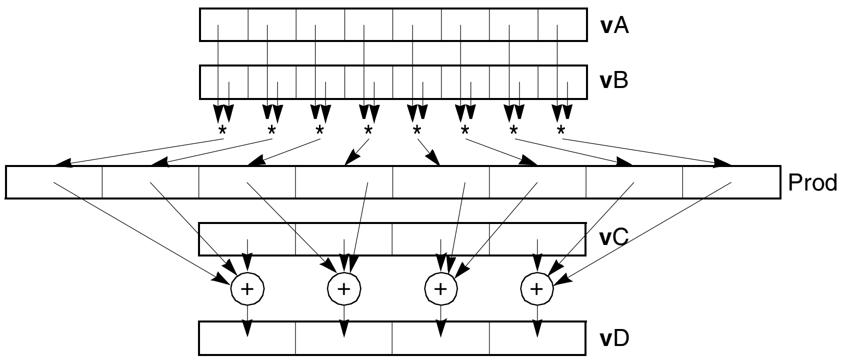


Figure 6-88. vmsumshm—Multiply-Sum of Signed Integer Elements (16- to 32-Bit)

vmsumshs   vmsumshs

Vector Multiply Sum Signed Half Word Saturate

vmsumshs **v**D,**v**A,**v**B,**v**C Form: VA

04	vD	vA	vB	vC	41
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    $temp_{0:33}$ ← SignExtend(($\mathbf{v}$C)$_{i:i+31}$,34)
     do j=0 to 31 by 16
        $prod_{0:31}$ ← ($\mathbf{v}$A)$_{i+j:i+j+15}$ $*_{si}$ ($\mathbf{v}$B)$_{i+j:i+j+15}$
        $temp_{0:33}$ ← $temp_{0:33}$ $+_{int}$ SignExtend($prod_{0:31}$,34)
        $\mathbf{v}D_{i:i+31}$ ← SItoSIsat($temp_{0:33}$,32)
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two signed-integer half-word elements in the corresponding word element of **v**A is multiplied by the corresponding signed-integer half-word element in **v**B, producing a signed-integer 32-bit product.
- The signed-integer sum of these two products is added to the signed-integer word element in **v**C.
- If this intermediate result is greater than $(2^{31}-1)$ it saturates to $(2^{31}-1)$ and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-89 shows the usage of the **vmsumshs** instruction. Each of the eight elements in the vectors, **v**A and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D, are 32 bits long.

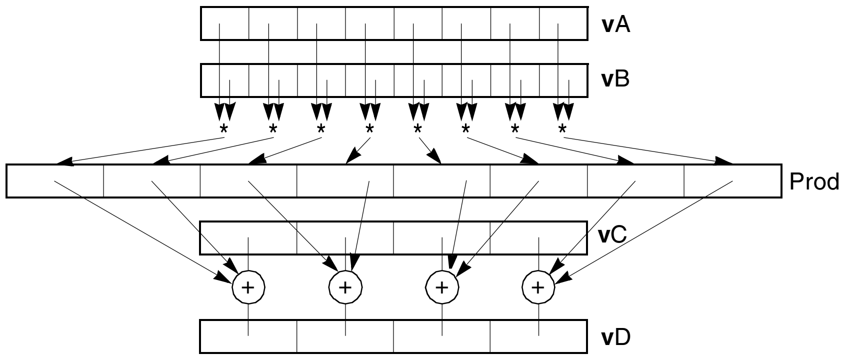


Figure 6-89. vmsumshs—Multiply-Sum of Signed Integer Elements (16- to 32-Bit)

vmsumubm vmsumubm

Vector Multiply Sum Unsigned Byte Modulo

vmsumubm **v**D,**v**A,**v**B,**v**C Form: VA

04	vD	vA	vB	vC	36
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    temp$_{0:31}$ ← (vC)$_{i:i+31}$
     do j=0 to 31 by 8
        prod$_{0:15}$ ← (vA)$_{i+j:i+j+7}$ *$_{ui}$ (vB)$_{i+j:i+j+7}$
        temp$_{0:32}$ ← temp$_{0:32}$ +$_{int}$ ZeroExtend(prod$_{0:15}$,32)
        vD$_{i:i+31}$ ← temp$_{0:31}$
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the four unsigned-integer byte elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B, producing an unsigned-integer 16-bit product.
- The unsigned-integer modulo sum of these four products is added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-90 shows the usage of the **vmsumubm** instruction. Each of the sixteen elements in the vectors, **v**A and **v**B, are 8 bits long. Each of the four elements in the vectors, **v**C and **v**D, are 32 bits long.

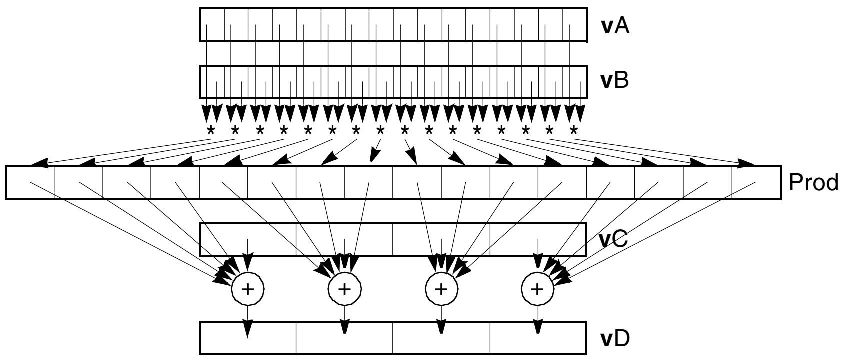


Figure 6-90. vmsumubm—Multiply-Sum of Unsigned Integer Elements (8- to 32-Bit)

vmsumuhm vmsumuhm

Vector Multiply Sum Unsigned Half Word Modulo

vmsumuhm vD,vA,vB,vC Form: VA

04	vD	vA	vB	vC	38
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    $temp_{0:31} \leftarrow (\mathbf{v}C)_{i:i+31}$
     do j=0 to 31 by 16
        $prod_{0:31} \leftarrow (\mathbf{v}A)_{i+j:i+j+15} *_{ui} (\mathbf{v}B)_{i+j:i+j+15}$
        $temp_{0:31} \leftarrow temp_{0:31} +_{int} prod_{0:31}$
        $\mathbf{v}D_{i:i+31} \leftarrow temp_{2:33}$
    end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two unsigned-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B, producing a unsigned-integer 32-bit product.
- The unsigned-integer sum of these two products is added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-91 shows the usage of the **vmsumuhm** instruction. Each of the eight elements in the vectors, **v**A and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D, are 32 bits long.

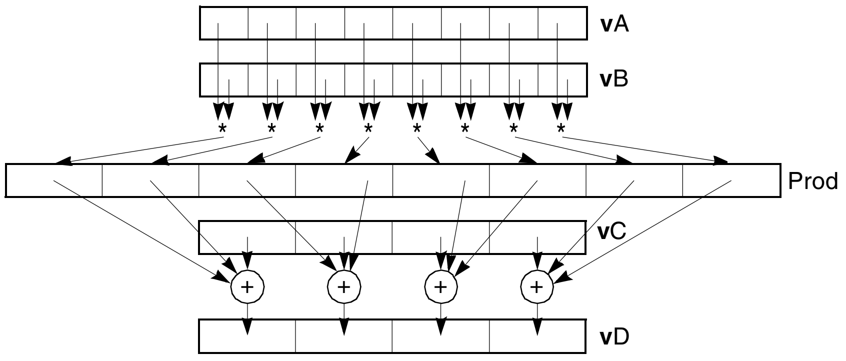


Figure 6-91. vmsumuhm—Multiply-Sum of Unsigned Integer Elements (16- to 32-Bit)

vmsumuhs　　　　vmsumuhs

Vector Multiply Sum Unsigned Half Word Saturate

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
     temp$_{0:33}$ ← ZeroExtend((vC)$_{i:i+31}$,34)
      do j=0 to 31 by 16
         prod$_{0:31}$ ← (vA)$_{i+j:i+j+15}$ *$_{ui}$ (vB)$_{i+j:i+j+15}$
         temp$_{0:33}$ ← temp$_{0:33}$ +$_{int}$ ZeroExtend(prod$_{0:31}$,34)
         vD$_{i:i+31}$ ← UItoUIsat(temp$_{0:33}$,32)
     end
end
```

For each word element in **v**C the following operations are performed in the order shown.

- Each of the two unsigned-integer half-word elements contained in the corresponding word element of **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B, producing an unsigned-integer 32-bit product.
- The unsigned-integer sum of these two products is saturate-added to the unsigned-integer word element in **v**C.
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-92 shows the usage of the **vmsumuhs** instruction. Each of the eight elements in the vectors, **v**A and **v**B, are 16 bits long. Each of the four elements in the vectors, **v**C and **v**D, are 32 bits long.

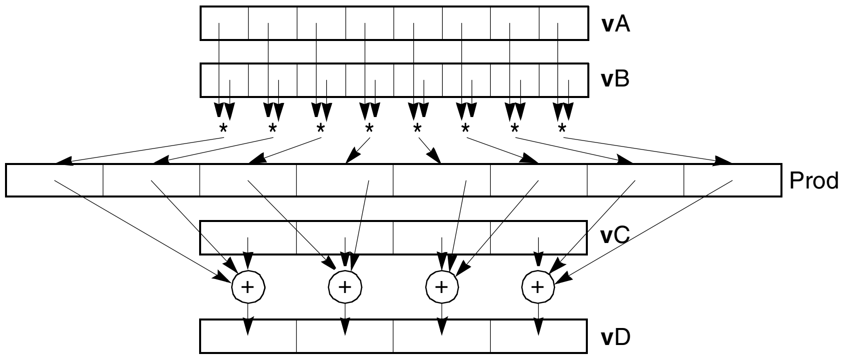


Figure 6-92. vmsumuhs—Multiply-Sum of Unsigned Integer Elements (16- to 32-Bit)

vmulesb vmulesb

Vector Multiply Even Signed Byte

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
    $prod_{0:15} \leftarrow (\mathbf{v}A)_{i:i+7} *_{si} (\mathbf{v}B)_{i:i+7}$
     $\mathbf{v}D_{i:i+15} \leftarrow prod_{0:15}$
end
```

Each even-numbered signed-integer byte element in **v**A is multiplied by the corresponding signed-integer byte element in **v**B. The eight 16-bit signed-integer products are placed, in the same order, into the eight half words of **v**D.

Other registers altered:

- None

Figure 6-93 shows the usage of the **vmulesb** instruction. Each of the sixteen elements in the vectors, **v**A and **v**B, is 8 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

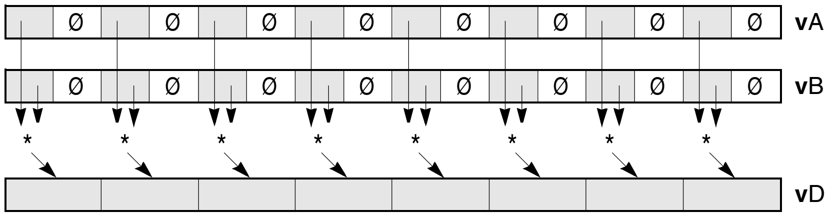


Figure 6-93. vmulesb—Even Multiply of Eight Signed Integer Elements (8-Bit)

vmulesh vmulesh

Vector Multiply Even Signed Half Word

vmulesh vD,vA,vB Form: VX

04	vD	vA	vB	840
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $prod_{0:31} \leftarrow (\mathbf{v}A)_{i:i+15} *_{si} (\mathbf{v}B)_{i:i+15}$
    $\mathbf{v}D_{i:i+31} \leftarrow prod_{0:31}$
end
```

Each even-numbered signed-integer half-word element in **v**A is multiplied by the corresponding signed-integer half-word element in **v**B. The four 32-bit signed-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-94 shows the usage of the **vmulesh** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the four elements in the vector, **v**D, is 32 bits long.

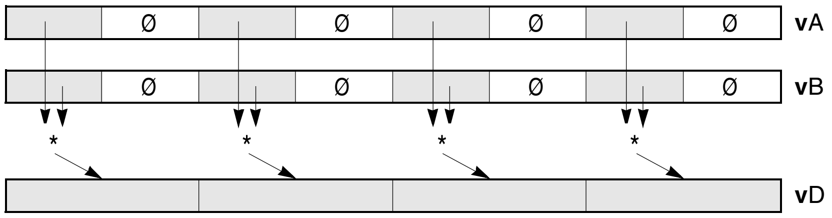


Figure 6-94. vmulesh—Even Multiply of Four Signed Integer Elements (16-Bit)

vmuleub vmuleub

Vector Multiply Even Unsigned Byte

vmuleub vD,vA,vB Form: VX

04	vD	vA	vB	520
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
     prod$_{0:15}$ ← (vA)$_{i:i+7}$ *$_{ui}$ (vB)$_{i:i+7}$
     (vD)$_{i:i+15}$ ← prod$_{0:15}$
end
```

Each even-numbered unsigned-integer byte element in register **v**A is multiplied by the corresponding unsigned-integer byte element in register **v**B. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight half words of register **v**D.

Other registers altered:

- None

Figure 6-95 shows the usage of the **vmuleub** instruction. Each of the sixteen elements in the vectors, **v**A and **v**B, is 8 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

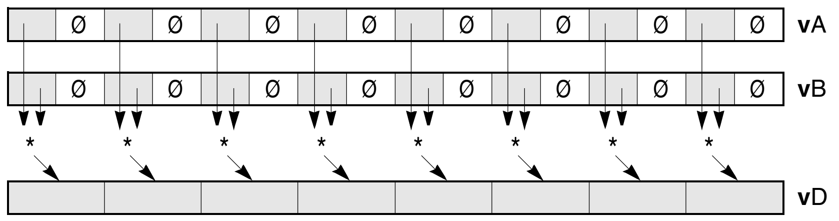


Figure 6-95. vmuleub—Even Multiply of Eight Unsigned Integer Elements (8-Bit)

vmuleuh vmuleuh

Vector Multiply Even Unsigned Half Word

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

04	vD	vA	vB	584
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $prod_{0:31} \leftarrow (vA)_{i:i+15} *_{ui} (vB)_{i:i+15}$
    $(vD)_{i:i+31} \leftarrow prod_{0:31}$
end
```

Each even-numbered unsigned-integer half-word element in register **v**A is multiplied by the corresponding unsigned-integer half-word element in register **v**B. The four 32-bit unsigned-integer products are placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-96 shows the usage of the **vmuleuh** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the four elements in the vector, **v**D, is 32 bits long.

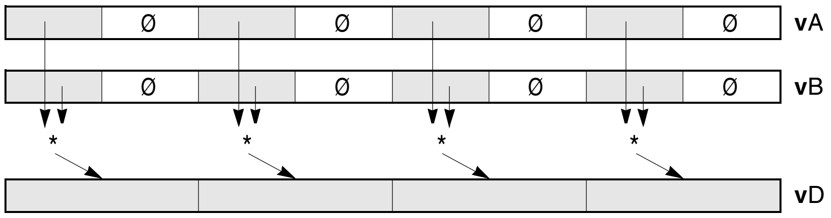


Figure 6-96. vmuleuh—Even Multiply of Four Unsigned Integer Elements (16-Bit)

vmulosb vmulosb

Vector Multiply Odd Signed Byte

vmulosb vD,vA,vB Form: VX

04	vD	vA	vB	264
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    $prod_{0:15} \leftarrow (\mathbf{v}A)_{i+8:i+15} *_{si} (\mathbf{v}B)_{i+8:i+15}$
    $\mathbf{v}D_{i:i+15} \leftarrow prod_{0:15}$
end
```

Each odd-numbered signed-integer byte element in **v**A is multiplied by the corresponding signed-integer byte element in **v**B. The eight 16-bit signed-integer products are placed, in the same order, into the eight half words of **v**D.

Other registers altered:

- None

Figure 6-97 shows the usage of the **vmulosb** instruction. Each of the sixteen elements in the vectors, **v**A and **v**B, is 8 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

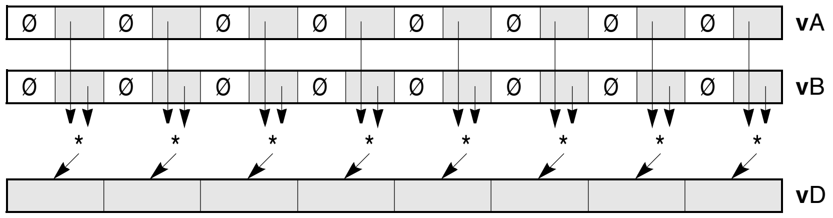


Figure 6-97. vmulosb—Odd Multiply of Eight Signed Integer Elements (8-Bit)

vmulosh vmulosh

Vector Multiply Odd Signed Half Word

vmulosh vD,vA,vB Form: VX

04	vD	vA	vB	328
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $prod_{0:31} \leftarrow (\mathbf{v}A)_{i+16:i+31} *_{si} (\mathbf{v}B)_{i+16:i+31}$
    $\mathbf{v}D_{i:i+31} \leftarrow prod_{0:31}$
end
```

Each odd-numbered signed-integer half-word element in **v**A is multiplied by the corresponding signed-integer half-word element in **v**B. The four 32-bit signed-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-98 shows the usage of the **vmuleuh** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the four elements in the vector, **v**D, is 32 bits long.

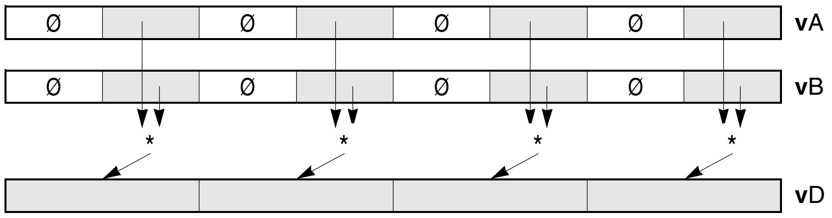


Figure 6-98. vmuleuh—Odd Multiply of Four Unsigned Integer Elements (16-Bit)

vmuloub vmuloub

Vector Multiply Odd Unsigned Byte

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

04	vD	vA	vB	8
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    $prod_{0:15} \leftarrow (vA)_{i+8:i+15} *_{ui} (vB)_{i+n:i+15}$
    $vD_{i:i+15} \leftarrow prod_{0:15}$
end
```

Each odd-numbered unsigned-integer byte element in **v**A is multiplied by the corresponding unsigned-integer byte element in **v**B. The eight 16-bit unsigned-integer products are placed, in the same order, into the eight half words of **v**D.

Other registers altered:

- None

Figure 6-99 shows the usage of the **vmuloub** instruction. Each of the sixteen elements in the vectors, **v**A and **v**B, is 8 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

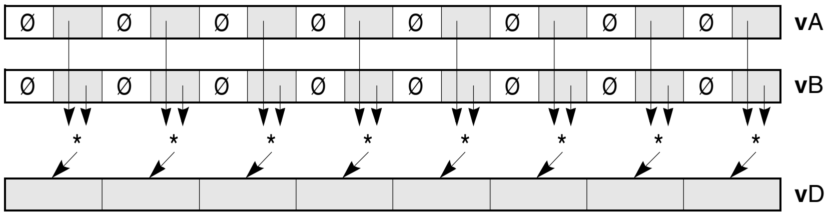


Figure 6-99. vmuloub—Odd Multiply of Eight Unsigned Integer Elements (8-Bit)

vmulouh vmulouh

Vector Multiply Odd Unsigned Half Word

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

04	vD	vA	vB	72
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
     $prod_{0:31}$ ← $(vA)_{i+16:i+31}$ $*_{ui}$ $(vB)_{i+n:i+311}$
     $vD_{i:i+31}$ ← $prod_{0:31}$
end
```

Each odd-numbered unsigned-integer half-word element in **v**A is multiplied by the corresponding unsigned-integer half-word element in **v**B. The four 32-bit unsigned-integer products are placed, in the same order, into the four words of **v**D.

Other registers altered:

- None

Figure 6-100 shows the usage of the **vmulouh** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the four elements in the vector, **v**D, is 32 bits long.

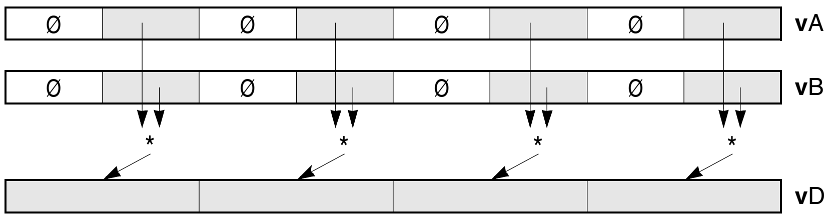


Figure 6-100. vmulouh—Odd Multiply of Four Unsigned Integer Elements (16-Bit)

vnmsubfp vnmsubfp

Vector Negative Multiply-Subtract Floating-Point

vnmsubfp vD,vA,vC,vB Form: VA

04	vD	vA	vB	vC	47
0 5	6 10	11 15	16 20	21 25	26 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← -RndToNearFP32(((vA)_{i:i+31} *_fp (vC)_{i:i+31}) -_fp (vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**A is multiplied by the corresponding single-precision floating-point word element in **v**C. The corresponding single-precision floating-point word element in **v**B is subtracted from the product. The sign of the difference is inverted. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

Note that only one rounding occurs in this operation. Also note that a QNaN result is not negated.

Other registers altered:

- None

Figure 6-101 shows the usage of the **vnmsubfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

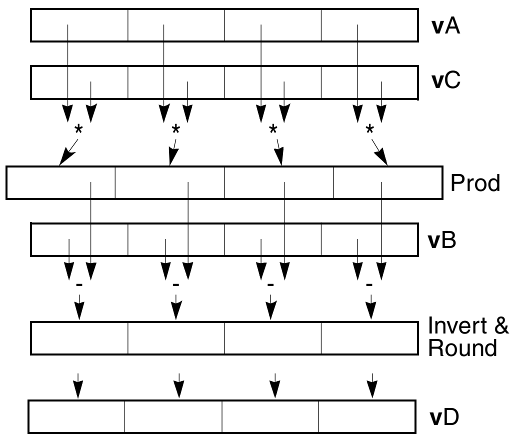


Figure 6-101. vnmsubfp—Negative Multiply-Subtract of Four Floating-Point Elements (32-Bit)

vnor vnor

Vector Logical NOR

vnor vD,vA,vB Form: VX

04	vD	vA	vB	1284
0 5	6 10	11 15	16 20	21 31

```
vD ← ¬((vA) | (vB))
```

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

Other registers altered:

- None

Simplified mnemonics:

vnotvD, **v**S equivalent to **vnorv**D, **v**S, **v**S

Figure 6-102 shows the usage of the **vnor** instruction.

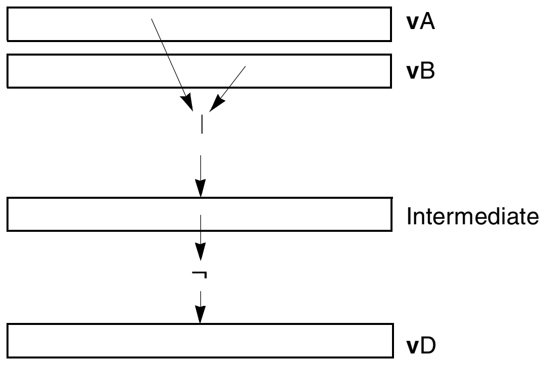


Figure 6-102. vnor—Bitwise NOR of 128-Bit Vector

vor vor

Vector Logical OR

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

04	vD	vA	vB	1156
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) | (vB)
```

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

Other registers altered:

- None

Simplified mnemonics:

vmr **v**D, **v**S equivalent to **vor** **v**D, **v**S, **v**S

Figure 6-103 shows the usage of the **vor** instruction.

