Vector Extensions to the MIPS-IV Instruction Set Architecture
(The V-IRAM Architecture Manual)
Revision 3.7.5

David Martin

March 4, 2000

Contents

1 Vector Architectural State **11**

1.1 Overview . . . 11

1.2 Vector Control Registers . . . 11

1.3 Vector Scalar Registers . . . 11

1.4 Vector Flag Registers . . . 12

1.5 Number of Virtual Processors . . . 12

1.6 Virtual Processor Width . . . 12

2 Instruction Set **19**

2.1 Open Questions . . . 19

2.2 Data Types . . . 19

2.3 Vector Operations . . . 19

2.4 Flag Register Use . . . 20

2.5 Memory Consistency . . . 20

2.6 Exception Model . . . 20

2.7 Speculative Execution . . . 20

2.8 Assembly Language . . . 20

2.9 Instructions . . . 21

2.9.1 Vector Integer Arithmetic Instructions . . . 21

2.9.2 Vector Logical Instructions . . . 21

2.9.3 Vector Floating-Point Arithmetic Instructions . . . 23

2.9.4 Vector Floating-Point Convert Instructions . . . 23

2.9.5 Vector Fixed-Point Arithmetic Instructions 23
2.9.6 Vector Logical Flag Instructions 24
2.9.7 Flag Processing Instructions 24
2.9.8 Vector Processing Instructions 25
2.9.9 Vector Memory Instructions 25
2.9.10 Coprocessor Interface Instructions 26
2.9.11 Miscellaneous Vector Instructions 26
2.9.12 Miscellaneous Kernel Vector Instructions 26

A Instruction Formats **45**

B Instruction Definitions **47**

CFC2 — Control From Coprocessor 2 48
CTC2 — Control To Coprocessor 2 49
MTC2 — Move To Coprocessor 2 50
VABS — Vector Integer Absolute Value 51
VABS.fmt — Vector Floating-Point Absolute Value 53
VACOMMIT — Commit Speculative Arithmetic 56
VADD — Signed Vector Integer Add 57
VADD.U — Unsigned Vector Integer Add 59
VADD.fmt — Vector Floating-Point Add 60
VAND — Vector And 64
VCEIL — Vector Floating-Point Ceiling 65
VCIOTA — Vector Continuous Iota 70
VCMP — Signed Vector Integer Compare 71
VCMP.U — Unsigned Vector Integer Compare 73
VCMP.fmt — Vector Floating-Point Compare 75
VCOMPRESS — Vector Compress 79
VCVT — Vector Convert 80
VDIV — Signed Vector Integer Divide 91

VDIVU — Unsigned Vector Integer Divide . . . 93
VDIV.fmt — Vector Floating-Point Divide . . . 95
VEXPAND — Vector Expand . . . 99
VEXT.SV — Signed Scalar-Vector Extract . . . 100
VEXT.U.SV — Unsigned Scalar-Vector Extract . . . 101
VEXT.VV — Vector-Vector Extract . . . 102
VEXTHALF — Vector Extract Half . . . 103
VHALF — Vector Half . . . 104
VHALFDN — Vector Half Down . . . 106
VHALFUP — Vector Half Up . . . 108
VFAND — Vector Flag And . . . 110
VFCLR — Vector Flag Clear . . . 111
VFCLR8 — Vector Flag Clear 8 . . . 112
VFFF1 — Vector Flag Find First One . . . 113
VFFL1 — Vector Flag Find Last One . . . 114
VFINS — Scalar-Vector Insert . . . 115
VFLD — Vector Flag Load . . . 116
VFLOOR — Vector Floating-Point Floor . . . 119
VFMF8 — Vector Flag Move From 8 . . . 125
VFMT8 — Vector Flag Move To 8 . . . 126
VFNOR — Vector Flag Nor . . . 127
VFOR — Vector Flag Or . . . 128
VFOR8 — Vector Flag Or 8 . . . 129
VFPOP — Vector Flag Population Count . . . 130
VFLUSH — Flush Vector Unit . . . 131
VFSET — Vector Flag Set . . . 132
VFSETBF — Vector Flag Set Before First One . . . 133
VFSETIF — Vector Flag Set Including First One . . . 134
VFSETOF — Vector Flag Set Only First One . . . 135