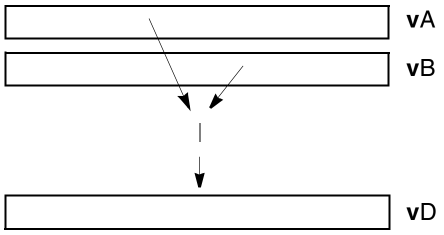


Figure 6-103. vor—Bitwise OR of 128-Bit Vector

vperm vperm

Vector Permute

vperm **v**D,**v**A,**v**B,**v**C Form: VA

04	vD	vA	vB	vC	43
0 5	6 10	11 15	16 20	21 25	26 31

```
temp$_{0:255}$ ← (vA) || (vB)
do i=0 to 127 by 8
     b ← (vC)$_{i+3:i+7}$ || 0b000
     vD$_{i:i+7}$ ← temp$_{b:b+7}$
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B. For each integer i in the range 0–15, the contents of the byte element in the source vector specified in bits 3–7 of byte element i in **v**C are placed into byte element i of **v**D.

Other registers altered:

- None

Programming note: See the programming notes with the Load Vector for Shift Left and Load Vector for Shift Right instructions for examples of usage on the **vperm** instruction.

Figure 6-104 shows the usage of the **vperm** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, **v**C, and **v**D, is 8 bits long.

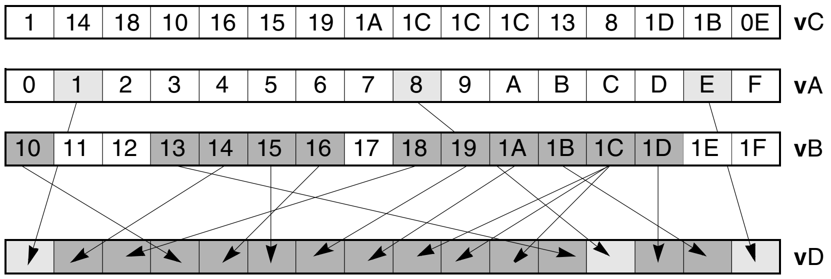

1 14 18 10 16 15 19 1A 1C 1C 1C 13 8 1D 1B 0E
vC
0 1 2 3 4 5 6 7 8 9 A B C D E F
vA
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
vB
vD

Figure 6-104. vperm—Concatenate Sixteen Integer Elements (8-Bit)

vpkpx vpkpx

Vector Pack Pixel32

vpkpx vD,vA,vB Form: VX

04	vD	vA	vB	782
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $vD_{i} \leftarrow (vA)_{i*2+7}$
    $vD_{i+1:i+5} \leftarrow (vA)_{(i*2)+8:(i*2)+12}$
    $vD_{i+6:i+1} \leftarrow (vA)_{(i*2)+16:(i*2)+20}$
    $vD_{i+11:i+15} \leftarrow (vA)(_{(i*2)+24:(i*2)+28}$
    $vD_{i+64} \leftarrow (vB)_{(i*2)+7}$
    $vD_{i+65:i+69} \leftarrow (vB)_{(i*2)+8:(i*2)+12}$
    $vD_{i+70:i+74} \leftarrow (vB)_{(i*2)+16:(i*2)+20}$
    $vD_{i+75:i+79} \leftarrow (vB)_{(i*2)+24:(i*2)+28}$
end
```

The source vector is the concatenation of the contents of **v**A followed by the contents of **v**B. Each 32-bit word element in the source vector is packed to produce a 16-bit half-word value as described below and placed into the corresponding half-word element of **v**D. A word is packed to 16 bits by concatenating, in order, the following bits.

- Bit 7 of the first byte (bit 7 of the word)
- Bits 0–4 of the second byte (bits 8–12 of the word)
- Bits 0–4 of the third byte (bits 16–20 of the word)
- Bits 0–4 of the fourth byte (bits 24–28 of the word)

Figure 6-105 shows which bits of the source word are packed to form the half word in the destination register.

Source Word																															
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
							▨	▨	▨	▨	▨	▨				▨	▨	▨	▨	▨				▨	▨	▨	▨	▨			

vD Packed Half Word															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	8	9	10	11	12	16	17	18	19	20	24	25	26	27	28

Figure 6-105. vpkpx—How a Word is Packed to a Half Word

Other registers altered:

- None

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

Figure 6-106 shows the usage of the **vpkpx** instruction. Each of the four elements in the vectors, **v**A and **v**B, is 32 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

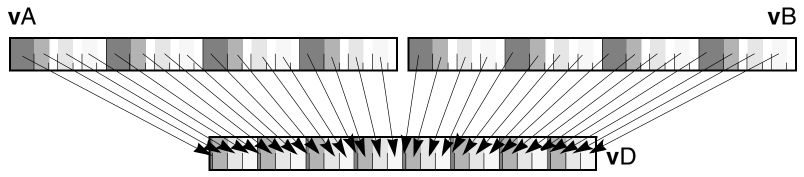


Figure 6-106. vpkpx—Pack Eight Elements (32-Bit) to Eight Elements (16-Bit)

vpkshss vpkshss

Vector Pack Signed Half Word Signed Saturate

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

04	vD	vA	vB	398
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i:i+7} ← SItoSIsat((vA)_{i*2:(i*2)+15},8)
    vD_{i+64:i+71} ← SItoSIsat((vB)_{i*2:(i*2)+15},8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer half-word element in the source vector is converted to an 8-bit signed integer. If the value of the element is greater than (2^7-1) the result saturates to (2^7-1) and if the value is less than -2^7 the result saturates to -2^7. The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-107 shows the usage of the **vpkshss** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the sixteen elements in the vector, **v**D, is 8 bits long.

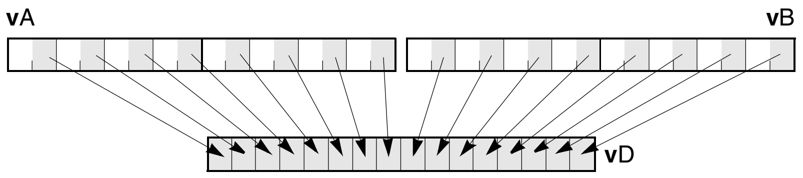


Figure 6-107. vpkshss—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Signed Integer Elements (8-Bit)

vpkshus vpkshus

Vector Pack Signed Half Word Unsigned Saturate

vpkshus vD,vA,vB Form: VX

04	vD	vA	vB	270
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i:i+7} ← SItoUIsat((vA)_{i*2:(i*2)+7},8)
    vD_{i+64:i+71} ← SItoUIsat((vB)_{i*2:(i*2)+7},8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each signed integer half-word element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than (2^8-1) the result saturates to (2^8-1) and if the value is less than 0 the result saturates to 0. The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-108 shows the usage of the **vpkshus** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the sixteen elements in the vector, **v**D, is 8 bits long.

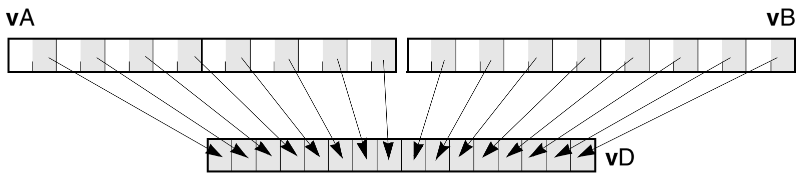


Figure 6-108. vpkshus—Pack Sixteen Signed Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkswss vpkswss

Vector Pack Signed Word Signed Saturate

vpkswss vD,vA,vB Form: VX

04	vD	vA	vB	462
0 5	6 10	11 15	16 20	21 31

```
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 **v**A followed by the contents of **v**B.

Each signed integer word element in the source vector is converted to a 16-bit signed integer half word. If the value of the element is greater than (2^{15}-1) the result saturates to (2^{15}-1) and if the value is less than -2^{15} the result saturates to -2^{15}. The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-109 shows the usage of the **vpkswss** instruction. Each of the four elements in the vectors, **v**A and **v**B, is 32 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

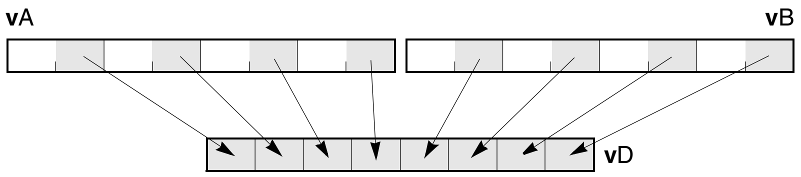


Figure 6-109. vpkswss—Pack Eight Signed Integer Elements (32-Bit) to Eight Signed Integer Elements (16-Bit)

vpkswus vpkswus

Vector Pack Signed Word Unsigned Saturate

vpkswus vD,vA,vB Form: VX

04	vD	vA	vB	334
0 5	6 10	11 15	16 20	21 31

```
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 **v**A followed by the contents of **v**B.

Each signed integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than $(2^{16}-1)$ the result saturates to $(2^{16}-1)$ and if the value is less than 0 the result saturates to 0. The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-110 shows the usage of the **vpkswus** instruction. Each of the four elements in the vectors, **v**A and **v**B, is 32 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

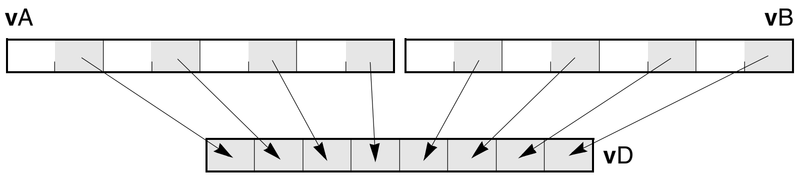


Figure 6-110. vpkswus—Pack Eight Signed Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vpkuhum vpkuhum

Vector Pack Unsigned Half Word Unsigned Modulo

vpkuhum vD,vA,vB Form: VX

04	vD	vA	vB	14
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    $\mathbf{v}D_{i:i+7} \leftarrow (\mathbf{v}A)_{(i*2)+8:(i*2)+15}$
    $\mathbf{v}D_{i+64:i+71} \leftarrow (\mathbf{v}B)_{(i*2)+8:(i*2)+15}$
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

The low-order byte of each half-word element in the source vector is placed into the corresponding byte element of **v**D.

Other registers altered:

- None

Figure 6-111 shows the usage of the **vpkuhum** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the sixteen elements in the vector, **v**D, is 8 bits long.

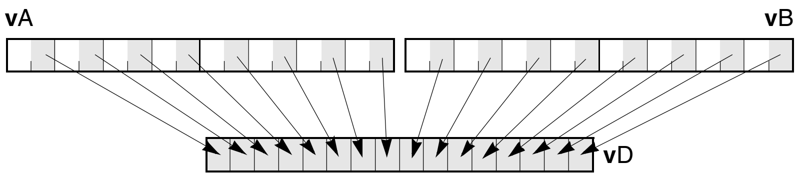


Figure 6-111. vpkuhum—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkuhus vpkuhus

Vector Pack Unsigned Half Word Unsigned Saturate

vpkuhus vD,vA,vB Form: VX

04	vD	vA	vB	142
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD$_{i:i+7}$ ← UItoUIsat((vA)$_{i*2:(i*2)+15}$,8)
    vD$_{i+64:i+71}$ ← UItoUIsat((vB)$_{i*2:(i*2)+15}$,8)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each unsigned integer half-word element in the source vector is converted to an 8-bit unsigned integer. If the value of the element is greater than (2^8-1) the result saturates to (2^8-1). The result is placed into the corresponding byte element of **v**D.

Other registers altered:

- SAT

Figure 6-112 shows the usage of the **vpkuhus** instruction. Each of the eight elements in the vectors, **v**A and **v**B, is 16 bits long. Each of the sixteen elements in the vector, **v**D, is 8 bits long.

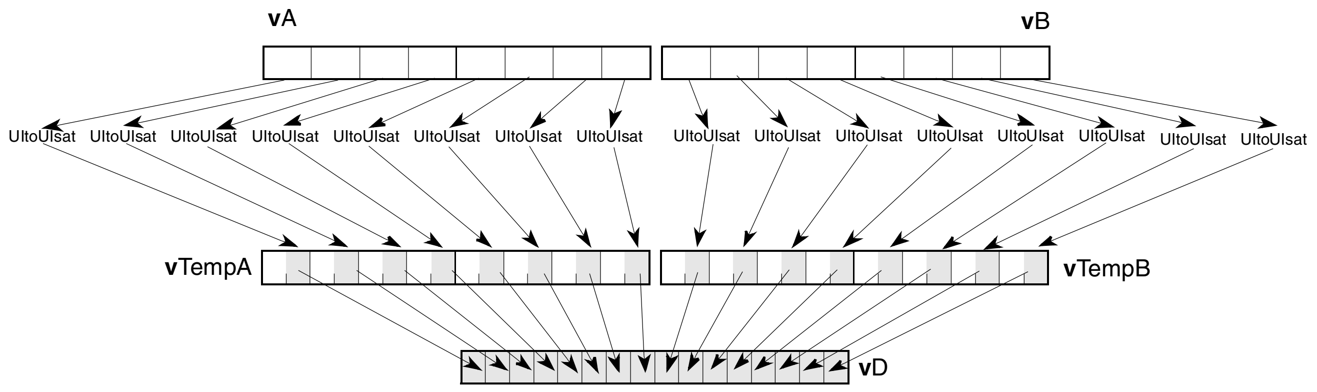


Figure 6-112. vpkuhus—Pack Sixteen Unsigned Integer Elements (16-Bit) to Sixteen Unsigned Integer Elements (8-Bit)

vpkuwum vpkuwum

Vector Pack Unsigned Word Unsigned Modulo

vpkuwum vD,vA,vB Form: VX

04	vD	vA	vB	78
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{(i*2)+16:(i*2)+31}$
    $\mathbf{v}D_{i+64:i+79} \leftarrow (\mathbf{v}B)_{(i*2)+16:(i*2)+31}$
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

The low-order half word of each word element in the source vector is placed into the corresponding half-word element of **v**D.

Other registers altered:

- None

Figure 6-113 shows the usage of the **vpkuwum** instruction. Each of the four elements in the vectors, **v**A and **v**B, is 32 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

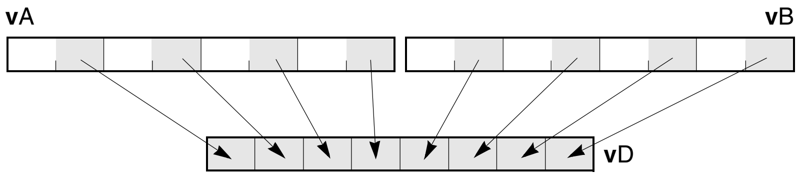


Figure 6-113. vpkuwum—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vpkuwus vpkuwus

Vector Pack Unsigned Word Unsigned Saturate

vpkuwus vD,vA,vB Form: VX

04	vD	vA	vB	206
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i:i+15} ← UItoUIsat((vA)_{i*2:(i*2)+31},16)
    vD_{i+64:i+79} ← UItoUIsat((vB)_{i*2:(i*2)+31},16)
end
```

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B.

Each unsigned integer word element in the source vector is converted to a 16-bit unsigned integer. If the value of the element is greater than (2^{16}-1) the result saturates to (2^{16}-1). The result is placed into the corresponding half-word element of **v**D.

Other registers altered:

- SAT

Figure 6-114 shows the usage of the **vpkuwus** instruction. Each of the four elements in the vectors, **v**A and **v**B, is 32 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

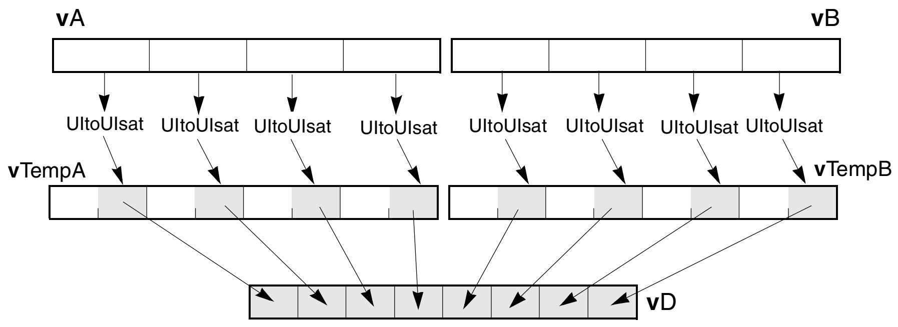


Figure 6-114. vpkuwus—Pack Eight Unsigned Integer Elements (32-Bit) to Eight Unsigned Integer Elements (16-Bit)

vrefp vrefp

Vector Reciprocal Estimate Floating-Point

vrefp **v**D,**v**B Form: VX

04	vD	0_0000	vB	266
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    x ← (vB)i:i+31
    vDi:i+31 ← 1/x
end
```

The single-precision floating-point estimate of the reciprocal of each single-precision floating-point element in **v**B is placed into the corresponding element of **v**D.

For results that are not a +0, -0, +∞, -∞, or QNaN, the estimate has a relative error in precision no greater than one part in 4096, that is:

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

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

Operation with various special values of the element in **v**B is summarized below in Table 6-7.

Table 6-7. 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 6-115 shows the usage of the **vrefp** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

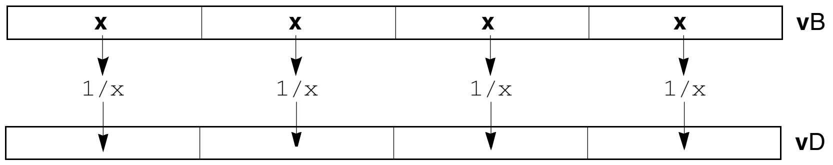


Figure 6-115. vrefp—Reciprocal Estimate of Four Floating-Point Elements (32-Bit)

vrfim vrfim

Vector Round to Floating-Point Integer Toward Minus Infinity

vrfim vD,vB Form: VX

04	vD	0_0000	vB	714
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← RndToFPInt32Floor((vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode round toward minus infinity, and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-116 shows the usage of the **vrfim** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

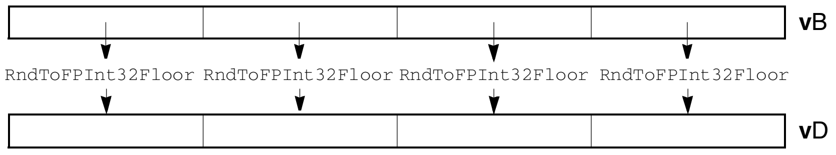


Figure 6-116. vrfim—Round to Minus Infinity of Four Floating-Point Integer Elements (32-Bit)

vrfin vrfin

Vector Round to Floating-Point Integer Nearest

vrfin vD,vB Form: VX

04	vD	0_0000	vB	522
0 5	6 10	11 15	16 20	21 31

```
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 **v**B is rounded to a single-precision floating-point integer, using the rounding mode round to nearest, and placed into the corresponding word element of **v**D.

Note the result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-117 shows the usage of the **vrfin** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

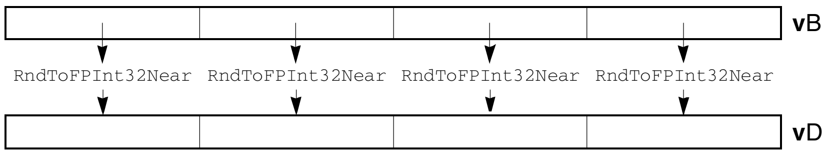


Figure 6-117. vrfin—Nearest Round to Nearest of Four Floating-Point Integer Elements (32-Bit)

vrfip vrfip

Vector Round to Floating-Point Integer Toward Plus Infinity

vrfip vD,vB Form: VX

04	vD	0_0000	vB	650
0 5	6 10	11 15	16 20	21 31

```
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 **v**B is rounded to a single-precision floating-point integer, using the rounding mode round toward plus infinity, and placed into the corresponding word element of **v**D.

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

Other registers altered:

- None

Figure 6-118 shows the usage of the **vrfip** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

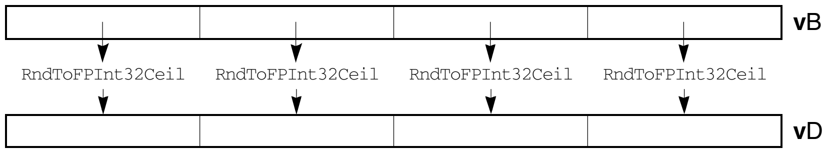


Figure 6-118. vrfip—Round to Plus Infinity of Four Floating-Point Integer Elements (32-Bit)

vrfiz vrfiz

Vector Round to Floating-Point Integer Toward Zero

vrfiz **v**D,**v**B Form: VX

04	vD	0_0000	vB	586
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
     $\mathbf{v}D_{i:i+31}$ ← RndToFPInt32Trunc(($\mathbf{v}B)_{i:i+31}$)
end
```

Each single-precision floating-point word element in **v**B is rounded to a single-precision floating-point integer, using the rounding mode round toward zero, and placed into the corresponding word element of **v**D.

Note, the result is independent of VSCR[NJ].

Other registers altered:

- None

Figure 6-119 shows the usage of the **vrfiz** instruction. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

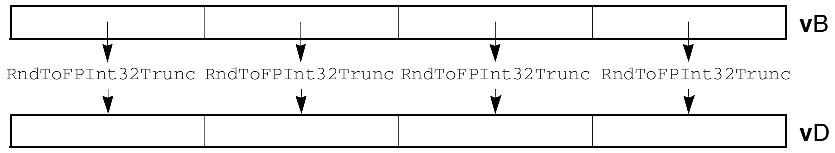


Figure 6-119. vrfiz—Round-to-Zero of Four Floating-Point Integer Elements (32-Bit)

vrlb vrlb

Vector Rotate Left Integer Byte

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

04	vD	vA	vB	4
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
     sh ← (vB)$_{i+5:i+7}$
     vD$_{i:i+7}$ ← ROTL((vA)$_{i:i+7}$,sh)
end
```

Each element is a byte. Each element in **v**A is rotated left by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-120 shows the usage of the **vrlb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

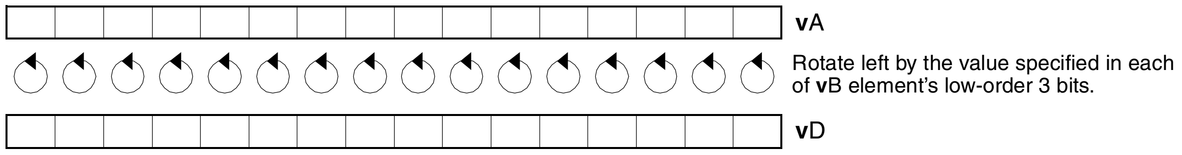

vA
Rotate left by the value specified in each of vB element's low-order 3 bits.
vD

Figure 6-120. vrlb—Left Rotate of Sixteen Integer Elements (8-Bit)

vrlh vrlh

Vector Rotate Left Integer Half Word

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

04	vD	vA	vB	68
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} ← ROTL((vA)_{i:i+15},sh)
end
```

Each element is a half word.

Each element in **v**A is rotated left by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-121 shows the usage of the **vrlh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

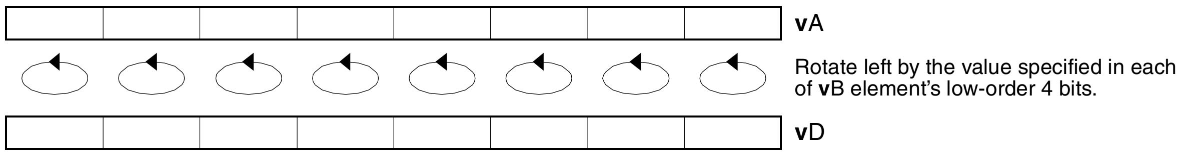

vA
Rotate left by the value specified in each of vB element's low-order 4 bits.
vD

Figure 6-121. vrlh—Left Rotate of Eight Integer Elements (16-Bit)

vrlw vrlw

Vector Rotate Left Integer Word

vrlw vD,vA,vB Form: VX

04	vD	vA	vB	132
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
     sh ← (vB)$_{i+27:i+31}$
     vD$_{i:i+31}$ ← ROTL((vA)$_{i:i+31}$,sh)
end
```

Each element is a word. Each element in **v**A is rotated left by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-122 shows the usage of the **vrlw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

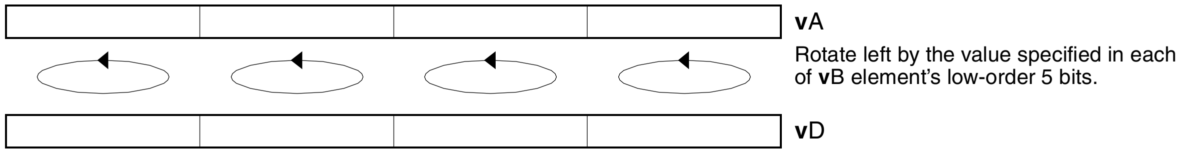

vA
Rotate left by the value specified in each of vB element's low-order 5 bits.
vD

Figure 6-122. vrlw—Left Rotate of Four Integer Elements (32-Bit)

vrsqrtefp vrsqrtefp

Vector Reciprocal Square Root Estimate Floating-Point

vrsqrtefp vD,vB Form: VX

04	vD	0_0000	vB	330
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    x ← (vB)$_{i:i+31}$
    vD$_{i:i+31}$ ← 1 ÷$_{fp}$ (√$_{fp}$(x))
end
```

The single-precision estimate of the reciprocal of the square root of each single-precision element in **v**B is placed into the corresponding word element of **v**D. The estimate has a relative error in precision no greater than one part in 4096, as explained below:

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

where *x* is the value of the element in **v**B. Note that the value placed into the element of **v**D may vary between implementations and between different executions on the same implementation. Operation with various special values of the element in **v**B is summarized below in Table 6-8.

Table 6-8. Special Values of the Element in vB

Value	Result	Value	Result
–∞	QNaN	+0	+∞
less than 0	QNaN	+∞	+0
–0	–∞	NaN	QNaN

Other registers altered:

- None

Figure 6-123 shows the usage of the **vrsqrtefp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

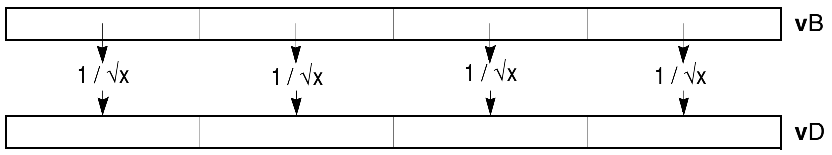


Figure 6-123. vrsqrtefp—Reciprocal Square Root Estimate of Four Floating-Point Elements (32-Bit)

vsel vsel

Vector Conditional Select

vsel **v**D,**v**A,**v**B,**v**C Form: VA

04	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 vD_i ← (vA)_i
      else vD_i ← (vB)_i
end
```

For each bit in **v**C that contains the value 0, the corresponding bit in **v**A is placed into the corresponding bit of **v**D. For each bit in **v**C that contains the value 1, the corresponding bit in **v**B is placed into the corresponding bit of **v**D.

Other registers altered:

- None

Figure 6-124 shows the usage of the **vsel** instruction. Each of the vectors, **v**A, **v**B, **v**C, and **v**D, is 128 bits long.

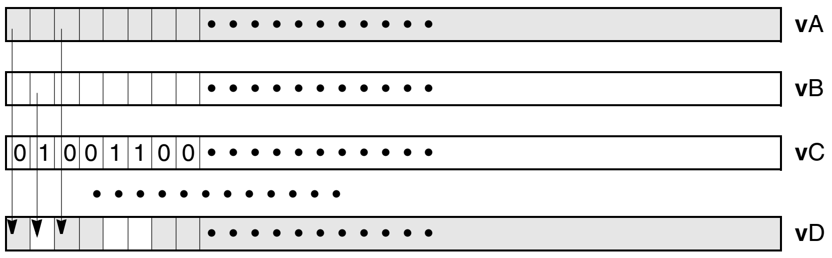


Figure 6-124. vsel—Bitwise Conditional Select of Vector Contents (128-Bit)

vsl vsl

Vector Shift Left

vsl **v**D,**v**A,**v**B 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)
     if t = 1 then vD ← (vA) <<ui sh
     else vD ← undefined
end
```

The contents of **v**A are shifted left by the number of bits specified in **v**B[125–127]. Bits shifted out of bit 0 are lost. Zeros are supplied to the vacated bits on the right. The result is placed into **v**D.

The contents of the low-order three bits of all byte elements in **v**B must be identical to **v**B[125–127]; otherwise the value placed into **v**D is undefined.

Other registers altered:

- None

Figure 6-125 shows the usage of the **vsl** instruction. In the example in Figure 6-125 the shift count specificed in **v**B[125–127] is 7 (0b111), that shift count is then put in the low-order three bits of all the byte elements in **v**B. In this example, the contents of **v**A are shifted left by 7 bits and replaced with zeros.

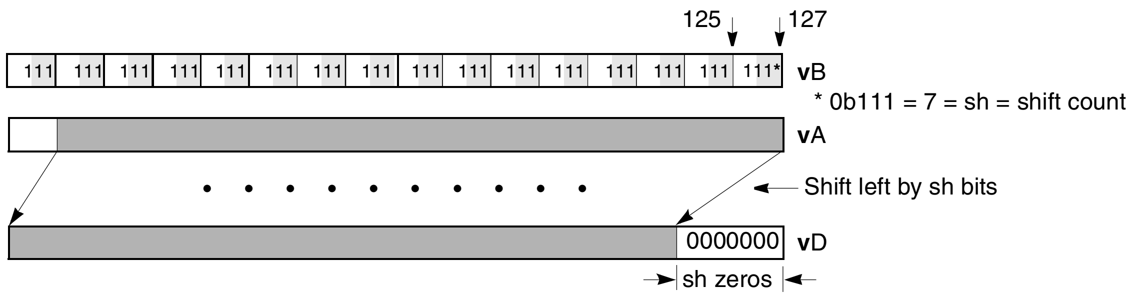


Figure 6-125. vsl—Shift Bits Left in Vector (128-Bit)

vslb vslb

Vector Shift Left Integer Byte

vslb vD,vA,vB Form: VX

04	vD	vA	vB	260
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
     sh ← (vB)i+5):i+7
     vDi:i+7 ← (vA)i:i+7 <<ui sh
end
```

Each element is a byte. Each element in **v**A is shifted left by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-126 shows the usage of the **vslb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long. In the example in Figure 6-126 the shift count specified is 7 (0b111) and this value is placed in each of the low-order 3 bits for all the byte elements in **v**B. In this example, 7 bits would be shifted out of each element in **v**A and replaced with zeros.

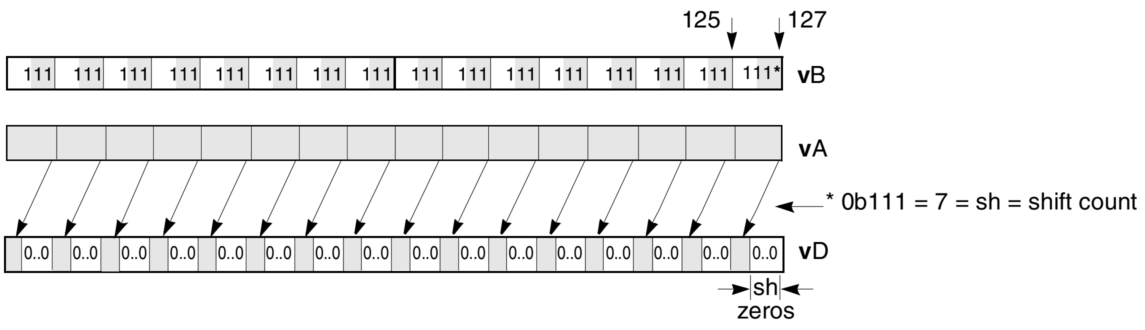


Figure 6-126. vslb—Shift Bits Left in Sixteen Integer Elements (8-Bit)

vsldoi vsldoi

Vector Shift Left Double by Octet Immediate

vsldoi **v**D, **v**A, **v**B, SHB Form: VA

04	vD	vA	vB	0	SHB	44
0 5	6 10	11 15	16 20	21	22 25	26 31

$$\mathbf{v}D \leftarrow ((\mathbf{v}A) \parallel (\mathbf{v}B)) \ll_{ui} (SHB \parallel 0b000)$$

Let the source vector be the concatenation of the contents of **v**A followed by the contents of **v**B. Bytes SHB:SHB+15 of the source vector are placed into **v**D.

Other registers altered:

- None

Figure 6-127 shows the usage of the **vsldoi** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long. In this example the shift count (SHB) is 4.

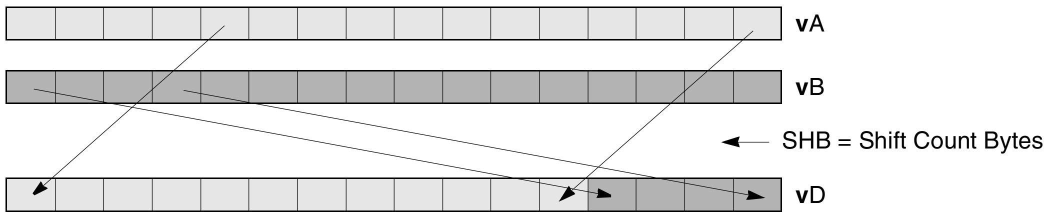


Figure 6-127. vsldoi—Shift Left by Bytes Specified

vslh vslh

Vector Shift Left Integer Half Word

vslh **v**D,**v**A,**v**B 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 half word. Each element in **v**A is shifted left by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-128 shows the usage of the **vslh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long. For the example used in Figure 6-128 the shift count specificed in the low-order 4 bits of each element in **v**B is 15 (0b1111); so 15 bits would be shifted out of each element in **v**A and replaced with zeros.

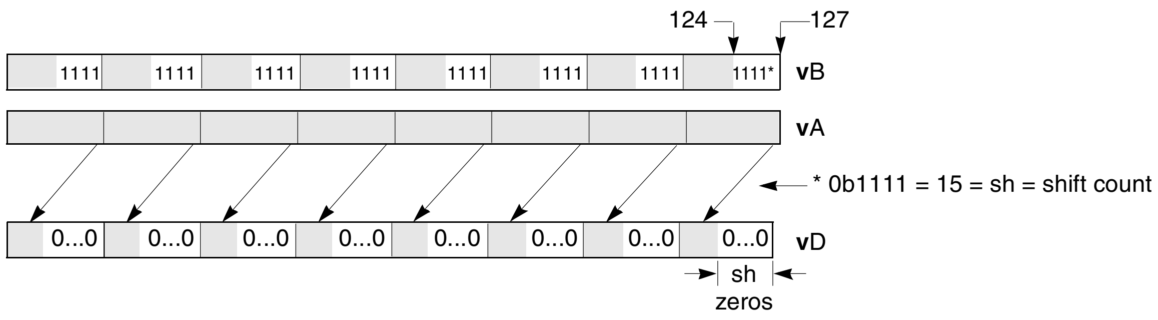


Figure 6-128. vslh—Shift Bits Left in Eight Integer Elements (16-Bit)

vslo vslo

Vector Shift Left by Octet

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

04	vD	vA	vB	1036
0 5	6 10	11 15	16 20	21 31

```
shb ← (vB)121:124
vD  ← (vA) <<ui (shb || 0b000)
```

The contents of **v**A are shifted left by the number of bytes specified in **v**B[121–124]. Bytes shifted out of byte 0 are lost. Zeros are supplied to the vacated bytes on the right. The result is placed into **v**D.

Other registers altered:

- None

Figure 6-129 shows the usage of the **vslo** instruction. For the example used in Figure 6-129 the shift count specificed in **v**B[121–124] is 4 (0b0010). So in this example, 4 bytes would be shifted out of **v**A and replaced with zeros.

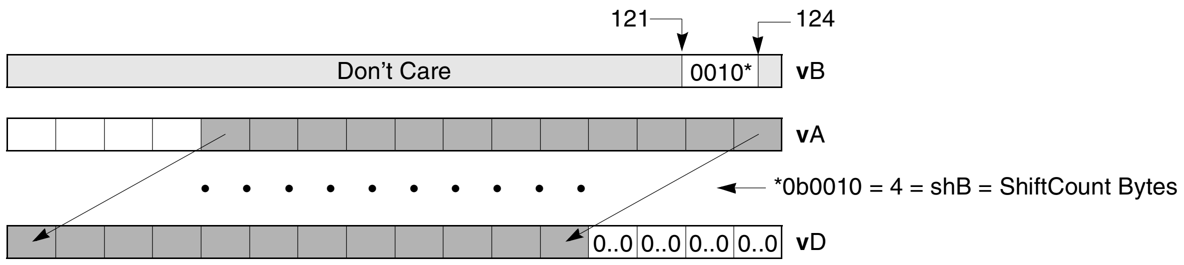


Figure 6-129. vslo—Left Byte Shift of Vector (128-Bit)

vslw vslw

Vector Shift Left Integer Word

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

04	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 element in **v**A is shifted left by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-130 shows the usage of the **vslw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long. For the example used in Figure 6-130, the shift count specificed in the low-order 5 bits of the corresponding elements in **v**B is 6 (0b0010). In this example, 6 bits would be shifted out of each element in **v**A and replaced with zeros.

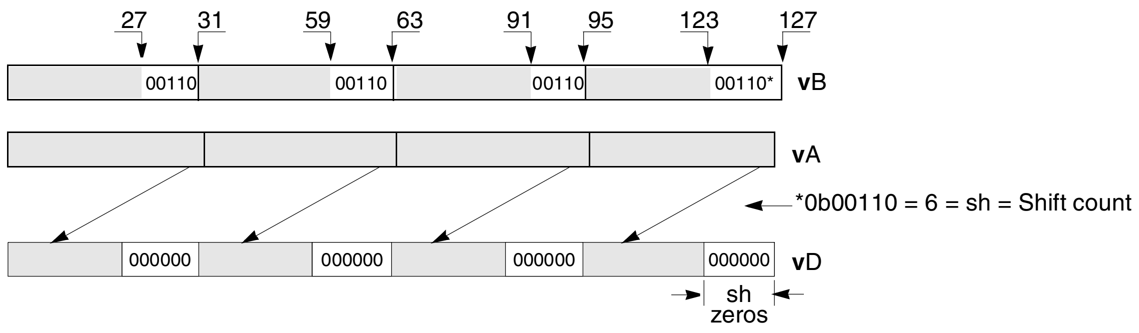


Figure 6-130. vslw—Shift Bits Left in Four Integer Elements (32-Bit)

vspltb   vspltb

Vector Splat Byte

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
```

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

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming Note: The vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a vector register by a constant).

Figure 6-131 shows the usage of the **vspltb** instruction. Each of the sixteen elements in the vectors, **v**B and **v**D, is 8 bits long. In the example UIMM = 7.

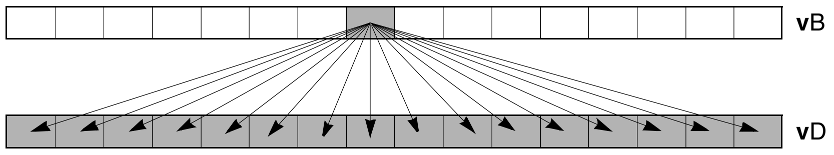


Figure 6-131. vspltb—Copy Contents to Sixteen Elements (8-Bit)

vsplth vsplth

Vector Splat Half Word

vsplth vD,vB,UIMM Form: VX

04	vD	//	UIMM	vB	588
0 5	6 10	11 12	13 15	16 20	21 31

```
b ← UIMM*16
do i=0 to 127 by 16
    $\mathbf{v}D_{i:i+15}$ ← $(\mathbf{v}B)_{b:b+15}$
end
```

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

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming Note: The vector splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a vector register by a constant).

Figure 6-132 shows the usage of the **vsplth** instruction. Each of the eight elements in the vectors, **v**B and **v**D, is 16 bits long. In the example UIMM = 1.

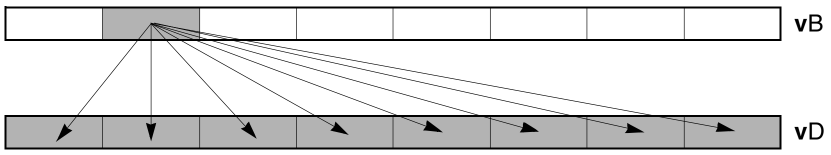


Figure 6-132. vsplth—Copy Contents to Eight Elements (16-Bit)

vspltisb vspltisb

Vector Splat Immediate Signed Byte

vspltisb vD,SIMM Form: VX

04	vD	SIMM	0000_0	780
0 5	6 10	11 15	16 20	21 31

```
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 the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-133 shows the usage of the **vspltisb** instruction. Each of the sixteen elements in the vector, **v**D, is 8 bits long.

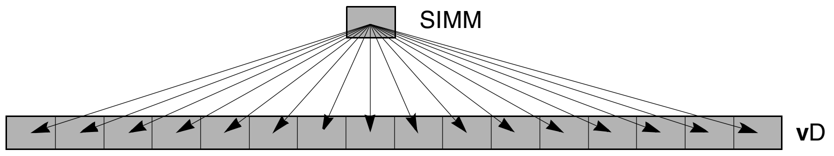


Figure 6-133. vspltisb—Copy Value into Sixteen Signed Integer Elements (8-Bit)

vspltish vspltish

Vector Splat Immediate Signed Half Word

vspltish vD,SIMM Form: VX

04	vD	SIMM	0000_0	844
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    vD_{i:i+15} ← SignExtend(SIMM,16)
end
```

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

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-134 shows the usage of the **vspltish** instruction. Each of the eight elements in vector, **v**D, is 16 bits long.

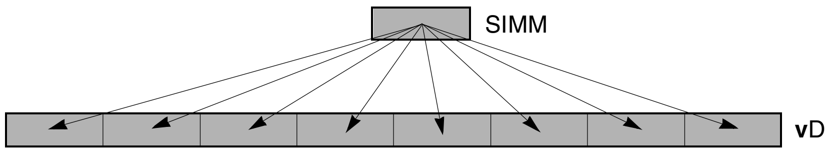


Figure 6-134. vspltish—Copy Value to Eight Signed Integer Elements (16-Bit)

vspltisw vspltisw

Vector Splat Immediate Signed Word

vspltisw vD,SIMM Form: VX

04	vD	SIMM	0000_0	908
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← SignExtend(SIMM,32)
end
```

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

The value of the SIMM field, sign-extended to the length of the element, is replicated into each element of **v**D.

Other registers altered:

- None

Figure 6-135 shows the usage of the **vspltisw** instruction. Each of the four elements in the vector, **v**D, is 32 bits long.