VFST — Vector Flag Store . . . 136
VFXOR — Vector Flag Xor . . . 138
VINS.SV — Scalar-Vector Insert . . . 139
VINS.VV — Vector-Vector Insert . . . 140
VIOTA — Vector Iota . . . 142
VLD — Unit Stride Signed Vector Load . . . 143
VLD.U — Unit Stride Unsigned Vector Load . . . 146
VLDS — Variable Stride Signed Vector Load . . . 148
VLDS.U — Variable Stride Unsigned Vector Load . . . 151
VLDX — Indexed Signed Vector Load . . . 153
VLDX.U — Indexed Unsigned Vector Load . . . 155
VMADD.fmt — Vector Floating-Point Multiply Add . . . 157
VMAX — Signed Vector Integer Maximum . . . 161
VMAX.U — Unsigned Vector Integer Maximum . . . 162
VMCOMMIT — Commit Speculative Arithmetic . . . 163
VMCTS — Vector Move Control To Scalar . . . 164
VMERGE — Vector Merge . . . 165
VMIN — Signed Vector Integer Minimum . . . 167
VMIN.U — Unsigned Vector Integer Minimum . . . 168
VMOD — Signed Vector Integer Modulus . . . 169
VMODU — Unsigned Vector Integer Modulus . . . 171
VMSTC — Vector Move Scalar To Control . . . 173
VMSUB.fmt — Vector Floating-Point Multiply Subtract . . . 174
VMUL.fmt — Vector Floating-Point Multiply . . . 178
VMULHI — Vector Integer Multiply High . . . 182
VMULHI.U — Vector Integer Multiply High . . . 183
VMULLO — Vector Integer Multiply Low . . . 184
VNEG.fmt — Vector Floating-Point Negate . . . 185
VNMADD.fmt — Vector Floating-Point Negative Multiply Add . . . 189

VNMSUB.fmt — Vector Floating-Point Negative Multiply Subtract 193
VNOR — Vector Nor 197
VOR — Vector Or 198
VRECIP.fmt — Vector Floating-Point Reciprocal 199
VROUND — Vector Floating-Point Round 203
VRSQRT.fmt — Vector Floating-Point Reciprocal Square Root 209
VRSYNC — Vector Register Sync 213
VSADD — Signed Saturating Add 214
VSADD.U — Unsigned Saturating Add 216
VSAT — Signed Vector Saturate 218
VSAT.SU — Signed to Unsigned Vector Saturate 220
VSAT.U — Unsigned Vector Saturate 222
VSATVL — Saturate Vector Length 224
VSLL — Vector Shift Left Logical 225
VSLS — Signed Saturating Left Shift 227
VSLS.U — Unsigned Saturating Left Shift 229
VSQRT.fmt — Vector Floating-Point Square Root 231
VSRA — Vector Arithmetic Right Shift 235
VSRL — Vector Shift Right Logical 237
VSRR — Signed Shift Right And Round 239
VSRR.U — Unsigned Shift Right And Round 240
VSSUB — Signed Saturating Subtract 241
VSSUB.U — Signed Saturating Subtract 243
VST — Unit Stride Vector Store 245
VSTS — Variable Stride Vector Store 247
VSTX — Unordered Indexed Vector Store 249
VSTXO — Ordered Indexed Vector Store 251
VSUB — Signed Vector Integer Subtract 253
VSUB.U — Unsigned Vector Integer Subtract 255

VSUB.fmt — Vector Floating-Point Subtract 257
VSYNC — Vector Sync 261
VTLBP — Vector TLB Probe 262
VTLBR — Vector TLB Read 263
VTLBWI — Vector TLB Write Indexed 264
VTLBWR — Vector TLB Write Random 265
VTRUNC — Vector Floating-Point Truncate 266
VXLMADD — Signed Multiply Add Lower Halves 272
VXLMADD.U — Unsigned Multiply Add Lower Halves 275
VXLMSUB — Signed Multiply Subtract Lower Halves 278
VXLMSUB.U — Unsigned Multiply Subtract Lower Halves 281
VXLMUL — Signed Multiply Lower Halves 284
VXLMUL.U — Unsigned Multiply Lower Halves 286
VXOR — Vector Xor 288
VXUMADD — Signed Multiply Add Upper Halves 289
VXUMADD.U — Unsigned Multiply Add Upper Halves 292
VXUMSUB — Signed Multiply Subtract Upper Halves 295
VXUMSUB.U — Unsigned Multiply Subtract Upper Halves 298
VXUMUL — Signed Multiply Upper Halves 301
VXUMUL.U — Unsigned Multiply Upper Halves 303