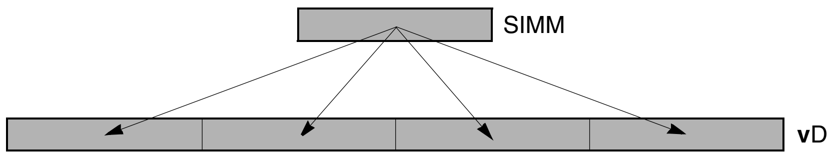


Figure 6-135. vspltisw—Copy Value to Four Signed Elements (32-Bit)

vspltw vspltw

Vector Splat Word

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
    $\mathbf{v}D_{i:i+31}$ ← $(\mathbf{v}B)_{b:b+31}$
end
```

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

The contents of element UIMM in **v**B are replicated into each element of **v**D.

Other registers altered:

- None

Programming Note: The Vector Splat instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (for example, multiplying all elements of a Vector Register by a constant).

Figure 6-136 shows the usage of the **vspltw** instruction. Each of the four elements in the vector, **v**D, is 32 bits long. In the example UIMM = 1.

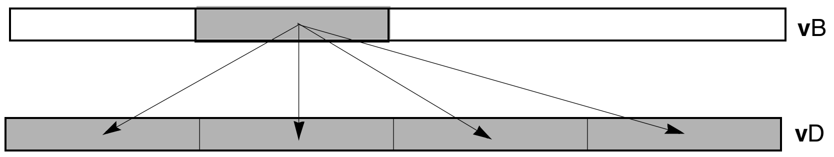


Figure 6-136. vspltw—Copy Contents to Four Elements (32-Bit)

vsr vsr

Vector Shift Right

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

04	vD	vA	vB	708
0 5	6 10	11 15	16 20	21 31

```
sh ← (vB)$_{125:127}$
t ← 1
do i = 0 to 127 by 8
    t ← t & ((vB)$_{i+5:i+7}$ = sh)
    if t = 1 then vD ← (vA) >>$_{ui}$ sh
    else vD ← undefined
end
```

Let sh = **v**B[125–127]; sh is the shift count in bits (0≤sh≤7). The contents of **v**A are shifted right by sh bits. Bits shifted out of bit 127 are lost. Zeros are supplied to the vacated bits on the left. The result is placed into **v**D.

The contents of the low-order three bits of all byte elements in register **v**B must be identical to **v**B[125–127]; otherwise the value placed into register **v**D is undefined.

Other registers altered:

- None

Programming Notes: A pair of **vslo** and **vsl** or **vsro** and **vsr** instructions, specifying the same shift count register, can be used to shift the contents of a vector register left or right by the number of bits (0–127) specified in the shift count register. The following example shifts the contents of **v**X left by the number of bits specified in **v**Y and places the result into **v**Z.

```
vslo     VZ,VX,VY
vsl      VZ,VZ,VY
```

A double-register shift by a dynamically specified number of bits (0–127) can be performed in six instructions. The following example shifts (**v**W) || (**v**X) left by the number of bits specified in **v**Y and places the high-order 128 bits of the result into **v**Z.

```
vslo     t1,VW,VY  #shift high-order reg left
vsl      t1,t1,VY
vsububm  t3,V0,VY  #adjust shift count ((V0)=0)
vsro     t2,VX,t3  #shift low-order reg right
vsr      t2,t2,t3
vor       VZ,t1,t2  #merge to get final result
```

Figure 6-137 shows the usage of the **vsr** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long. In the example in Figure 6-137 the shift count specificed in **v**B[125–127] is 7 (0b111), that shift count must then be put in the low-order three bits of all the byte elements in **v**B. In this example, the contents of **v**A are shifted right by 7 bits and replaced with zeros.

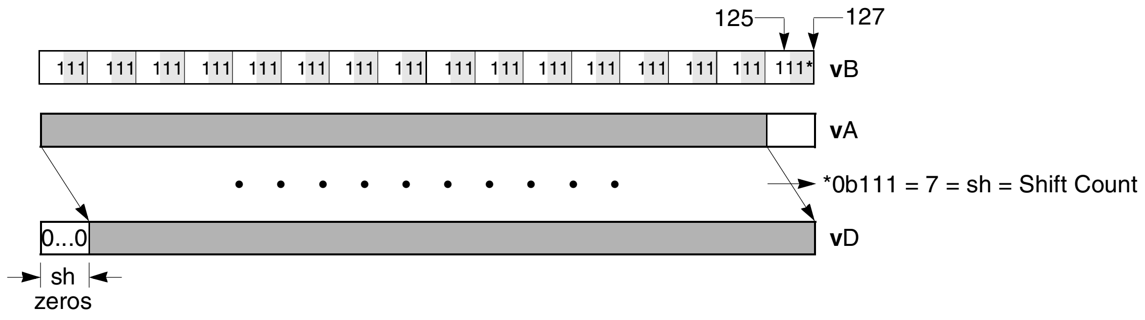


Figure 6-137. vsr—Shift Bits Right for Vectors (128-Bit)

vsrab vsrab

Vector Shift Right Algebraic Byte

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 element in **v**A is shifted right by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit n-1 of the element are lost. Bit 0 of each element in **v**A is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-138 shows the usage of the **vsrab** instruction. Each of the sixteen elements in the vectors, **v**A and **v**D, is 8 bits long. In the example in Figure 6-138, the shift count specified is 7 (0b111) so this value is placed in each of the low-order three bits for all the byte elements in **v**B. In this example, 7 bits would be shifted out of each element in **v**A and replaced with bit 0 of each element in **v**A.

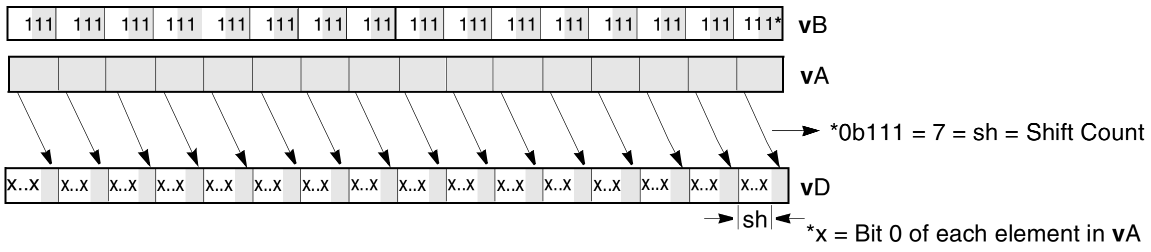

111 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111*
vB
vA
*0b111 = 7 = sh = Shift Count
x..x x..x x..x x..x x..x x..x x..x x..x x..x x..x x..x x..x x..x x..x x..x x..x
vD
sh
*x = Bit 0 of each element in vA

Figure 6-138. vsrab—Shift Bits Right in Sixteen Integer Elements (8-Bit)

vsrah vsrah

Vector Shift Right Algebraic Half Word

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

04	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 element is a half word. Each element in **v**A is shifted right by the number of bits specified in the low-order 4 bits of the corresponding element in **v**B. Bits shifted out of bit 15 of the element are lost. Bit 0 of the element in **v**A is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-139 shows the usage of the **vsrah** instruction. Each of the eight elements in the vectors, **v**A and **v**D, is 16 bits long. In the example in Figure 6-139, the shift count specified is 15 (0b1111) so this value is placed in each of the low-order four bits for all the half-word elements in **v**B. In this example, 15 bits would be shifted out of each element in **v**A and replaced with bit 0 of each element in **v**A.

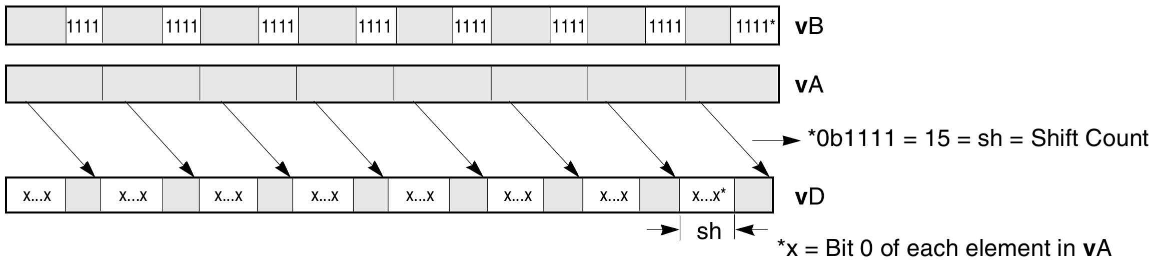


Figure 6-139. vsrah—Shift Bits Right for Eight Integer Elements (16-Bit)

vsraw vsraw

Vector Shift Right Algebraic Word

vsraw vD,vA,vB Form: VX

04	vD	vA	vB	900
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)$_{i:i+31}$ >>$_{si}$ sh
end
```

Each element is a word. Each element in **v**A is shifted right by the number of bits specified in the low-order five bits of the corresponding element in **v**B. Bits shifted out of bit 31 of the element are lost. Bit 0 of each element in **v**A is replicated to fill the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-140 shows the usage of the **vsraw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long. In the example in Figure 6-140, the shift count specified is 3 (0b00011) so this value is placed in each of the low-order five bits for all the byte elements in **v**B. In this example, 3 bits would be shifted out of each element in **v**A and replaced with bit 0 of each element in **v**A.

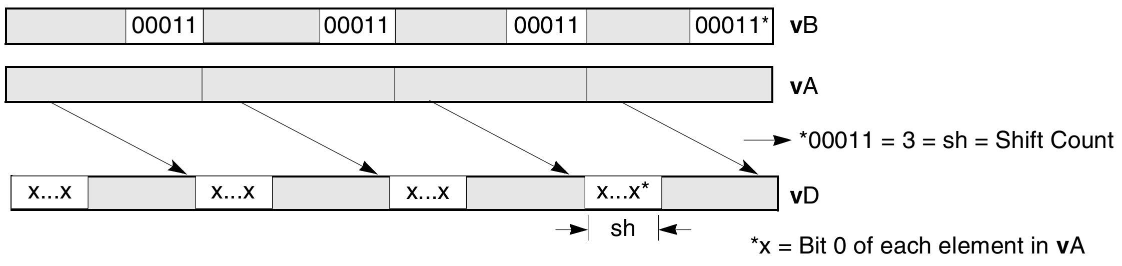

00011
00011
00011
00011*
vB
vA
*00011 = 3 = sh = Shift Count
x...x
x...x
x...x
x...x*
vD
sh
*x = Bit 0 of each element in vA

Figure 6-140. vsraw—Shift Bits Right in Four Integer Elements (32-Bit)

vsrb vsrb

Vector Shift Right Byte

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

04	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 **v**A is shifted right by the number of bits specified in the low-order 3 bits of the corresponding element in **v**B. Bits shifted out of bit 7 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-141 shows the usage of the **vsrb** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long. In the example in Figure 6-141 the shift count specified is 7 (0b111) so this value is placed in each of the low-order 3 bits for all the byte elements in **v**B. In this example, 7 bits would be shifted out of each element in **v**A and replaced with zeros.

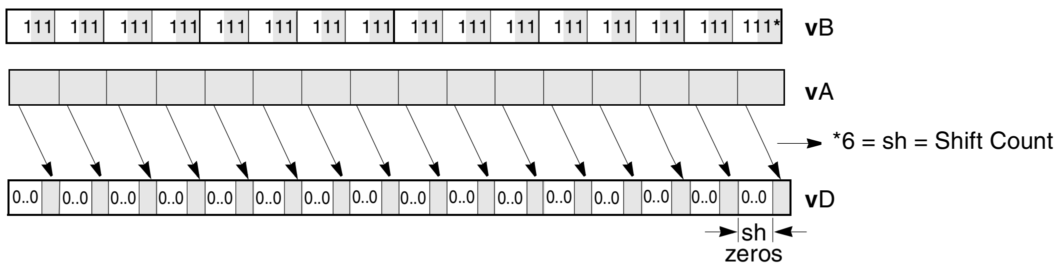


Figure 6-141. vsrb—Shift Bits Right in Sixteen Integer Elements (8-Bit)

vsrh vsrh

Vector Shift Right Half Word

vsrh vD,vA,vB 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
     vDi:i+15 ← (vA)i:i+15 >>ui sh
end
```

Each element is a half word. Each element in **v**A is shifted right by the number of bits specified in the low-order four bits of the corresponding element in **v**B. Bits shifted out of bit 15 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-142 shows the usage of the **vsrh** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long. In the example in Figure 6-142, the shift count specified is 15 (0b1111) so this value is placed in each of the low-order four bits for all the half-word elements in **v**B. In this example, 15 bits would be shifted out of each element in **v**A and replaced with zeros.

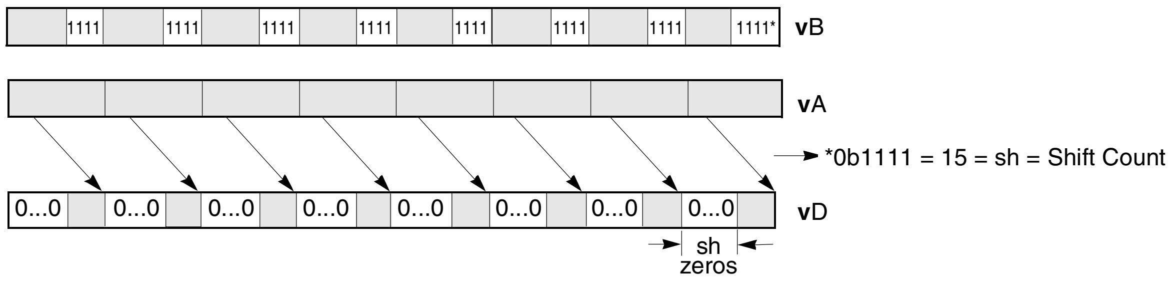


Figure 6-142. vsrh—Shift Bits Right for Eight Integer Elements (16-Bit)

vsro vsro

Vector Shift Right Octet

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

04	vD	vA	vB	1100
0 5	6 10	11 15	16 20	21 31

```
shb ← (vB)121:124
vD  ← (vA) >>ui (shb || 0b000)
```

The contents of **v**A are shifted right by the number of bytes specified in **v**B[121–124]. Bytes shifted out of **v**A are lost. Zeros are supplied to the vacated bytes on the left. The result is placed into **v**D.

Other registers altered:

- None

Figure 6-143 shows the usage of the **vsro** instruction. Each of the sixteen elements in the vectors, **v**A and **v**D, is 8 bits long. In the example in Figure 6-143, the shift count specified is 5 (0b0101). In this example, 5 bytes would be shifted out of **v**A and replaced with zeros.

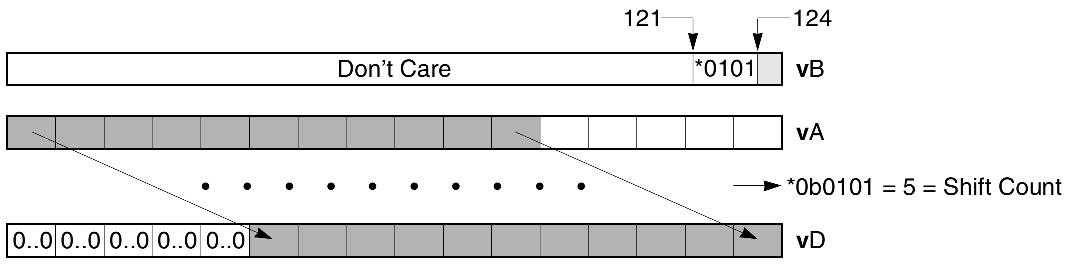


Figure 6-143. vsro—Vector Shift Right Octet

vsrw vsrw

Vector Shift Right Word

vsrw vD,vA,vB 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
     vDi:i+31 ← (vA)i:i+31 >>ui sh
end
```

Each element is a word. Each element in **v**A is shifted right by the number of bits specified in the low-order 5 bits of the corresponding element in **v**B. Bits shifted out of bit 31 of the element are lost. Zeros are supplied to the vacated bits on the left. The result is placed into the corresponding element of **v**D.

Other registers altered:

- None

Figure 6-144 shows the usage of the **vsrw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.In the example in Figure 6-144 the shift count specified is 3 (0b00011) so this value is placed in each of the low-order five bits for all the byte elements in **v**B. In this example, 3 bits would be shifted out of each element in **v**A and replaced with zeros.

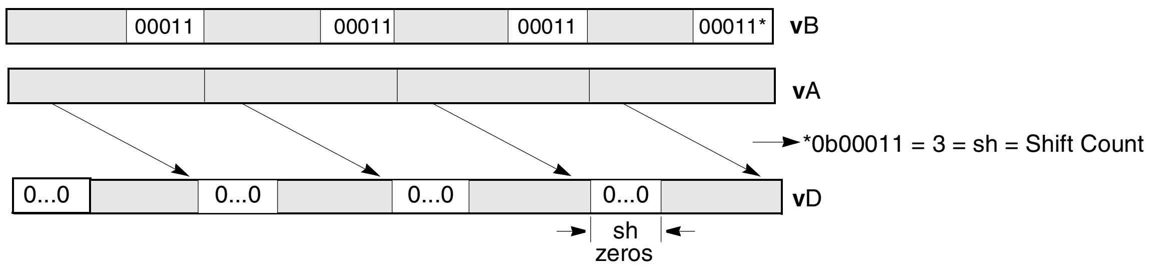


Figure 6-144. vsrw—Shift Bits Right in Four Integer Elements (32-Bit)

vsubcuw vsubcuw

Vector Subtract Carryout Unsigned Word

vsubcuw 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
    aop$_{0:32}$ ← ZeroExtend((vA)$_{i:i+31}$,33)
    bop$_{0:32}$ ← ZeroExtend((vB)$_{i:i+31}$,33)
    temp$_{0:32}$ ← aop$_{0:32}$ +$_{int}$ ¬bop$_{0:32}$ +$_{int}$ 1
    vD$_{i:i+31}$ ← ZeroExtend(temp$_0$,32)
end
```

Each unsigned-integer word element in **v**B is subtracted from the corresponding unsigned-integer word element in **v**A. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into the corresponding word element of **v**D.

Other registers altered:

- None

Figure 6-145 shows the usage of the **vsubcuw** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

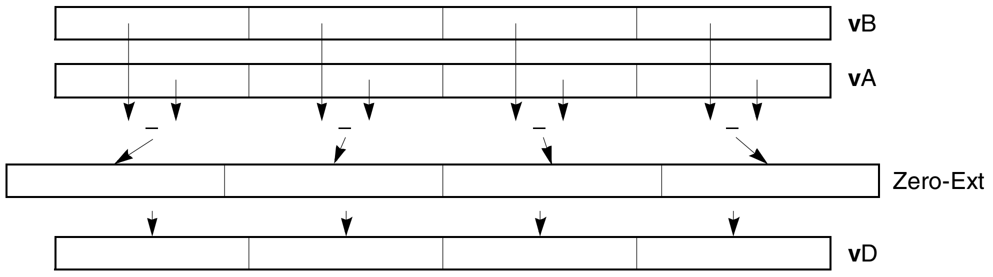


Figure 6-145. vsubcuw—Subtract Carryout of Four Unsigned Integer Elements (32-Bit)

vsubfp vsubfp

Vector Subtract Floating-Point

vsubfp vD,vA,vB Form: VX

04	vD	vA	vB	74
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← RndToNearFP32((vA)_{i:i+31} -fp (vB)_{i:i+31})
end
```

Each single-precision floating-point word element in **v**B is subtracted from the corresponding single-precision floating-point word element in **v**A. The result is rounded to the nearest single-precision floating-point number and placed into the corresponding word element of **v**D.

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

Other registers altered:

- None

Figure 6-146 shows the usage of the **vsubfp** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

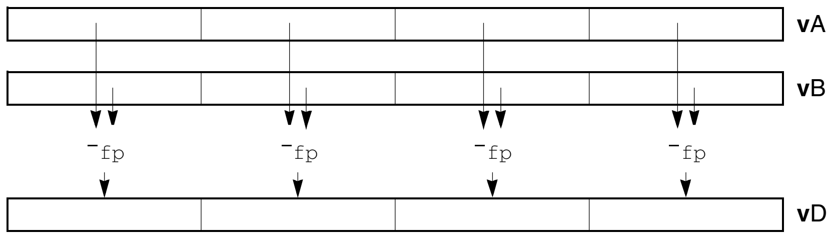


Figure 6-146. vsubfp—Subtract Four Floating-Point Elements (32-Bit)

vsubsbs vsubsbs

Vector Subtract Signed Byte Saturate

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

04	vD	vA	vB	1792
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
     $aop_{0:8}$ ← SignExtend((vA)$_{i:i+7}$,9)
     $bop_{0:8}$ ← SignExtend((vB)$_{i:i+7}$,9)
     $temp_{0:8}$ ← $aop_{0:8}$ $+_{int}$ $-bop_{0:8}$ $+_{int}$ 1
     $vD_{i:i+7}$ ← SItoSIsat($temp_{0:8}$,8)
end
```

Each element is a byte. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than (2^7-1) it saturates to (2^7-1) and if it is less than -2^7 it saturates to -2^7, where 8 is the length of the element.

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

Other registers altered:

- SAT

Figure 6-147 shows the usage of the **vsubsbs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

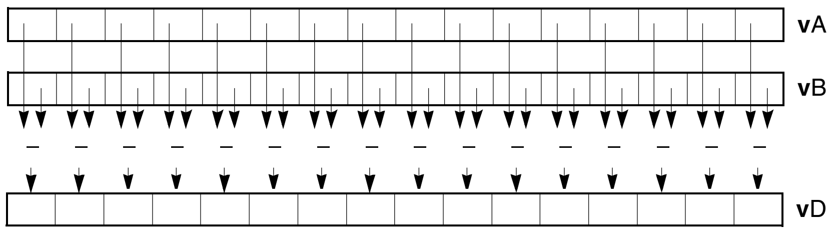


Figure 6-147. vsubsbs—Subtract Sixteen Signed Integer Elements (8-Bit)

vsubshs vsubshs

Vector Subtract Signed Half Word Saturate

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

04	vD	vA	vB	1856
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    $aop_{0:16}$ ← SignExtend((vA)$_{i:i+15}$,17)
    $bop_{0:16}$ ← SignExtend((vB)$_{i:i+15}$,17)
    $temp_{0:16}$ ← $aop_{0:16}$ $+_{int}$ $-bop_{0:16}$ $+_{int}$ 1
    $vD_{i:i+15}$ ← SItoSIsat($temp_{0:16}$,16)
end
```

Each element is a half word. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than (2^{15}-1) it saturates to (2^{15}-1) and if it is less than -2^{15} it saturates to -2^{15}, where 16 is the length of the element.

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

Other registers altered:

- SAT

Figure 6-148 shows the usage of the **vsubshs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

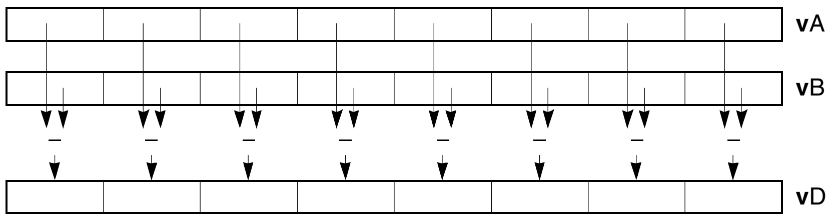


Figure 6-148. vsubshs—Subtract Eight Signed Integer Elements (16-Bit)

vsubsws vsubsws

Vector Subtract Signed Word Saturate

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

04	vD	vA	vB	1920
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $aop_{0:32}$ ← SignExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32}$ ← SignExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32}$ $+_{int}$ ¬$bop_{0:32}$ $+_{int}$ 1
    $vD_{i:i+31}$ ← SItoSIsat($temp_{0:32}$,32)
end
```

Each element is a word. Each signed-integer element in **v**B is subtracted from the corresponding signed-integer element in **v**A.

If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}, where 32 is the length of the element.

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

Other registers altered:

- SAT

Figure 6-149 shows the usage of the **vsubsws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

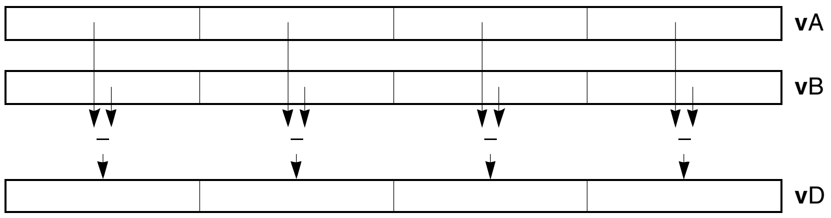


Figure 6-149. vsubsws—Subtract Four Signed Integer Elements (32-Bit)

vsububm vsububm

Vector Subtract Byte Modulo

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

04	vD	vA	vB	1024
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    vD_{i:i+7} ← (vA)_{i:i+7} +int −(vB)_{i:i+7}
end
```

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

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

Other registers altered:

- None

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

Figure 6-150 shows the usage of the **vsububm** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

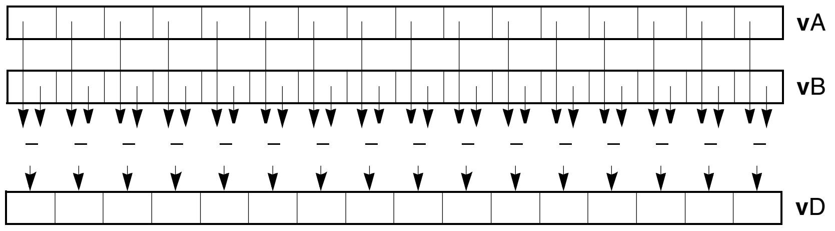


Figure 6-150. vsububm—Subtract Sixteen Integer Elements (8-Bit)

vsububs vsububs

Vector Subtract Unsigned Byte Saturate

vsububs vD,vA,vB Form: VX

04	vD	vA	vB	1536
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 8
    $aop_{0:8}$ ← ZeroExtend((vA)$_{i:i+7}$,9)
    $bop_{0:8}$ ← ZeroExtend((vB)$_{i:i+7}$,9)
    $temp_{0:8}$ ← $aop_{0:8}$ $+_{int}$ $\neg bop_{0:8}$ $+_{int}$ 1
    vD$_{i:i+7}$ ← SItoUIsat($temp_{0:8}$,8)
end
```

Each element is a byte. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 8 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-151 shows the usage of the **vsububs** instruction. Each of the sixteen elements in the vectors, **v**A, **v**B, and **v**D, is 8 bits long.

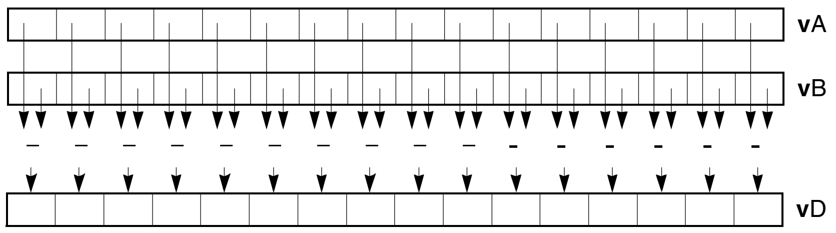


Figure 6-151. vsububs—Subtract Sixteen Unsigned Integer Elements (8-Bit)

vsubuhm vsubuhm

Vector Subtract Half Word Modulo

vsubuhm vD,vA,vB Form: VX

04	vD	vA	vB	1088
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 16
    $\mathbf{v}D_{i:i+15} \leftarrow (\mathbf{v}A)_{i:i+15} +_{int} -(\mathbf{v}B)_{i:i+15}$
end
```

Each element is a half word. Each integer element in **v**B is subtracted from the corresponding integer element in **v**A. The integer result is placed into the corresponding element of **v**D.

Other registers altered:

- None

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

Figure 6-152 shows the usage of the **vsubuhm** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

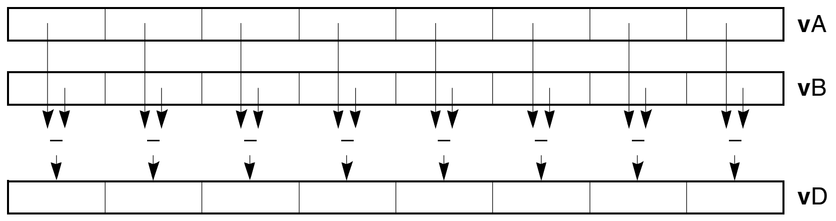


Figure 6-152. vsubuhm—Subtract Eight Integer Elements (16-Bit)

vsubuhs vsubuhs

Vector Subtract Unsigned Half Word Saturate

vsubuhs 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
    aop_{0:16} ← ZeroExtend((vA)_{i:i+15},17)
    bop_{0:16} ← ZeroExtend((vB)_{i:i+n:1},17)
    temp_{0:16} ← aop_{0:n} +_{int} ¬bop_{0:16} +_{int} 1
    vD_{i:i+15} ← SItoUIsat(temp_{0:16},16)
end
```

Each element is a half word. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 16 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-153 shows the usage of the **vsubuhs** instruction. Each of the eight elements in the vectors, **v**A, **v**B, and **v**D, is 16 bits long.

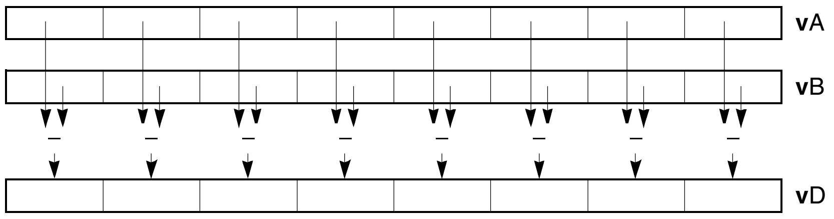


Figure 6-153. vsubuhs—Subtract Eight Unsigned Integer Elements (16-Bit)

vsubuwm vsubuwm

Vector Subtract Word Modulo

vsubuwm vD,vA,vB Form: VX

04	vD	vA	vB	1152
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    vD_{i:i+31} ← (vA)_{i:i+31} +_int −(vB)_{i:i+31}
end
```

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

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

Other registers altered:

- None

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

Figure 6-154 shows the usage of the **vsubuwm** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

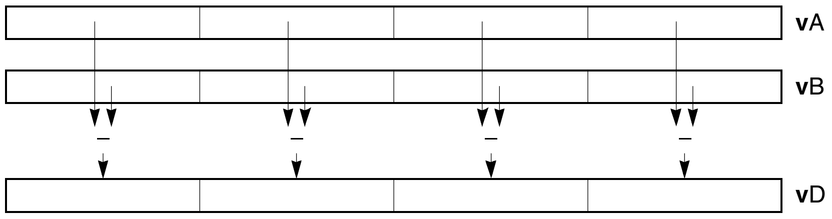


Figure 6-154. vsubuwm—Subtract Four Integer Elements (32-Bit)

vsubuws vsubuws

Vector Subtract Unsigned Word Saturate

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

04	vD	vA	vB	1664
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    $aop_{0:32}$ ← ZeroExtend((vA)$_{i:i+31}$,33)
    $bop_{0:32}$ ← ZeroExtend((vB)$_{i:i+31}$,33)
    $temp_{0:32}$ ← $aop_{0:32}$ $+_{int}$ ¬$bop_{0:32}$ $+_{int}$ 1
    vD$_{i:i+31}$ ← SItoUIsat($temp_{0:32}$,32)
end
```

Each element is a word. Each unsigned-integer element in **v**B is subtracted from the corresponding unsigned-integer element in **v**A.

If the intermediate result is less than 0 it saturates to 0, where 32 is the length of the element. The unsigned-integer result is placed into the corresponding element of **v**D.

Other registers altered:

- SAT

Figure 6-155 shows the usage of the **vsubuws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

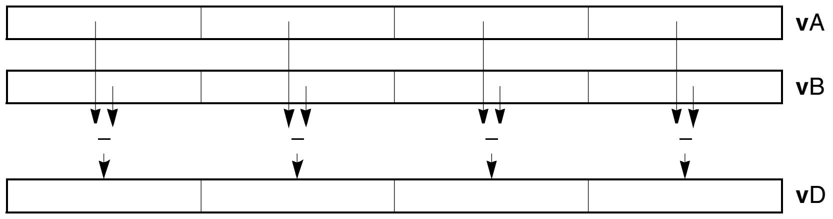


Figure 6-155. vsubuws—Subtract Four Signed Integer Elements (32-Bit)

vsumsws vsumsws

Vector Sum Across Signed Word Saturate

vsumsws vD,vA,vB Form: VX

04	vD	vA	vB	1928
0 5	6 10	11 15	16 20	21 31

```
temp_{0:34} ← SignExtend((vB)_{96:127},35)
do i=0 to 127 by 32
    temp_{0:34} ← temp_{0:34} +_{int} SignExtend((vA)_{i:i+31},35)
    vD ← ^{96}0 || SItoSIsat(temp_{0:34},32)
end
```

The signed-integer sum of the four signed-integer word elements in **v**A is added to the signed-integer word element in bits of **v**B[96–127]. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into bits **v**D[96–127]. Bits **v**D[0–95] are cleared.

Other registers altered:

- SAT

Figure 6-156 shows the usage of the **vsumsws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

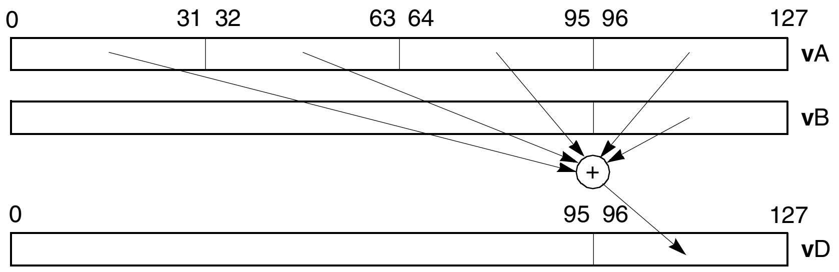


Figure 6-156. vsumsws—Sum Four Signed Integer Elements (32-Bit)

vsum2sws vsum2sws

Vector Sum Across Partial (1/2) Signed Word Saturate

vsum2sws vD,vA,vB Form: VX

04	vD	vA	vB	1672
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 64
    temp$_{0:33}$ ← SignExtend((vB)$_{i+32:i+63}$,34)
    do j=0 to 63 by 32
        temp$_{0:33}$ ← temp$_{0:33}$ +$_{int}$ SignExtend((vA)$_{i+j:i+j+31}$,34)
    end
    vD$_{i:i+63}$ ← $^{32}$0 || SItoSIsat(temp$_{0:33}$,32)
end
```

The signed-integer sum of the first two signed-integer word elements in register **v**A is added to the signed-integer word element in **v**B[32–63]. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into **v**D[32–63]. The signed-integer sum of the last two signed-integer word elements in register **v**A is added to the signed-integer word element in **v**B[96–127]. If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}. The signed-integer result is placed into **v**D[96–127]. The register **v**D[0–31,64–95] are cleared to 0.

Other registers altered:

- SAT

Figure 6-157 shows the usage of the **vsum2sws** instruction. Each of the four elements in the vectors, **v**A, **v**B, and **v**D, is 32 bits long.

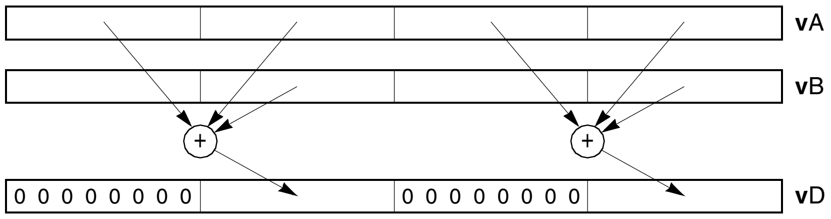


Figure 6-157. vsum2sws—Two Sums in the Four Signed Integer Elements (32-Bit)

vsum4sbs vsum4sbs

Vector Sum Across Partial (1/4) Signed Byte Saturate

vsum4sbs vD,vA,vB Form: VX

04	vD	vA	vB	1800
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
  temp0:32 ← SignExtend((vB)i:i+31,33)
  do j=0 to 31 by 8
        temp0:32 ← temp0:32 +int SignExtend((vA)i+j:i+j+7,33)
     end
     vDi:i+31 ← SItoSIsat(temp0:32,32)
end
```

For each word element in **v**B the following operations are performed in the order shown.

- The signed-integer sum of the four signed-integer byte elements contained in the corresponding word element of register **v**A is added to the signed-integer word element in register **v**B.
- If the intermediate result is greater than (2^{31}-1) it saturates to (2^{31}-1) and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-158 shows the usage of the **vsum4sbs** instruction. Each of the sixteen elements in the vector, **v**A, is 8 bits long. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

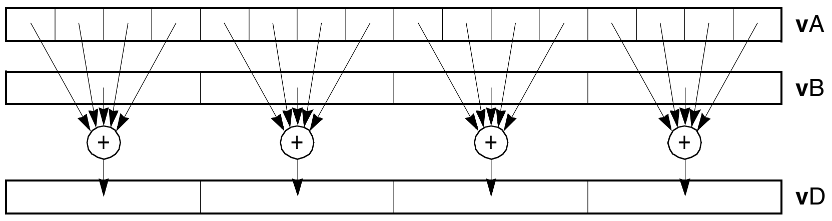


Figure 6-158. vsum4sbs—Sum of Four Signed Integer Byte Elements with a Word Element (32-Bit)

vsum4shs vsum4shs

Vector Sum Across Partial (1/4) Signed Half Word Saturate

vsum4shs **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	1608
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    temp$_{0:32}$ ← SignExtend((vB)$_{i:i+31}$,33)
    do j=0 to 31 by 16
        temp$_{0:32}$ ← temp$_{0:32}$ +$_{int}$ SignExtend((vA)$_{i+j:i+j+15}$,33)
  end
    vD$_{i:i+31}$ ← SItoSIsat(temp$_{0:32}$,32)
end
```

For each word element in register **v**B the following operations are performed, in the order shown.

- The signed-integer sum of the two signed-integer half-word elements contained in the corresponding word element of register **v**A is added to the signed-integer word element in **v**B.
- If the intermediate result is greater than $(2^{31}-1)$ it saturates to $(2^{31}-1)$ and if it is less than -2^{31} it saturates to -2^{31}.
- The signed-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-159 shows the usage of the **vsum4shs** instruction. Each of the eight elements in the vector, **v**A, is 16 bits long. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

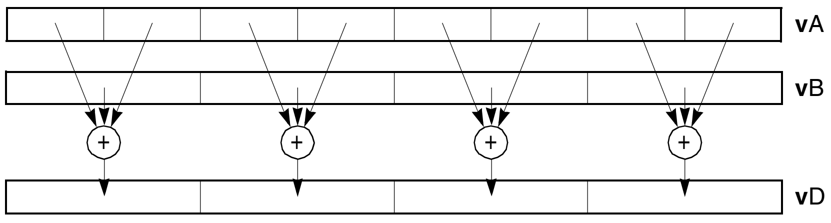


Figure 6-159. vsum4shs—Sum of Two Signed Integer Half-Word Elements with a Word Element (32-Bit)

vsum4ubs vsum4ubs

Vector Sum Across Partial (1/4) Unsigned Byte Saturate

vsum4ubs **v**D,**v**A,**v**B Form: VX

04	vD	vA	vB	1544
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 127 by 32
    temp$_{0:32}$ ← ZeroExtend((vB)$_{i:i+31}$,33)
    do j=0 to 31 by 8
    temp$_{0:32}$ ← temp$_{0:32}$ +$_{int}$ ZeroExtend((vA)$_{i+j:i+j+7}$,33)
  end
    vD$_{i:i+31}$ ← UItoUIsat(temp$_{0:32}$,32)
end
```

For each word element in **v**B the following operations are performed in the order shown.

- The unsigned-integer sum of the four unsigned-integer byte elements contained in the corresponding word element of register **v**A is added to the unsigned-integer word element in register **v**B.
- If the intermediate result is greater than (2^{32}-1) it saturates to (2^{32}-1).
- The unsigned-integer result is placed into the corresponding word element of **v**D.

Other registers altered:

- SAT

Figure 6-160 shows the usage of the **vsum4ubs** instruction. Each of the four elements in the vector, **v**A, is 8 bits long. Each of the four elements in the vectors, **v**B and **v**D, is 32 bits long.

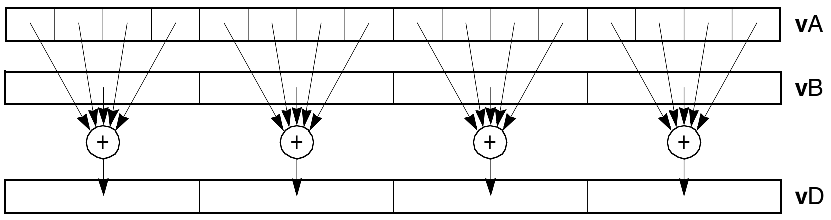


Figure 6-160. vsum4ubs—Sum of Four Unsigned Integer Byte Elements with an Unsigned Integer Word Element (32-Bit)

vupkhpx vupkhpx

Vector Unpack High Pixel16

vupkhpx **vD**,**vB** Form: VX

04	vD	0_0000	vB	846
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i*2:(i*2)+7} ← SignExtend((vB)_i,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 half-word 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 **vD**.

A half word is unpacked to 32 bits by concatenating, in order, the results of the following operations.

- sign-extend bit 0 of the half word to 8 bits
- zero-extend bits 1–5 of the half word to 8 bits
- zero-extend bits 6–10 of the half word to 8 bits
- zero-extend bits 11–15 of the half word to 8 bits

Other registers altered:

- None

The source and target elements can be considered to be 16- and 32-bit ‘pixels,’ respectively, having the formats described in the programming note for the Vector Pack Pixel instruction.

Figure 6-161 shows the usage of the **vupkhpx** instruction. Each of the eight elements in the vector, **vB**, is 16 bits long. Each of the four elements in the vector, **vD**, is 32 bits long.

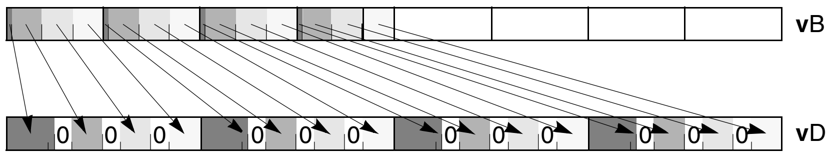


Figure 6-161. vupkhpx—Unpack High-Order Elements (16-Bit) to Elements (32-Bit)

vupkhsb vupkhsb

Vector Unpack High Signed Byte

vupkhsb vD,vB Form: VX

04	vD	0_0000	vB	526
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    vD_{i*2:(i*2)+15} ← SignExtend((vB)_{i:i+7},16)
end
```

Each signed integer byte element in the high-order half of register **v**B is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight half words of register **v**D.

Other registers altered:

- None

Figure 6-162 shows the usage of the **vupkhsb** instruction. Each of the sixteen elements in the vector, **v**B, is 8 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

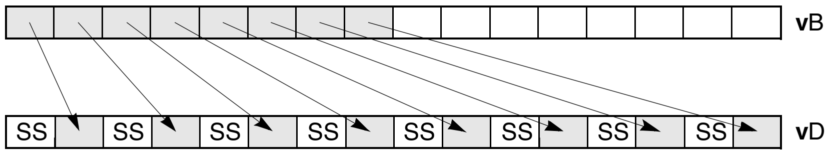


Figure 6-162. vupkhsb—Unpack High-Order Signed Integer Elements (8-Bit) to Signed Integer Elements (16-Bit)

vupkhsh   vupkhsh

Vector Unpack High Signed Half Word

vupkhsh vD,vB Form: VX

04	vD	0_0000	vB	590
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i*2:(i*2)+31} ← SignExtend((vB)_{i:i+15},32)
end
```

Each signed integer half-word element in the high-order half of register **v**B is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-163 shows the usage of the **vupkhsh** instruction. Each of the eight elements in the vector, **v**B, is 16 bits long. Each of the four elements in the vector, **v**D, is 32 bits long.

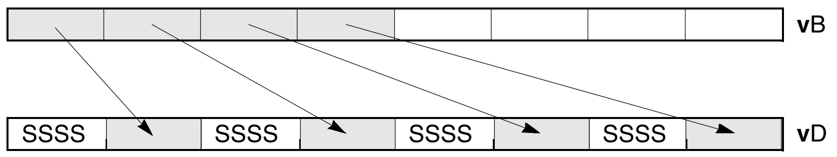


Figure 6-163. vupkhsh—Unpack Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

vupklpx vupklpx

Vector Unpack Low Pixel16

vupklpx vD,vB Form: VX

04	vD	0_0000	vB	974
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i*2:(i*2)+7} ← SignExtend((vB)_{i+64},8)
    vD_{(i*2)+8:(i*2)+15} ← ZeroExtend((vB)_{i+65:i+69},8)
    vD_{(i*2)+16:(i*2)+23} ← ZeroExtend((vB)_{i+70:i+74},8)
    vD_{(i*2)+24:(i*2)+31} ← ZeroExtend((vB)_{i+75:i+79},8)
end
```

Each half-word element in the low-order half of register **v**B is unpacked to produce a 32-bit value as described below and placed, in the same order, into the four words of register **v**D.

A half word is unpacked to 32 bits by concatenating, in order, the results of the following operations.

- sign-extend bit 0 of the half word to 8 bits
- zero-extend bits 1–5 of the half word to 8 bits
- zero-extend bits 6–10 of the half word to 8 bits
- zero-extend bits 11–15 of the half word to 8 bits

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 6-164 shows the usage of the **vupklpx** instruction. Each of the eight elements in the vector, **v**B, is 16 bits long. Each of the four elements in the vector, **v**D, is 32 bits long.