List of Figures

1.1 Vector Architectural State 14

1.2 Data and Flag Register Element Mappings 15

1.3 Vector Control Registers 16

1.4 Vector Flag Register Contents 17

2.1 IEEE Floating-Point Comparisions 22

2.2 The Vector Convert Instruction 23

2.3 Fixed-Point Rounding Modes 24

2.4 Vector Instruction Set Summary 27

2.5 Instruction Qualifiers 28

2.6 Signed Vector Integer Arithmetic Instructions 29

2.7 Unsigned Vector Integer Arithmetic Instructions 30

2.8 Vector Logical Instructions 31

2.9 Vector Floating-Point Instructions 32

2.10 Vector Floating-Point Convert Instructions 33

2.11 Signed Vector Fixed-Point Instructions 34

2.12 Unsigned Fixed-Point Instructions 35

2.13 Vector Flag Logical Instructions 36

2.14 Vector Flag Processing Instructions 37

2.15 Vector Processing Instructions I 38

2.16 Vector Processing Instructions II 39

2.17 Vector Load Instructions 40

2.18 Vector Store Instructions 41

2.19 Coprocessor Interface Instructions 42
2.20 Miscellaneous Instructions 43
2.21 Miscellaneous Kernel-Mode Instructions 44

Chapter 1

Vector Architectural State

1.1 Overview

The vector architectural state is shown in Figure 1.1. A vector processor is a SIMD array of virtual processors (VP), where each VP has 32 data registers and 32 flag registers. The number of virtual processors is equal to the vector length (VL), which is never greater than the maximum vector length (MVL). MVL must be at least 16. A vector instruction performs the same operation on each VP in lockstep.

The VP width is set by the vector control register $\texttt{vc}_{\texttt{vpw}}$. MVL may be different for different VP widths, but the number of registers does not change. A register's contents are defined if and only if they are read and written with the same VP width.

Apart from the vector flag and data register files, there are two additional register files. The vector control register file is used in a variety of ways to control the operation of the vector unit (see Section 1.2). The vector scalar registers are used as source and destination registers for vector computations that require scalar sources and/or destinations.[1]

1.2 Vector Control Registers

Figure 1.3 shows the contents of the 64 64-bit vector control registers. See the IRAM microarchitecture specification for a detailed description of these registers. Vector control register `vinc0` is fixed at zero.

1.3 Vector Scalar Registers

There are 32 64-bit vector scalar registers. All are general purpose registers. Vector scalar register 0 is fixed at zero.

[1] A vector architecture that was not constrained by a coprocessor interface would probably use the MIPS general purpose integer registers and floating-point registers for this purpose.

1.4 Vector Flag Registers

Figure 1.4 shows the conventional usage of the flag registers. Either $\texttt{vf}_{\texttt{mask0}}$ or $\texttt{vf}_{\texttt{mask1}}$ is used as a mask for nearly all vector operations.

NOTE: *More detail coming soon. May flags are used for exception processing. Many may be used as general purpose registers.*

1.5 Number of Virtual Processors

The number of active virtual processors is given by the contents of the vector length register $\texttt{vc}_{\texttt{vl}}$. The number of active VPs may vary anywhere between 0 and the implementation-defined maximum vector length, which is stored in the control register $\texttt{vc}_{\texttt{mvl}}$. The architecture defines a minimum maximum vector length. This value is chosen to mediate two opposing pressures:

- MVL should be small to enable minimal implementations.
- MVL should be large so that more loops with a known number of iterations do not need to be strip-mined.

The minimum MVL is 16.

1.6 Virtual Processor Width

The width of each VP is given by the virtual processor width register $\texttt{vc}_{\texttt{vpw}}$. Valid VP widths are 8, 16, 32, and 64 bits.[2] In order make maximal use of available hardware resources, an implementation should increase MVL when VPW decreases. Figure 1.2 shows how MVL is doubled when VPW is halved in order to use all bits in the data register file for all VP widths. Moving to smaller VP widths causes the data elements to be fragmented into smaller pieces. Moving to larger VP widths causes the data elements to be catenated. Note that moving to smaller VP widths is always well defined. Moving to larger VP widths is defined for data elements only when the number of small elements is an exact multiple of the number of small elements per large element.