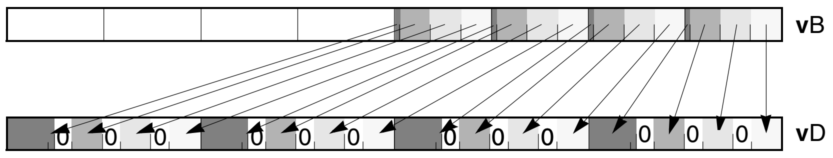


Figure 6-164. vupklpx—Unpack Low-Order Elements (16-Bit) to Elements (32-Bit)

vupklsb vupklsb

Vector Unpack Low Signed Byte

vupklsb vD,**v**B Form: VX

04	vD	0_0000	vB	654
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 8
    $\mathbf{v}D_{i*2:(i*2)+15} \leftarrow$ SignExtend(($\mathbf{v}$B)$_{i+64:i+71}$,16)
end
```

Each signed integer byte element in the low-order half of register **v**B is sign-extended to produce a 16-bit signed integer and placed, in the same order, into the eight half words of register **v**D.

Other registers altered:

- None

Figure 6-165 shows the usage of the **vupklsb** instruction. Each of the sixteen elements in the vector, **v**B, is 8 bits long. Each of the eight elements in the vector, **v**D, is 16 bits long.

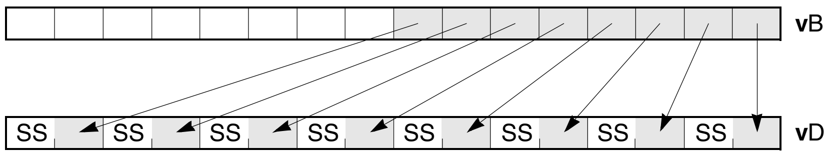


Figure 6-165. vupklsb—Unpack Low-Order Elements (8-Bit) to Elements (16-Bit)

vupklsh vupklsh

Vector Unpack Low Signed Half Word

vupklsh vD,**v**B Form: VX

04	vD	0_0000	vB	718
0 5	6 10	11 15	16 20	21 31

```
do i=0 to 63 by 16
    vD_{i*2:(i*2)+31} ← SignExtend((vB)_{i+64:i+79},32)
end
```

Each signed integer half word element in the low-order half of register **v**B is sign-extended to produce a 32-bit signed integer and placed, in the same order, into the four words of register **v**D.

Other registers altered:

- None

Figure 6-166 shows the usage of the **vupklsh** instruction. Each of the eight elements in the vector, **v**B, is 16 bits long. Each of the four elements in the vector, **v**D, is 32 bits long.

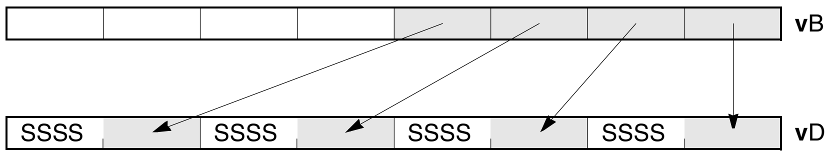


Figure 6-166. vupklsh—Unpack Low-Order Signed Integer Elements (16-Bit) to Signed Integer Elements (32-Bit)

vxor vxor

Vector Logical XOR

vxor vD,vA,vB Form: VX

04	vD	vA	vB	1220
0 5	6 10	11 15	16 20	21 31

```
vD ← (vA) ⊕ (vB)
```

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

Other registers altered:

- None

Figure 6-167 shows the usage of the **vxor** instruction.

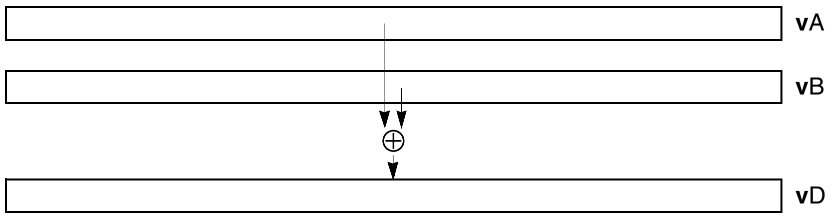


Figure 6-167. vxor—Bitwise XOR (128-Bit)

Appendix A
AltiVec Instruction Set Listings

This appendix lists the instruction set for AltiVec technology. Instructions are sorted by mnemonic, opcode, and form. Note that split fields, which represent the concatenation of sequences from left to right, are shown in lower case.

The following key applies to the tables in this appendix.

Key: ▨ Reserved Bits

A.1 Instructions Sorted by Mnemonic in Decimal Format

Table A-1 lists the instructions implemented in the AltiVec architecture in alphabetical order by mnemonic. The primary and extended opcodes are decimal numbers.

Table A-1. Instructions Sorted by Mnemonic in Decimal Format

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
dss	31	0	//	STRM	///	///	822	/
dssall	31	1	//	STRM	///	///	822	/
dst	31	0	//	STRM	A	B	342	/
dstst	31	0	//	STRM	A	B	374	/
dststt	31	1	//	STRM	A	B	374	/
dstt	31	1	//	STRM	A	B	342	/
lvebx	31		vD		A	B	7	/
lvehx	31		vD		A	B	39	/
lvepx[1]	31		vD		A	B	295	/
lvepxl[1]	31		vD		A	B	263	/
lvewx	31		vD		A	B	71	/
lvexbx	31		vD		A	B	261	/
lvexhx	31		vD		A	B	293	/
lvexwx	31		vD		A	B	325	/
lvsl	31		vD		A	B	6	/
lvsm	31		vD		A	B	773	/
lvsr	31		vD		A	B	38	/
lvswx	31		vD		A	B	613	/
lvswxl	31		vD		A	B	869	/
lvtlx	31		vD		A	B	581	/

Table A-1. Instructions Sorted by Mnemonic in Decimal Format (continued)

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
lvtlxl	31	vD	A	B	837	/
lvtrx	31	vD	A	B	549	/
lvtrxl	31	vD	A	B	805	/
lvx	31	vD	A	B	103	/
lvxl	31	vD	A	B	359	/
mfvscr	04	vD	///	///	1540	
mtvscr	04	///	///	vB	1604	
mvidsplt	31	vD	A	B	110	/
mviwsplt	31	vD	A	B	46	/
stvebx	31	vS	A	B	135	/
stvehx	31	vS	A	B	167	/
stvepx[1]	31	vD	A	B	807	/
stvepxl[1]	31	vD	A	B	775	/
stvewx	31	vS	A	B	199	/
stvexbx	31	vS	A	B	389	/
stvexhx	31	vS	A	B	421	/
stvexwx	31	vS	A	B	453	/
stvflx	31	vS	A	B	709	/
stvflxl	31	vS	A	B	965	/
stvfrx	31	vS	A	B	677	/
stvfrxl	31	vS	A	B	933	/
stvswx	31	vS	A	B	741	/
stvswxl	31	vS	A	B	997	/
stvx	31	vS	A	B	231	/
stvxl	31	vS	A	B	487	/
vabsdub	04	vD	vA	vB	1027	
vabsduh	04	vD	vA	vB	1091	
vabsduw	04	vD	vA	vB	1155	
vaddcuw	04	vD	vA	vB	384	
vaddfp	04	vD	vA	vB	10	
vaddsbs	04	vD	vA	vB	768	
vaddshs	04	vD	vA	vB	832	
vaddsws	04	vD	vA	vB	896	
vaddubm	04	vD	vA	vB	0	
vaddubs	04	vD	vA	vB	512	
vadduhm	04	vD	vA	vB	64	
vadduhs	04	vD	vA	vB	576	

Table A-1. Instructions Sorted by Mnemonic in Decimal Format (continued)

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
vadduwm	04	vD	vA	vB	128		
vadduws	04	vD	vA	vB	640		
vand	04	vD	vA	vB	1028		
vandc	04	vD	vA	vB	1092		
vavgsb	04	vD	vA	vB	1282		
vavgsh	04	vD	vA	vB	1346		
vavgsw	04	vD	vA	vB	1410		
vavgub	04	vD	vA	vB	1026		
vavguh	04	vD	vA	vB	1090		
vavguw	04	vD	vA	vB	1154		
vcfsx	04	vD	UIMM	vB	842		
vcfux	04	vD	UIMM	vB	778		
vcmpbfp*x*	04	vD	vA	vB	Rc	966	
vcmpeqfp*x*	04	vD	vA	vB	Rc	198	
vcmpequb*x*	04	vD	vA	vB	Rc	6	
vcmpequh*x*	04	vD	vA	vB	Rc	70	
vcmpequw*x*	04	vD	vA	vB	Rc	134	
vcmpgefp*x*	04	vD	vA	vB	Rc	454	
vcmpgtfp*x*	04	vD	vA	vB	Rc	710	
vcmpgtsb*x*	04	vD	vA	vB	Rc	774	
vcmpgtsh*x*	04	vD	vA	vB	Rc	838	
vcmpgtsw*x*	04	vD	vA	vB	Rc	902	
vcmpgtub*x*	04	vD	vA	vB	Rc	518	
vcmpgtuh*x*	04	vD	vA	vB	Rc	582	
vcmpgtuw*x*	04	vD	vA	vB	Rc	646	
vctsxs	04	vD	UIMM	vB	970		
vctuxs	04	vD	UIMM	vB	906		
vexptefp	04	vD	///	vB	394		
vlogefp	04	vD	///	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		

Table A-1. Instructions Sorted by Mnemonic in Decimal Format (continued)

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
vminfp	04	vD	vA	vB	1098	
vminsb	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	
vmladduhm	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
vmsumuhm	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
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	

Table A-1. Instructions Sorted by Mnemonic in Decimal Format (continued)

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
vpkswss	04	vD	vA				vB	462		
vpkswus	04	vD	vA				vB	334		
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	///				vB	266		
vrfim	04	vD	///				vB	714		
vrfin	04	vD	///				vB	522		
vrfip	04	vD	///				vB	650		
vrfiz	04	vD	///				vB	586		
vrlb	04	vD	vA				vB	4		
vrlh	04	vD	vA				vB	68		
vrlw	04	vD	vA				vB	132		
vrsqrtefp	04	vD	///				vB	330		
vsel	04	vD	vA				vB	vC		42
vsl	04	vD	vA				vB	452		
vslb	04	vD	vA				vB	260		
vsldoi	04	vD	vA				vB	/	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				///	780		
vspltish	04	vD	SIMM				///	844		
vspltisw	04	vD	SIMM				///	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		
vsrw	04	vD	vA				vB	644		
vsubcuw	04	vD	vA				vB	1408		

Table A-1. Instructions Sorted by Mnemonic in Decimal Format (continued)

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
vsubfp	04	vD	vA	vB	74
vsubsbs	04	vD	vA	vB	1792
vsubshs	04	vD	vA	vB	1856
vsubsws	04	vD	vA	vB	1920
vsububm	04	vD	vA	vB	1024
vsububs	04	vD	vA	vB	1536
vsubuhm	04	vD	vA	vB	1088
vsubuhs	04	vD	vA	vB	1600
vsubuwm	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	///	vB	846
vupkhsb	04	vD	///	vB	526
vupkhsh	04	vD	///	vB	590
vupklpx	04	vD	///	vB	974
vupklsb	04	vD	///	vB	654
vupklsh	04	vD	///	vB	718
vxor	04	vD	vA	vB	1220

[1] The **lvepx**, **lvepxl**, **stvepx**, and **stvepxl** instructions are implemented only if category E.PD is implemented.

A.2 Instructions Sorted by Mnemonic in Binary Format

Table A-2 lists the instructions implemented in the AltiVec architecture in alphabetical order by mnemonic. The primary and extended opcodes are decimal numbers.

Table A-2. Instructions Sorted by Mnemonic in Binary Format

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
dss	0111_11	0	//	STRM	///	///	110_0110_110	/
dssall	0111_11	1	//	STRM	///	///	110_0110_110	/
dst	0111_11	0	//	STRM	A	B	010_1010_110	/
dstst	0111_11	0	//	STRM	A	B	010_1110_110	/
dststt	0111_11	1	//	STRM	A	B	010_1110_110	/
dstt	0111_11	1	//	STRM	A	B	010_1010_110	/
lvebx	0111_11	vD			A	B	000_0000_111	/
lvehx	0111_11	vD			A	B	000_0100_111	/
lvepx[1]	0111_11	vD			A	B	010_0100_111	/
lvepxl[1]	0111_11	vD			A	B	010_0000_111	/
lvewx	0111_11	vD			A	B	000_1000_111	/
lvexbx	0111_11	vD			A	B	010_0000_101	/
lvexhx	0111_11	vD			A	B	010_0100_101	/
lvexwx	0111_11	vD			A	B	010_1000_101	/
lvsl	0111_11	vD			A	B	000_0000_110	/
lvsm	0111_11	vD			A	B	110_0000_101	/
lvsr	0111_11	vD			A	B	000_0100_110	/
lvswx	0111_11	vD			A	B	100_1100_101	/
lvswxl	0111_11	vD			A	B	110_1100_101	/
lvtlx	0111_11	vD			A	B	100_1000_101	/
lvtlxl	0111_11	vD			A	B	110_1000_101	/
lvtrx	0111_11	vD			A	B	100_0100_101	/
lvtrxl	0111_11	vD			A	B	110_0100_101	/
lvx	0111_11	vD			A	B	000_1100_111	/
lvxl	0111_11	vD			A	B	010_1100_111	/
mfvscr	0001_00	vD			///	///	110_0000_0100	
mtvscr	0001_00	///			///	vB	110_0100_0100	
mvidsplt	0111_11	vD			A	B	000_1101_110	/
mviwsplt	0111_11	vD			A	B	000_0101_110	/
stvebx	0111_11	vS			A	B	001_0000_111	/
stvehx	0111_11	vS			A	B	001_0100_111	/
stvepx[1]	0111_11	vD			A	B	110_0100_111	/
stvepxl[1]	0111_11	vD			A	B	110_0000_111	/
stvewx	0111_11	vS			A	B	001_1000_111	/

Table A-2. Instructions Sorted by Mnemonic in Binary Format (continued)

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
stvexbx	0111_11	vS	A	B	011_0000_101		/
stvexhx	0111_11	vS	A	B	011_0100_101		/
stvexwx	0111_11	vS	A	B	011_1000_101		/
stvflx	0111_11	vS	A	B	101_1000_101		/
stvflxl	0111_11	vS	A	B	111_1000_101		/
stvfrx	0111_11	vS	A	B	101_0100_101		/
stvfrxl	0111_11	vS	A	B	111_0100_101		/
stvswx	0111_11	vS	A	B	101_1100_101		/
stvswxl	0111_11	vS	A	B	111_1100_101		/
stvx	0111_11	vS	A	B	001_1100_111		/
stvxl	0111_11	vS	A	B	011_1100_111		/
vabsdub	0001_00	vD	vA	vB	100_0000_0011		
vabsduh	0001_00	vD	vA	vB	100_0100_0011		
vabsduw	0001_00	vD	vA	vB	100_1000_0011		
vaddcuw	0001_00	vD	vA	vB	001_1000_0000		
vaddfp	0001_00	vD	vA	vB	000_0000_1010		
vaddsbs	0001_00	vD	vA	vB	011_0000_0000		
vaddshs	0001_00	vD	vA	vB	011_0100_0000		
vaddsws	0001_00	vD	vA	vB	011_1000_0000		
vaddubm	0001_00	vD	vA	vB	000_0000_0000		
vaddubs	0001_00	vD	vA	vB	010_0000_0000		
vadduhm	0001_00	vD	vA	vB	000_0100_0000		
vadduhs	0001_00	vD	vA	vB	010_0100_0000		
vadduwm	0001_00	vD	vA	vB	000_1000_0000		
vadduws	0001_00	vD	vA	vB	010_1000_0000		
vand	0001_00	vD	vA	vB	100_0000_0100		
vandc	0001_00	vD	vA	vB	100_0100_0100		
vavgsb	0001_00	vD	vA	vB	101_0000_0010		
vavgsh	0001_00	vD	vA	vB	101_0100_0010		
vavgsw	0001_00	vD	vA	vB	101_1000_0010		
vavgub	0001_00	vD	vA	vB	100_0000_0010		
vavguh	0001_00	vD	vA	vB	100_0100_0010		
vavguw	0001_00	vD	vA	vB	100_1000_0010		
vcfsx	0001_00	vD	UIMM	vB	011_0100_1010		
vcfux	0001_00	vD	UIMM	vB	011_0000_1010		
vcmpbfp*x*	0001_00	vD	vA	vB	Rc	11_1100_0110	
vcmpeqfp*x*	0001_00	vD	vA	vB	Rc	00_1100_0110	

Table A-2. Instructions Sorted by Mnemonic in Binary Format (continued)

Name	0 5	6 10	11 15	16 20	21	22 31
vcmpequb*x*	0001_00	vD	vA	vB	Rc	00_0000_0110
vcmpequh*x*	0001_00	vD	vA	vB	Rc	00_0100_0110
vcmpequw*x*	0001_00	vD	vA	vB	Rc	00_1000_0110
vcmpgefp*x*	0001_00	vD	vA	vB	Rc	01_1100_0110
vcmpgtfp*x*	0001_00	vD	vA	vB	Rc	10_1100_0110
vcmpgtsb*x*	0001_00	vD	vA	vB	Rc	11_0000_0110
vcmpgtsh*x*	0001_00	vD	vA	vB	Rc	11_0100_0110
vcmpgtsw*x*	0001_00	vD	vA	vB	Rc	11_1000_0110
vcmpgtub*x*	0001_00	vD	vA	vB	Rc	10_0000_0110
vcmpgtuh*x*	0001_00	vD	vA	vB	Rc	10_0100_0110
vcmpgtuw*x*	0001_00	vD	vA	vB	Rc	10_1000_0110
vctsxs	0001_00	vD	UIMM	vB	011_1100_1010	
vctuxs	0001_00	vD	UIMM	vB	011_1000_1010	
vexptefp	0001_00	vD	///	vB	001_1000_1010	
vlogefp	0001_00	vD	///	vB	001_1100_1010	
vmaddfp	0001_00	vD	vA	vB	vC	10_1110
vmaxfp	0001_00	vD	vA	vB	100_0000_1010	
vmaxsb	0001_00	vD	vA	vB	001_0000_0010	
vmaxsh	0001_00	vD	vA	vB	001_0100_0010	
vmaxsw	0001_00	vD	vA	vB	001_1000_0010	
vmaxub	0001_00	vD	vA	vB	000_0000_0010	
vmaxuh	0001_00	vD	vA	vB	000_0100_0010	
vmaxuw	0001_00	vD	vA	vB	000_1000_0010	
vmhaddshs	0001_00	vD	vA	vB	vC	10_0000
vmhraddshs	0001_00	vD	vA	vB	vC	10_0001
vminfp	0001_00	vD	vA	vB	100_0100_1010	
vminsb	0001_00	vD	vA	vB	011_0000_0010	
vminsh	0001_00	vD	vA	vB	011_0100_0010	
vminsw	0001_00	vD	vA	vB	011_1000_0010	
vminub	0001_00	vD	vA	vB	010_0000_0010	
vminuh	0001_00	vD	vA	vB	010_0100_0010	
vminuw	0001_00	vD	vA	vB	010_1000_0010	
vmladduhm	0001_00	vD	vA	vB	vC	10_0010
vmrghb	0001_00	vD	vA	vB	000_0000_1100	
vmrghh	0001_00	vD	vA	vB	000_0100_1100	
vmrghw	0001_00	vD	vA	vB	000_1000_1100	
vmrglb	0001_00	vD	vA	vB	001_0000_1100	

Table A-2. Instructions Sorted by Mnemonic in Binary Format (continued)

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
vmrglh	0001_00	vD	vA	vB	001_0100_1100	
vmrglw	0001_00	vD	vA	vB	001_1000_1100	
vmsummbm	0001_00	vD	vA	vB	vC	10_0101
vmsumshm	0001_00	vD	vA	vB	vC	10_1000
vmsumshs	0001_00	vD	vA	vB	vC	10_1001
vmsumubm	0001_00	vD	vA	vB	vC	10_0100
vmsumuhm	0001_00	vD	vA	vB	vC	10_0110
vmsumuhs	0001_00	vD	vA	vB	vC	10_0111
vmulesb	0001_00	vD	vA	vB	011_0000_1000	
vmulesh	0001_00	vD	vA	vB	011_0100_1000	
vmuleub	0001_00	vD	vA	vB	010_0000_1000	
vmuleuh	0001_00	vD	vA	vB	010_0100_1000	
vmulosb	0001_00	vD	vA	vB	001_0000_1000	
vmulosh	0001_00	vD	vA	vB	001_0100_1000	
vmuloub	0001_00	vD	vA	vB	000_0000_1000	
vmulouh	0001_00	vD	vA	vB	000_0100_1000	
vnmsubfp	0001_00	vD	vA	vB	vC	10_1111
vnor	0001_00	vD	vA	vB	101_0000_0100	
vor	0001_00	vD	vA	vB	100_1000_0100	
vperm	0001_00	vD	vA	vB	vC	10_1011
vpkpx	0001_00	vD	vA	vB	011_0000_1110	
vpkshss	0001_00	vD	vA	vB	001_1000_1110	
vpkshus	0001_00	vD	vA	vB	001_0000_1110	
vpkswss	0001_00	vD	vA	vB	001_1100_1110	
vpkswus	0001_00	vD	vA	vB	001_0100_1110	
vpkuhum	0001_00	vD	vA	vB	000_0000_1110	
vpkuhus	0001_00	vD	vA	vB	000_1000_1110	
vpkuwum	0001_00	vD	vA	vB	000_0100_1110	
vpkuwus	0001_00	vD	vA	vB	000_1100_1110	
vrefp	0001_00	vD	///	vB	001_0000_1010	
vrfim	0001_00	vD	///	vB	010_1100_1010	
vrfin	0001_00	vD	///	vB	010_0000_1010	
vrfip	0001_00	vD	///	vB	010_1000_1010	
vrfiz	0001_00	vD	///	vB	010_0100_1010	
vrlb	0001_00	vD	vA	vB	000_0000_0100	
vrlh	0001_00	vD	vA	vB	000_0100_0100	
vrlw	0001_00	vD	vA	vB	000_1000_0100	

Table A-2. Instructions Sorted by Mnemonic in Binary Format (continued)

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
vrsqrtefp	0001_00	vD	///	vB	001_0100_1010
vsel	0001_00	vD	vA	vB	vC 10_1010
vsl	0001_00	vD	vA	vB	001_1100_0100
vslb	0001_00	vD	vA	vB	001_0000_0100
vsldoi	0001_00	vD	vA	vB	/ SH 10_1100
vslh	0001_00	vD	vA	vB	001_0100_0100
vslo	0001_00	vD	vA	vB	100_0000_1100
vslw	0001_00	vD	vA	vB	001_1000_0100
vspltb	0001_00	vD	/ UIMM	vB	010_0000_1100
vsplth	0001_00	vD	// UIMM	vB	010_0100_1100
vspltisb	0001_00	vD	SIMM	///	011_0000_1100
vspltish	0001_00	vD	SIMM	///	011_0100_1100
vspltisw	0001_00	vD	SIMM	///	011_1000_1100
vspltw	0001_00	vD	/// UIMM	vB	010_1000_1100
vsr	0001_00	vD	vA	vB	010_1100_0100
vsrab	0001_00	vD	vA	vB	011_0000_0100
vsrah	0001_00	vD	vA	vB	011_0100_0100
vsraw	0001_00	vD	vA	vB	011_1000_0100
vsrb	0001_00	vD	vA	vB	010_0000_0100
vsrh	0001_00	vD	vA	vB	010_0100_0100
vsro	0001_00	vD	vA	vB	100_0100_1100
vsrw	0001_00	vD	vA	vB	010_1000_0100
vsubcuw	0001_00	vD	vA	vB	101_1000_0000
vsubfp	0001_00	vD	vA	vB	000_0100_1010
vsubsbs	0001_00	vD	vA	vB	111_0000_0000
vsubshs	0001_00	vD	vA	vB	111_0100_0000
vsubsws	0001_00	vD	vA	vB	111_1000_0000
vsububm	0001_00	vD	vA	vB	100_0000_0000
vsububs	0001_00	vD	vA	vB	110_0000_0000
vsubuhm	0001_00	vD	vA	vB	100_0100_0000
vsubuhs	0001_00	vD	vA	vB	110_0100_0000
vsubuwm	0001_00	vD	vA	vB	100_1000_0000
vsubuws	0001_00	vD	vA	vB	110_1000_0000
vsumsws	0001_00	vD	vA	vB	111_1000_1000
vsum2sws	0001_00	vD	vA	vB	110_1000_1000
vsum4sbs	0001_00	vD	vA	vB	111_0000_1000
vsum4shs	0001_00	vD	vA	vB	110_0100_1000

Table A-2. Instructions Sorted by Mnemonic in Binary Format (continued)

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
vsum4ubs	0001_00	vD	vA	vB	110_0000_1000
vupkhpx	0001_00	vD	///	vB	011_0100_1110
vupkhsb	0001_00	vD	///	vB	010_0000_1110
vupkhsh	0001_00	vD	///	vB	010_0100_1110
vupklpx	0001_00	vD	///	vB	011_1100_1110
vupklsb	0001_00	vD	///	vB	010_1000_1110
vupklsh	0001_00	vD	///	vB	010_1100_1110
vxor	0001_00	vD	vA	vB	100_1100_0100

1 The **lvepx**, **lvepxl**, **stvepx**, and **stvepxl** instructions are implemented only if category E.PD is implemented.

A.3 Instructions Sorted by Opcode in Decimal Format

Table A-3 lists AltiVec instructions grouped by opcode in decimal format.

Table A-3. Instructions Sorted by Opcode in Decimal Format

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
vaddubm	04	vD	vA	vB	0
vmaxub	04	vD	vA	vB	2
vrlb	04	vD	vA	vB	4
vmuloub	04	vD	vA	vB	8
vaddfp	04	vD	vA	vB	10
vmrghb	04	vD	vA	vB	12
vpkuhum	04	vD	vA	vB	14
vadduhm	04	vD	vA	vB	64
vmaxuh	04	vD	vA	vB	66
vrlh	04	vD	vA	vB	68
vmulouh	04	vD	vA	vB	72
vsubfp	04	vD	vA	vB	74
vmrghh	04	vD	vA	vB	76
vpkuwum	04	vD	vA	vB	78
vadduwm	04	vD	vA	vB	128
vmaxuw	04	vD	vA	vB	130
vrlw	04	vD	vA	vB	132
vmrghw	04	vD	vA	vB	140
vpkuhus	04	vD	vA	vB	142
vpkuwus	04	vD	vA	vB	206
vmaxsb	04	vD	vA	vB	258
vslb	04	vD	vA	vB	260
vmulosb	04	vD	vA	vB	264
vrefp	04	vD	///	vB	266
vmrglb	04	vD	vA	vB	268
vpkshus	04	vD	vA	vB	270
vmaxsh	04	vD	vA	vB	322
vslh	04	vD	vA	vB	324
vmulosh	04	vD	vA	vB	328
vrsqrtefp	04	vD	///	vB	330
vmrglh	04	vD	vA	vB	332
vpkswus	04	vD	vA	vB	334
vaddcuw	04	vD	vA	vB	384
vmaxsw	04	vD	vA	vB	386

Table A-3. Instructions Sorted by Opcode in Decimal Format (continued)

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
vslw	04	vD	vA	vB	388
vexptefp	04	vD	///	vB	394
vmrglw	04	vD	vA	vB	396
vpkshss	04	vD	vA	vB	398
vsl	04	vD	vA	vB	452
vlogefp	04	vD	///	vB	458
vpkswss	04	vD	vA	vB	462
vaddubs	04	vD	vA	vB	512
vminub	04	vD	vA	vB	514
vsrb	04	vD	vA	vB	516
vmuleub	04	vD	vA	vB	520
vrfin	04	vD	///	vB	522
vspltb	04	vD	/ UIMM	vB	524
vupkhsb	04	vD	///	vB	526
vadduhs	04	vD	vA	vB	576
vminuh	04	vD	vA	vB	578
vsrh	04	vD	vA	vB	580
vmuleuh	04	vD	vA	vB	584
vrfiz	04	vD	///	vB	586
vsplth	04	vD	// UIMM	vB	588
vupkhsh	04	vD	///	vB	590
vadduws	04	vD	vA	vB	640
vminuw	04	vD	vA	vB	642
vsrw	04	vD	vA	vB	644
vrfip	04	vD	///	vB	650
vspltw	04	vD	/// UIMM	vB	652
vupklsb	04	vD	///	vB	654
vsr	04	vD	vA	vB	708
vrfim	04	vD	///	vB	714
vupklsh	04	vD	///	vB	718
vaddsbs	04	vD	vA	vB	768
vminsb	04	vD	vA	vB	770
vsrab	04	vD	vA	vB	772
vmulesb	04	vD	vA	vB	776
vcfux	04	vD	UIMM	vB	778
vspltisb	04	vD	SIMM	///	780
vpkpx	04	vD	vA	vB	782

Table A-3. Instructions Sorted by Opcode in Decimal Format (continued)

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
vaddshs	04	vD	vA	vB	832
vminsh	04	vD	vA	vB	834
vsrah	04	vD	vA	vB	836
vmulesh	04	vD	vA	vB	840
vcfsx	04	vD	UIMM	vB	842
vspltish	04	vD	SIMM	///	844
vupkhpx	04	vD	///	vB	846
vaddsws	04	vD	vA	vB	896
vminsw	04	vD	vA	vB	898
vsraw	04	vD	vA	vB	900
vctuxs	04	vD	UIMM	vB	906
vspltisw	04	vD	SIMM	///	908
vctsxs	04	vD	UIMM	vB	970
vupklpx	04	vD	///	vB	974
vsububm	04	vD	vA	vB	1024
vavgub	04	vD	vA	vB	1026
vabsdub	04	vD	vA	vB	1027
vand	04	vD	vA	vB	1028
vmaxfp	04	vD	vA	vB	1034
vslo	04	vD	vA	vB	1036
vsubuhm	04	vD	vA	vB	1088
vavguh	04	vD	vA	vB	1090
vabsduh	04	vD	vA	vB	1091
vandc	04	vD	vA	vB	1092
vminfp	04	vD	vA	vB	1098
vsro	04	vD	vA	vB	1100
vsubuwm	04	vD	vA	vB	1152
vavguw	04	vD	vA	vB	1154
vabsduw	04	vD	vA	vB	1155
vor	04	vD	vA	vB	1156
vxor	04	vD	vA	vB	1220
vavgsb	04	vD	vA	vB	1282
vnor	04	vD	vA	vB	1284
vavgsh	04	vD	vA	vB	1346
vsubcuw	04	vD	vA	vB	1408
vavgsw	04	vD	vA	vB	1410
vsububs	04	vD	vA	vB	1536

Table A-3. Instructions Sorted by Opcode in Decimal Format (continued)

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
mfvscr	04	vD	///	///	1540		
vsum4ubs	04	vD	vA	vB	1544		
vsubuhs	04	vD	vA	vB	1600		
mtvscr	04	///	///	vB	1604		
vsum4shs	04	vD	vA	vB	1608		
vsubuws	04	vD	vA	vB	1664		
vsum2sws	04	vD	vA	vB	1672		
vsubsbs	04	vD	vA	vB	1792		
vsum4sbs	04	vD	vA	vB	1800		
vsubshs	04	vD	vA	vB	1856		
vsubsws	04	vD	vA	vB	1920		
vsumsws	04	vD	vA	vB	1928		
vcmpequb*x*	04	vD	vA	vB	Rc	6	
vcmpequh*x*	04	vD	vA	vB	Rc	70	
vcmpequw*x*	04	vD	vA	vB	Rc	134	
vcmpeqfp*x*	04	vD	vA	vB	Rc	198	
vcmpgefp*x*	04	vD	vA	vB	Rc	454	
vcmpgtub*x*	04	vD	vA	vB	Rc	518	
vcmpgtuh*x*	04	vD	vA	vB	Rc	582	
vcmpgtuw*x*	04	vD	vA	vB	Rc	646	
vcmpgtfp*x*	04	vD	vA	vB	Rc	710	
vcmpgtsb*x*	04	vD	vA	vB	Rc	774	
vcmpgtsh*x*	04	vD	vA	vB	Rc	838	
vcmpgtsw*x*	04	vD	vA	vB	Rc	902	
vcmpbfp*x*	04	vD	vA	vB	Rc	966	
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	/	SH	44

Table A-3. Instructions Sorted by Opcode in Decimal Format (continued)

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
vmaddfp	04	vD			vA	vB	vC	46	
vnmsubfp	04	vD			vA	vB	vC	47	
lvsl	31	vD			A	B	6		/
lvebx	31	vD			A	B	7		/
lvsr	31	vD			A	B	38		/
lvehx	31	vD			A	B	39		/
mviwsplt	31	vD			A	B	46		/
lvewx	31	vD			A	B	71		/
lvx	31	vD			A	B	103		/
mvidsplt	31	vD			A	B	110		/
stvebx	31	vS			A	B	135		/
stvehx	31	vS			A	B	167		/
stvewx	31	vS			A	B	199		/
stvx	31	vS			A	B	231		/
lvexbx	31	vD			A	B	261		/
lvepxl[1]	31	vD			A	B	263		/
lvexhx	31	vD			A	B	293		/
lvepx[1]	31	vD			A	B	295		/
lvexwx	31	vD			A	B	325		/
dst	31	0	//	STRM	A	B	342		/
dstt	31	1	//	STRM	A	B	342		/
lvxl	31	vD			A	B	359		/
dstst	31	0	//	STRM	A	B	374		/
dststt	31	1	//	STRM	A	B	374		/
stvexbx	31	vS			A	B	389		/
stvexhx	31	vS			A	B	421		/
stvexwx	31	vS			A	B	453		/
stvxl	31	vS			A	B	487		/
lvtrx	31	vD			A	B	549		/
lvtlx	31	vD			A	B	581		/
lvswx	31	vD			A	B	613		/
stvfrx	31	vS			A	B	677		/
stvflx	31	vS			A	B	709		/
stvswx	31	vS			A	B	741		/
lvsm	31	vD			A	B	773		/
stvepxl[1]	31	vD			A	B	775		/
lvtrxl	31	vD			A	B	805		/

Table A-3. Instructions Sorted by Opcode in Decimal Format (continued)

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
stvepx[1]	31	vD					A					B					807										/
dss	31	0	//		STRM		///					///					822										/
dssall	31	1	//		STRM		///					///					822										/
lvtlxl	31	vD					A					B					837										/
lvswxl	31	vD					A					B					869										/
stvfrxl	31	vS					A					B					933										/
stvflxl	31	vS					A					B					965										/
stvswxl	31	vS					A					B					997										/

[1] The **lvepx**, **lvepxl**, **stvepx**, and **stvepxl** instructions are implemented only if category E.PD is implemented.

A.4 Instructions Sorted by Opcode in Binary Format

Table A-4 lists Altivec instructions grouped by opcode in binary format.

Table A-4. Instructions Sorted by Opcode in Binary Format

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
vaddubm	0001_00	vD	vA	vB	000_0000_0000
vmaxub	0001_00	vD	vA	vB	000_0000_0010
vrlb	0001_00	vD	vA	vB	000_0000_0100
vmuloub	0001_00	vD	vA	vB	000_0000_1000
vaddfp	0001_00	vD	vA	vB	000_0000_1010
vmrghb	0001_00	vD	vA	vB	000_0000_1100
vpkuhum	0001_00	vD	vA	vB	000_0000_1110
vadduhm	0001_00	vD	vA	vB	000_0100_0000
vmaxuh	0001_00	vD	vA	vB	000_0100_0010
vrlh	0001_00	vD	vA	vB	000_0100_0100
vmulouh	0001_00	vD	vA	vB	000_0100_1000
vsubfp	0001_00	vD	vA	vB	000_0100_1010
vmrghh	0001_00	vD	vA	vB	000_0100_1100
vpkuwum	0001_00	vD	vA	vB	000_0100_1110
vadduwm	0001_00	vD	vA	vB	000_1000_0000
vmaxuw	0001_00	vD	vA	vB	000_1000_0010
vrlw	0001_00	vD	vA	vB	000_1000_0100
vmrghw	0001_00	vD	vA	vB	000_1000_1100
vpkuhus	0001_00	vD	vA	vB	000_1000_1110
vpkuwus	0001_00	vD	vA	vB	000_1100_1110
vmaxsb	0001_00	vD	vA	vB	001_0000_0010
vslb	0001_00	vD	vA	vB	001_0000_0100
vmulosb	0001_00	vD	vA	vB	001_0000_1000
vrefp	0001_00	vD	///	vB	001_0000_1010
vmrglb	0001_00	vD	vA	vB	001_0000_1100
vpkshus	0001_00	vD	vA	vB	001_0000_1110
vmaxsh	0001_00	vD	vA	vB	001_0100_0010
vslh	0001_00	vD	vA	vB	001_0100_0100
vmulosh	0001_00	vD	vA	vB	001_0100_1000
vrsqrtefp	0001_00	vD	///	vB	001_0100_1010
vmrglh	0001_00	vD	vA	vB	001_0100_1100
vpkswus	0001_00	vD	vA	vB	001_0100_1110
vaddcuw	0001_00	vD	vA	vB	001_1000_0000
vmaxsw	0001_00	vD	vA	vB	001_1000_0010

Table A-4. Instructions Sorted by Opcode in Binary Format (continued)

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
vslw	0001_00	vD	vA	vB	001_1000_0100
vexptefp	0001_00	vD	///	vB	001_1000_1010
vmrglw	0001_00	vD	vA	vB	001_1000_1100
vpkshss	0001_00	vD	vA	vB	001_1000_1110
vsl	0001_00	vD	vA	vB	001_1100_0100
vlogefp	0001_00	vD	///	vB	001_1100_1010
vpkswss	0001_00	vD	vA	vB	001_1100_1110
vaddubs	0001_00	vD	vA	vB	010_0000_0000
vminub	0001_00	vD	vA	vB	010_0000_0010
vsrb	0001_00	vD	vA	vB	010_0000_0100
vmuleub	0001_00	vD	vA	vB	010_0000_1000
vrfin	0001_00	vD	///	vB	010_0000_1010
vspltb	0001_00	vD	/ UIMM	vB	010_0000_1100
vupkhsb	0001_00	vD	///	vB	010_0000_1110
vadduhs	0001_00	vD	vA	vB	010_0100_0000
vminuh	0001_00	vD	vA	vB	010_0100_0010
vsrh	0001_00	vD	vA	vB	010_0100_0100
vmuleuh	0001_00	vD	vA	vB	010_0100_1000
vrfiz	0001_00	vD	///	vB	010_0100_1010
vsplth	0001_00	vD	// UIMM	vB	010_0100_1100
vupkhsh	0001_00	vD	///	vB	010_0100_1110
vadduws	0001_00	vD	vA	vB	010_1000_0000
vminuw	0001_00	vD	vA	vB	010_1000_0010
vsrw	0001_00	vD	vA	vB	010_1000_0100
vrfip	0001_00	vD	///	vB	010_1000_1010
vspltw	0001_00	vD	/// UIMM	vB	010_1000_1100
vupklsb	0001_00	vD	///	vB	010_1000_1110
vsr	0001_00	vD	vA	vB	010_1100_0100
vrfim	0001_00	vD	///	vB	010_1100_1010
vupklsh	0001_00	vD	///	vB	010_1100_1110
vaddsbs	0001_00	vD	vA	vB	011_0000_0000
vminsb	0001_00	vD	vA	vB	011_0000_0010
vsrab	0001_00	vD	vA	vB	011_0000_0100
vmulesb	0001_00	vD	vA	vB	011_0000_1000
vcfux	0001_00	vD	UIMM	vB	011_0000_1010
vspltisb	0001_00	vD	SIMM	///	011_0000_1100
vpkpx	0001_00	vD	vA	vB	011_0000_1110

Table A-4. Instructions Sorted by Opcode in Binary Format (continued)

Name	0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
vaddshs	0001_00	vD	vA	vB	011_0100_0000
vminsh	0001_00	vD	vA	vB	011_0100_0010
vsrah	0001_00	vD	vA	vB	011_0100_0100
vmulesh	0001_00	vD	vA	vB	011_0100_1000
vcfsx	0001_00	vD	UIMM	vB	011_0100_1010
vspltish	0001_00	vD	SIMM	///	011_0100_1100
vupkhpx	0001_00	vD	///	vB	011_0100_1110
vaddsws	0001_00	vD	vA	vB	011_1000_0000
vminsw	0001_00	vD	vA	vB	011_1000_0010
vsraw	0001_00	vD	vA	vB	011_1000_0100
vctuxs	0001_00	vD	UIMM	vB	011_1000_1010
vspltisw	0001_00	vD	SIMM	///	011_1000_1100
vctsxs	0001_00	vD	UIMM	vB	011_1100_1010
vupklpx	0001_00	vD	///	vB	011_1100_1110
vsububm	0001_00	vD	vA	vB	100_0000_0000
vavgub	0001_00	vD	vA	vB	100_0000_0010
vabsdub	0001_00	vD	vA	vB	100_0000_0011
vand	0001_00	vD	vA	vB	100_0000_0100
vmaxfp	0001_00	vD	vA	vB	100_0000_1010
vslo	0001_00	vD	vA	vB	100_0000_1100
vsubuhm	0001_00	vD	vA	vB	100_0100_0000
vavguh	0001_00	vD	vA	vB	100_0100_0010
vabsduh	0001_00	vD	vA	vB	100_0100_0011
vandc	0001_00	vD	vA	vB	100_0100_0100
vminfp	0001_00	vD	vA	vB	100_0100_1010
vsro	0001_00	vD	vA	vB	100_0100_1100
vsubuwm	0001_00	vD	vA	vB	100_1000_0000
vavguw	0001_00	vD	vA	vB	100_1000_0010
vabsduw	0001_00	vD	vA	vB	100_1000_0011
vor	0001_00	vD	vA	vB	100_1000_0100
vxor	0001_00	vD	vA	vB	100_1100_0100
vavgsb	0001_00	vD	vA	vB	101_0000_0010
vnor	0001_00	vD	vA	vB	101_0000_0100
vavgsh	0001_00	vD	vA	vB	101_0100_0010
vsubcuw	0001_00	vD	vA	vB	101_1000_0000
vavgsw	0001_00	vD	vA	vB	101_1000_0010
vsububs	0001_00	vD	vA	vB	110_0000_0000

Table A-4. Instructions Sorted by Opcode in Binary Format (continued)

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
mfvscr	0001_00	vD	///	///	110_0000_0100		
vsum4ubs	0001_00	vD	vA	vB	110_0000_1000		
vsubuhs	0001_00	vD	vA	vB	110_0100_0000		
mtvscr	0001_00	///	///	vB	110_0100_0100		
vsum4shs	0001_00	vD	vA	vB	110_0100_1000		
vsubuws	0001_00	vD	vA	vB	110_1000_0000		
vsum2sws	0001_00	vD	vA	vB	110_1000_1000		
vsubsbs	0001_00	vD	vA	vB	111_0000_0000		
vsum4sbs	0001_00	vD	vA	vB	111_0000_1000		
vsubshs	0001_00	vD	vA	vB	111_0100_0000		
vsubsws	0001_00	vD	vA	vB	111_1000_0000		
vsumsws	0001_00	vD	vA	vB	111_1000_1000		
vcmpbfp*x*	0001_00	vD	vA	vB	Rc	11_1100_0110	
vcmpeqfp*x*	0001_00	vD	vA	vB	Rc	00_1100_0110	
vcmpequb*x*	0001_00	vD	vA	vB	Rc	00_0000_0110	
vcmpequh*x*	0001_00	vD	vA	vB	Rc	00_0100_0110	
vcmpequw*x*	0001_00	vD	vA	vB	Rc	00_1000_0110	
vcmpgefp*x*	0001_00	vD	vA	vB	Rc	01_1100_0110	
vcmpgtfp*x*	0001_00	vD	vA	vB	Rc	10_1100_0110	
vcmpgtsb*x*	0001_00	vD	vA	vB	Rc	11_0000_0110	
vcmpgtsh*x*	0001_00	vD	vA	vB	Rc	11_0100_0110	
vcmpgtsw*x*	0001_00	vD	vA	vB	Rc	11_1000_0110	
vcmpgtub*x*	0001_00	vD	vA	vB	Rc	10_0000_0110	
vcmpgtuh*x*	0001_00	vD	vA	vB	Rc	10_0100_0110	
vcmpgtuw*x*	0001_00	vD	vA	vB	Rc	10_1000_0110	
vmhaddshs	0001_00	vD	vA	vB	vC		10_0000
vmhraddshs	0001_00	vD	vA	vB	vC		10_0001
vmladduhm	0001_00	vD	vA	vB	vC		10_0010
vmsumubm	0001_00	vD	vA	vB	vC		10_0100
vmsummbm	0001_00	vD	vA	vB	vC		10_0101
vmsumuhm	0001_00	vD	vA	vB	vC		10_0110
vmsumuhs	0001_00	vD	vA	vB	vC		10_0111
vmsumshm	0001_00	vD	vA	vB	vC		10_1000
vmsumshs	0001_00	vD	vA	vB	vC		10_1001
vsel	0001_00	vD	vA	vB	vC		10_1010
vperm	0001_00	vD	vA	vB	vC		10_1011
vsldoi	0001_00	vD	vA	vB	/	SH	10_1100

Table A-4. Instructions Sorted by Opcode in Binary Format (continued)

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
vmaddfp	0001_00	vD			vA	vB	vC	10_1110	
vnmsubfp	0001_00	vD			vA	vB	vC	10_1111	
lvsl	0111_11	vD			A	B	000_0000_110		/
lvebx	0111_11	vD			A	B	000_0000_111		/
lvsr	0111_11	vD			A	B	000_0100_110		/
lvehx	0111_11	vD			A	B	000_0100_111		/
mviwsplt	0111_11	vD			A	B	000_0101_110		/
lvewx	0111_11	vD			A	B	000_1000_111		/
lvx	0111_11	vD			A	B	000_1100_111		/
mvidsplt	0111_11	vD			A	B	000_1101_110		/
stvebx	0111_11	vS			A	B	001_0000_111		/
stvehx	0111_11	vS			A	B	001_0100_111		/
stvewx	0111_11	vS			A	B	001_1000_111		/
stvx	0111_11	vS			A	B	001_1100_111		/
lvexbx	0111_11	vD			A	B	010_0000_101		/
lvepxl[1]	0111_11	vD			A	B	010_0000_111		/
lvexhx	0111_11	vD			A	B	010_0100_101		/
lvepx[1]	0111_11	vD			A	B	010_0100_111		/
lvexwx	0111_11	vD			A	B	010_1000_101		/
dst	0111_11	0	//	STRM	A	B	010_1010_110		/
dstt	0111_11	1	//	STRM	A	B	010_1010_110		/
lvxl	0111_11	vD			A	B	010_1100_111		/
dstst	0111_11	0	//	STRM	A	B	010_1110_110		/
dststt	0111_11	1	//	STRM	A	B	010_1110_110		/
stvexbx	0111_11	vS			A	B	011_0000_101		/
stvexhx	0111_11	vS			A	B	011_0100_101		/
stvexwx	0111_11	vS			A	B	011_1000_101		/
stvxl	0111_11	vS			A	B	011_1100_111		/
lvtrx	0111_11	vD			A	B	100_0100_101		/
lvtlx	0111_11	vD			A	B	100_1000_101		/
lvswx	0111_11	vD			A	B	100_1100_101		/
stvfrx	0111_11	vS			A	B	101_0100_101		/
stvflx	0111_11	vS			A	B	101_1000_101		/
stvswx	0111_11	vS			A	B	101_1100_101		/
lvsm	0111_11	vD			A	B	110_0000_101		/
stvepxl[1]	0111_11	vD			A	B	110_0000_111		/
lvtrxl	0111_11	vD			A	B	110_0100_101		/

Table A-4. Instructions Sorted by Opcode in Binary Format (continued)

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
stvepx[1]	0111_11	vD			A	B	110_0100_111	/
dss	0111_11	0	//	STRM	///	///	110_0110_110	/
dssall	0111_11	1	//	STRM	///	///	110_0110_110	/
lvtlxl	0111_11	vD			A	B	110_1000_101	/
lvswxl	0111_11	vD			A	B	110_1100_101	/
stvfrxl	0111_11	vS			A	B	111_0100_101	/
stvflxl	0111_11	vS			A	B	111_1000_101	/
stvswxl	0111_11	vS			A	B	111_1100_101	/

[1] The **lvepx**, **lvepxl**, **stvepx**, and **stvepxl** instructions are implemented only if category E.PD is implemented.

A.5 Instructions Sorted by Form

Table A-5 through Table A-8 list the AltiVec instructions grouped by form.

Table A-5. VA-Form

OPCD	vD	vA	vB	vC3.5		XO
OPCD	vD	vA	vB	/	SH	XO

Specific Instructions

Name	0–5	6–10	11–15	16–20	21	22–25	26–31
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	/	SH	44
vmaddfp	04	vD	vA	vB	vC		46
vnmsubfp	04	vD	vA	vB	vC		47

Table A-6. VX-Form

OPCD	vD	vA	vB	XO	
OPCD	vD	///	///	XO	/
OPCD	///	///	vB	XO	/
OPCD	vD	///	vB	XO	
OPCD	vD	UIMM	vB	XO	
OPCD	vD	SIMM	///	XO	

Specific Instructions

Name	0–5	6–10	11–15	16–20	21–31
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

Table A-6. VX-Form (continued)

Name	0–5	6–10	11–15	16–20	21–31
vaddshs	04	vD	vA	vB	832
	Specific Instructions				
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
vminuw	04	vD	vA	vB	642
vminsb	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
vavgsb	04	vD	vA	vB	1282
vavgsh	04	vD	vA	vB	1346
vavgsw	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

Table A-6. VX-Form (continued)

vsrw	04	vD	vA	vB	644
	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
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	///	///	1540
mtvscr	04	///	///	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
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	///	vB	266
vrsqrtefp	04	vD	///	vB	330
vexptefp	04	vD	///	vB	394
vlogefp	04	vD	///	vB	458
vrfin	04	vD	///	vB	522
vrfiz	04	vD	///	vB	586
vrfip	04	vD	///	vB	650
vrfim	04	vD	///	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

Table A-6. VX-Form (continued)

vminfp	04	vD	vA	vB	1098

Specific Instructions

Name	0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
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	///	780
vspltish	04	vD	SIMM	///	844
vspltisw	04	vD	SIMM	///	908
vslo	04	vD	vA	vB	1036
vsro	04	vD	vA	vB	1100
vpkuhum	04	vD	vA	vB	14
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	///	vB	526
vupkhsh	04	vD	///	vB	590
vupklsb	04	vD	///	vB	654
vupklsh	04	vD	///	vB	718
vpkpx	04	vD	vA	vB	782
vupkhpx	04	vD	///	vB	846
vupklpx	04	vD	///	vB	974
vxor	04	vD	vA	vB	1220
vabsdub	04	vD	vA	vB	1027
vabsduh	04	vD	vA	vB	1091
vabsduw	04	vD	vA	vB	1155

Table A-7. X-Form

OPCD	vD	vA	vB	XO	/
OPCD	vS	vA	vB	XO	/
OPCD	T // STRM	A	B	XO	/

Table A-7. X-Form (continued)

	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
dst	31	T	//	STRM	A	B	342	/
dstt	31	1	//	STRM	A	B	342	/
dstst	31	T	//	STRM	A	B	374	/
dststt	31	1	//	STRM	A	B	374	/
dss	31	A	//	STRM	///	///	822	/
dssall	31	1	//	STRM	///	///	822	/
lvebx	31	vD			A	B	7	/
lvehx	31	vD			A	B	39	/
lvewx	31	vD			A	B	71	/
lvexbx	31	vD			A	B	261	/
lvepxl	31	vD			A	B	263	/
lvexhx	31	vD			A	B	293	/
lvepx	31	vD			A	B	295	/
lvexwx	31	vD			A	B	325	/
lvsm	31	vD			A	B	773	
lvswx	31	vD			A	B	613	
lvswxl	31	vD			A	B	869	
lvsl	31	vD			A	B	6	/
lvsr	31	vD			A	B	38	/
lvtlx	31	vD			A	B	581	
lvtlxl	31	vD			A	B	837	
lvtrx	31	vD			A	B	549	
lvtrxl	31	vD			A	B	805	
lvx	31	vD			A	B	103	/
lvxl	31	vD			A	B	359	/
mvidsplt	31	vD			A	B	110	
mviwsplt	31	vD			A	B	46	
stvebx	31	vS			A	B	135	/
stvehx	31	vS			A	B	167	/
stvewx	31	vS			A	B	199	/
stvx	31	vS			A	B	231	/
stvxl	31	vS			A	B	487	/
stvepx	31	vD			A	B	807	
stvepxl	31	vD			A	B	775	
stvexbx	31	vS			A	B	389	
stvexhx	31	vS			A	B	421	

Table A-7. X-Form (continued)

stvexwx	31	vS	A	B	453	
stvflx	31	vS	A	B	709	
stvflxl	31	vS	A	B	965	
stvfrx	31	vS	A	B	677	
stvfrxl	31	vS	A	B	933	
stvswx	31	vS	A	B	741	
stvswxl	31	vS	A	B	997	

Table A-8. VXR-Form

Name	OPCD (0–5)	vD (6–10)	vA (11–15)	vB (16–20)	Rc (21)	XO (22–31)
	Specific Instructions					
vcmpbfpx	04	vD	vA	vB	Rc	966
vcmpeqfpx	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

Appendix B
Revision History

This is the initial revision of the document.

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.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Asynchronous exception. Exceptions that are caused by events external to the processor's execution. In this document, the term 'asynchronous exception' is used interchangeably with the word *interrupt*.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC® architecture implements atomic accesses through the **lwarx/stwcx.** instruction pair.

B

BAT (block address translation) mechanism. A software-controlled array that stores the available block address translations on-chip.

Biased exponent. An exponent whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *most significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the *most significant byte*. See *Little-endian*.

Block. An area of memory that ranges from 128 Kbytes to 256 Mbytes whose size, translation, and protection attributes are controlled by the BAT mechanism.

Boundedly undefined. A characteristic of certain operation results 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.

Branch folding. The replacement with target instructions of a branch instruction and any instructions along the not-taken path when a branch is either taken or predicted as taken.

Branch prediction. The process of guessing whether a branch will be taken. Such predictions can be correct or incorrect; the term 'predicted' as it is used here does not imply that the prediction is correct (successful). The PowerPC architecture defines a means for static branch prediction as part of the instruction encoding.

Branch resolution. The determination of whether a branch is taken or not taken. A branch is said to be resolved when the processor can determine which instruction path to take. If the branch is resolved as predicted, the instructions following the predicted branch that may have been speculatively executed can complete. If the branch is not resolved as predicted, instructions on the mispredicted path, and any results of speculative execution, are purged from the pipeline and fetching continues from the nonpredicted path.

Burst. A multiple-beat data transfer whose total size is typically equal to a cache block.

Bus clock. Clock that causes the bus state transitions.

Bus master. The owner of the address or data bus; the device that initiates or requests the transaction.

C

Cache. High-speed memory 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 *cache*. The size of a cache block may vary among processors; the maximum block size is one *page*. In PowerPC processors, *cache coherency* is maintained on a cache-block basis. Note that the term 'cache block' is often used interchangeably with 'cache line.'

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the *cache block* is bypassed and the load or store is performed to or from main memory.

Cast out. A *cache block* that must be written to memory when a cache miss causes a cache block to be replaced.

Change bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the change bit if any store is performed into the *page*. See also *Referenced bit*.

Clean. An operation that causes a cache block to be written to memory, if modified, and then left in a valid, unmodified state in the cache.

Clear. To cause a bit or bit field to register a value of zero. See also *Set*.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *exception*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an exception).

Copy-back operation. A cache operation in which a cache line is copied back to memory to enforce cache coherency. Copy-back operations consist of snoop push-out operations and cache cast-out operations.

D

Deadline Scheduling. Deadline scheduling determines the execution order of tasks based on their deadlines.

Denormalized number. A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

Direct-mapped cache. A cache in which each main memory address can appear in only one location within the cache, operates more quickly when the memory request is a cache hit.

Direct-store segment access. An access to an I/O address space. The MPC603 defines separate memory-mapped and I/O address spaces, or segments, distinguished by the corresponding segment register T bit in the address translation logic of the MPC603. If the T bit is cleared, the memory reference is a normal memory-mapped access and can use the virtual memory management hardware of the MPC603. If the T bit is set, the memory reference is a direct-store access.

Double-word swap. AltiVec processors implement a double-word swap when moving quad words between vector registers and memory. The double word swap performs an additional swap to keep vector registers and memory consistent in little-endian mode. Double-word swap is referred to as ‘swizzling’ in the AltiVec technology architecture specification. This feature is not supported by the PowerPC architecture.

E

Effective address (EA). The 32-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address.

Exception. A condition encountered by the processor that requires special, supervisor-level processing.

Exception handler. A software routine that executes when an exception is taken. Normally, the exception handler corrects the condition that caused the exception, or performs some other meaningful task (that may include aborting the program that caused the exception). The address for each exception handler is identified by an exception vector offset defined by the architecture and a prefix selected via the MSR.

Extended opcode. A secondary opcode field generally located in instruction bits 21–30, that further defines the instruction type. All PowerPC instructions are one word in length. The most significant 6 bits of the instruction are the *Primary opcode*, identifying the type of instruction. See also *Primary opcode*.

Exclusive state. MEI state (E) in which only one caching device contains data that is also in system memory.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. See also *Biased exponent*.

F

Fetch. Retrieving instructions from either the cache or main memory and placing them into the instruction queue.

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs and store instructions move data from FPRs to memory. The FPRs are 64 bits wide and store floating-point values in double-precision format

Floating-point unit. The functional unit in the MPC603e processor responsible for executing all floating-point instructions.

Flush. An operation that causes a cache block to be invalidated and the data, if modified, to be written to memory.

Fraction. In the binary representation of a floating-point number, the field of the *significand* that lies to the right of its implied binary point.

Fully associative. Addressing scheme where every cache location (every byte) can have any possible address.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H

Harvard architecture. An architectural model featuring separate caches and other memory management resources for instructions and data.

Hashing. An algorithm used in the *page table* search process.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point numbers.

Illegal instructions. A class of instructions that are not implemented for a particular PowerPC processor. These include instructions not defined by the PowerPC architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the PowerPC architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The PowerPC architecture has many different implementations.

Implementation-dependent. An aspect of a feature in a processor's design that is defined by a processor's design specifications rather than by the PowerPC architecture.

Implementation-specific. An aspect of a feature in a processor's design that is not required by the PowerPC architecture, but for which the PowerPC architecture may provide concessions to ensure that processors that implement the feature do so consistently.

Imprecise exception. A type of *synchronous exception* that is allowed not to adhere to the precise exception model (see *Precise exception*). The PowerPC architecture allows only floating-point exceptions to be handled imprecisely.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

Instruction queue. A holding place for instructions fetched from the current instruction stream.

Integer unit. The functional unit in the MPC603e responsible for executing all integer instructions.

In-order. An aspect of an operation that adheres to a sequential model. An operation is said to be performed in-order if, at the time that it is performed, it is known to be required by the sequential execution model. See *Out-of-order*.

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Instruction parallelism. A feature of PowerPC processors that allows instructions to be processed in parallel.

Interrupt. An external signal that causes the MPC603e to suspend current execution and take a predefined exception.

L

Latency. The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.

L2 cache. See *Secondary cache*.

Least-significant bit (lsb). The bit of least value in an address, register, field, data element, or instruction encoding.

Least-significant byte (LSB). The byte of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. See *Big endian*.

Loop unrolling. Loop unrolling provides a way of increasing performance by allowing more instructions to be issued in a clock cycle. The compiler replicates the loop body to increase the number of instructions executed between a loop branch.

M

Mantissa. The decimal part of logarithm.

MESI (modified/exclusive/shared/invalid). *Cache coherency* protocol used to manage caches on different devices that share a memory system. Note that the PowerPC architecture does not specify the implementation of a MESI protocol to ensure cache coherency.

Memory access ordering. The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory-mapped accesses. Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU). The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Microarchitecture. The hardware details of a microprocessor's design. Such details are not defined by the PowerPC architecture.

Mnemonic. The abbreviated name of an instruction used for coding.

Modified state. MEI state (M) in which one, and only one, caching device has the valid data for that address. The data at this address in external memory is not valid.

Modular Arithmetic. Arithmetic in which integers that are congruent modulo any given integer are considered equal.

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.

Munging. A modification performed on an *effective address* that allows it to appear to the processor that individual aligned scalars are stored as *little-endian* values, when in fact it is stored in *big-endian* order, but at different byte addresses within double words. Note that munging affects only the effective address and not the byte order. Note also that this term is not used by the PowerPC architecture.

Multiprocessing. The capability of software, especially operating systems, to support execution on more than one processor at the same time.

N

NaN. An abbreviation for not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs—*signaling NaNs* and *quiet NaNs*.

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 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. Since the 32-bit registers of the MPC603e cannot represent this sum, an overflow condition occurs.

P

Page. A region in memory. The OEA defines a page as a 4-Kbyte area of memory, aligned on a 4-Kbyte boundary.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory*. On PowerPC processors, a page fault exception condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Page table. A table in memory is comprised of *page table entries*, or PTEs. It is further organized into eight PTEs per PTEG (page table entry group). The number of PTEGs in the page table depends on the size of the page table (as specified in the SDR1 register).

Page table entry (PTE). Data structures containing information used to translate *effective address* to physical address on a 4-Kbyte page basis. A PTE consists of 8 bytes of information in a 32-bit processor and 16 bytes of information in a 64-bit processor.

Park. The act of allowing a bus master to maintain bus mastership without having to arbitrate.

Persistent data stream. A data stream is considered to be persistent when it is expected to be loaded from frequently.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise exceptions. A category of exception for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete and subsequent instructions can be flushed and redispatched after exception handling has completed. See *Imprecise exceptions*.

Primary opcode. The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction.

Program order. The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache

Protection boundary. A boundary between *protection domains*.

Protection domain. A protection domain is a segment, a virtual page, a BAT area, or a range of unmapped effective addresses. It is defined only when the appropriate relocate bit in the MSR (IR or DR) is 1.

Q

Quad word. A group of 16 contiguous locations starting at an address divisible by 16.

Quiet NaN. A type of *NaN* that can propagate through most arithmetic operations without signaling exceptions. A quiet NaN is used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. See *Signaling NaN*.

R

rA. The **r**A instruction field is used to specify a GPR to be used as a source or destination.

rB. The **r**B instruction field is used to specify a GPR to be used as a source.

rD. The **r**D instruction field is used to specify a GPR to be used as a destination.

rS. The **r**S instruction field is used to specify a GPR to be used as a source.

Real address mode. An MMU mode when no address translation is performed and the *effective address* specified is the same as the physical address. The processor's MMU is operating in real address mode if its ability to perform address translation has been disabled through the MSR registers IR and/or DR bits.

Record bit. Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Reference bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the *reference bit* whenever the page is accessed for a read or write.

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.

Rename register. Temporary buffers used by instructions that have finished execution but have not completed.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes a **lwarx** instruction to read a memory semaphore into a GPR.

Reservation station. A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.

RISC (reduced instruction set computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Saturate. A value v which lies outside the range of numbers representable by a destination type is replaced by the representable number closest to v.

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.

Set (*v*)**.** To write a nonzero value to a bit or bit field; the opposite of *clear*. The term 'set' may also be used to generally describe the updating of a bit or bit field.

Set (*n*). A subdivision of a *cache*. Cacheable data can be stored in a given location in one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. See *Set associative*.

Set-associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Signaling NaN. A type of *NaN* that generates an invalid operation program exception when it is specified as arithmetic operands. See *Quiet NaN*.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

SIMD. Single instruction stream, multiple data streams. A vector instruction can operate on several data elements within a single instruction in a single functional unit. SIMD is a way to work with all the data at once (in parallel), which can make execution faster.

Simplified mnemonics. Assembler mnemonics that represent a more complex form of a common operation.

Snooping. Monitoring addresses driven by a bus master to detect the need for coherency actions.

Snoop push. Response to a snooped transaction that hits a modified cache block. The cache block is written to memory and made available to the snooping device.

Splat. A splat instruction will take one element and replicates (splats) that value into a vector register. The purpose being to have all elements have the same value so they can be used as a constant to multiply other vector registers.

Split-transaction. A transaction with independent request and response tenures.

Split-transaction bus. A bus that allows address and data transactions from different processors to occur independently.

Stage. The term 'stage' is used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. In the latter case, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. Typically, the latency of a stage is one processor clock cycle. Some events, such as dispatch, write-back, and completion, happen instantaneously and may be thought to occur at the end of a stage. An instruction can spend multiple cycles in one stage. An integer multiply, for example, takes multiple cycles in the execute stage. When this occurs, subsequent instructions may stall. An instruction may also occupy more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource—for example, when instructions are dispatched they are assigned a place in the CQ at the same time they are passed to the execute stage. They can be said to occupy both the complete and execute stages in the same clock cycle.

Stall. An occurrence when an instruction cannot proceed to the next stage.

Static branch prediction. Mechanism by which software (for example, compilers) can hint to the machine hardware about the direction a branch is likely to take.

Sticky bit. A bit that when *set* must be cleared explicitly.

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.

Swizzling. See *Double-word swap*.

Synchronization. A process to ensure that operations occur strictly *in order*. See *Context synchronization*.

Synchronous exception. An *exception* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous exceptions, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

Tenure. The period of bus mastership. For the MPC603e, there can be separate address bus tenures and data bus tenures. A tenure consists of three phases: arbitration, transfer, and termination.

TLB (translation lookaside buffer). A cache that holds recently-used *page table entries*.

Throughput. The measure of the number of instructions that are processed per clock cycle.

Tiny. A floating-point value that is too small to be represented for a particular precision format, including *denormalized* numbers; they do not include ±0.

Transaction. A complete exchange between two bus devices. A transaction is typically comprised of an address tenure and one or more data tenures, which may overlap or occur separately from the address tenure. A transaction may be minimally comprised of an address tenure only.

Transient stream. A data stream is considered to be transient when it is likely to be referenced from infrequently.

U

UISA (user instruction set architecture). The level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and exception model as seen by user programs, and the memory and programming models.

Underflow. A condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or *mantissa* than the single-precision format can provide. In other words, the result is too small to be represented accurately.

User mode. The operating state of a processor used typically by application software. In user mode, software can access only certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

vA. The **v**A instruction field is used to specify a vector register to be used as a source or destination.

vB. The **v**B instruction field is used to specify a vector register to be used as a source.

vC. The **v**C instruction field is used to specify a vector register to be used as a source.

vD. The **v**D instruction field is used to specify a vector register to be used as a destination.

vS. The **v**S instruction field is used to specify a vector register to be used as a source.

VEA (virtual environment architecture). The level of the *architecture* that describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time-base facility from a user-level perspective. *Implementations* that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

Vector. The spatial parallel processing of short, fixed-length one-dimensional matrices performed by an execution unit.

Vector Register (VR). Any of the 32 registers in the vector register file. Each vector register is 128 bits wide. These registers can provide the source operands and destination results for AltiVec instructions.

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to virtual memory is possible only when it coincides with *physical memory*.

W

Way. A location in the cache that holds a cache block, its tags and status bits.

Weak ordering. A memory access model that allows bus operations to be reordered dynamically, which improves overall performance and in particular reduces the effect of memory latency on instruction throughput.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.