Note that the mapping of small VPs within largeer VPs exposes the endian-ness of the implementation. When an implementation is big-endian, the mapping of small VPs within large VPs should also be big-endian. When an implementation is little-endian, the VP mapping should also be little-endian. For example, if 8-bit data is loaded into 64-bit VPs with 64-bit loads on a big-endian machine, the first 8-bit VP will contain the most-significant byte of the first 64-bit VP.

The mapping of flag register elements between different VP widths is more complex than for the data elements. For all VP widths, there are 32 1-bit flag registers per VP. Since MVL is larger for a smaller VPW, the number of bits of flag state is larger for a smaller VPW. This means that from an architectural point of view, the flag registers are different sizes for different VP widths. Because of this variability, the simple inclusion property that holds for the data registers does not work. Nevertheless, it is useful to define the state of the flag registers across VP width changes.

Moving to smaller VP widths causes flag elements to be replicated, and is always well defined. Moving to larger VP widths causes flag elements to be combined, under two restrictions. First, only identical flag values may be combined.

[2] The first implementation of this architecture – VIRAM-1 – will not implement 8-bit VPs.

An attempt to combine unlike flag values produces an undefined flag value. Second, the number of flags in the small VP width must be an exact multiple of the number of flags that combine to produce one flag in the large VP width.

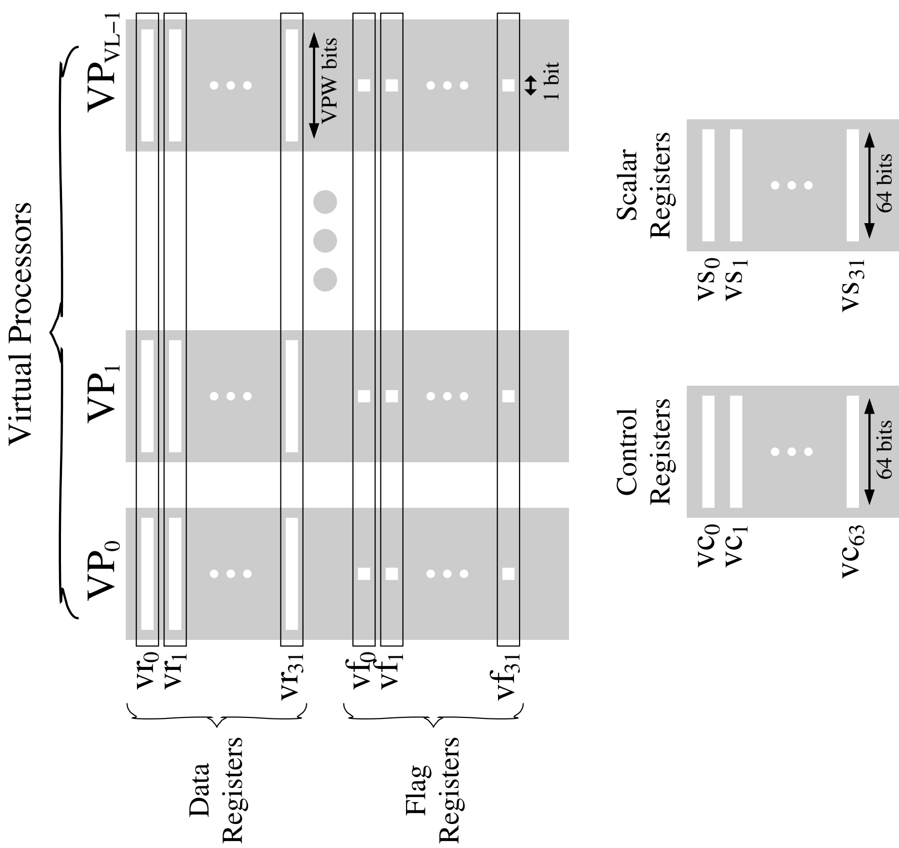

Figure 1.1: Vector Architectural State

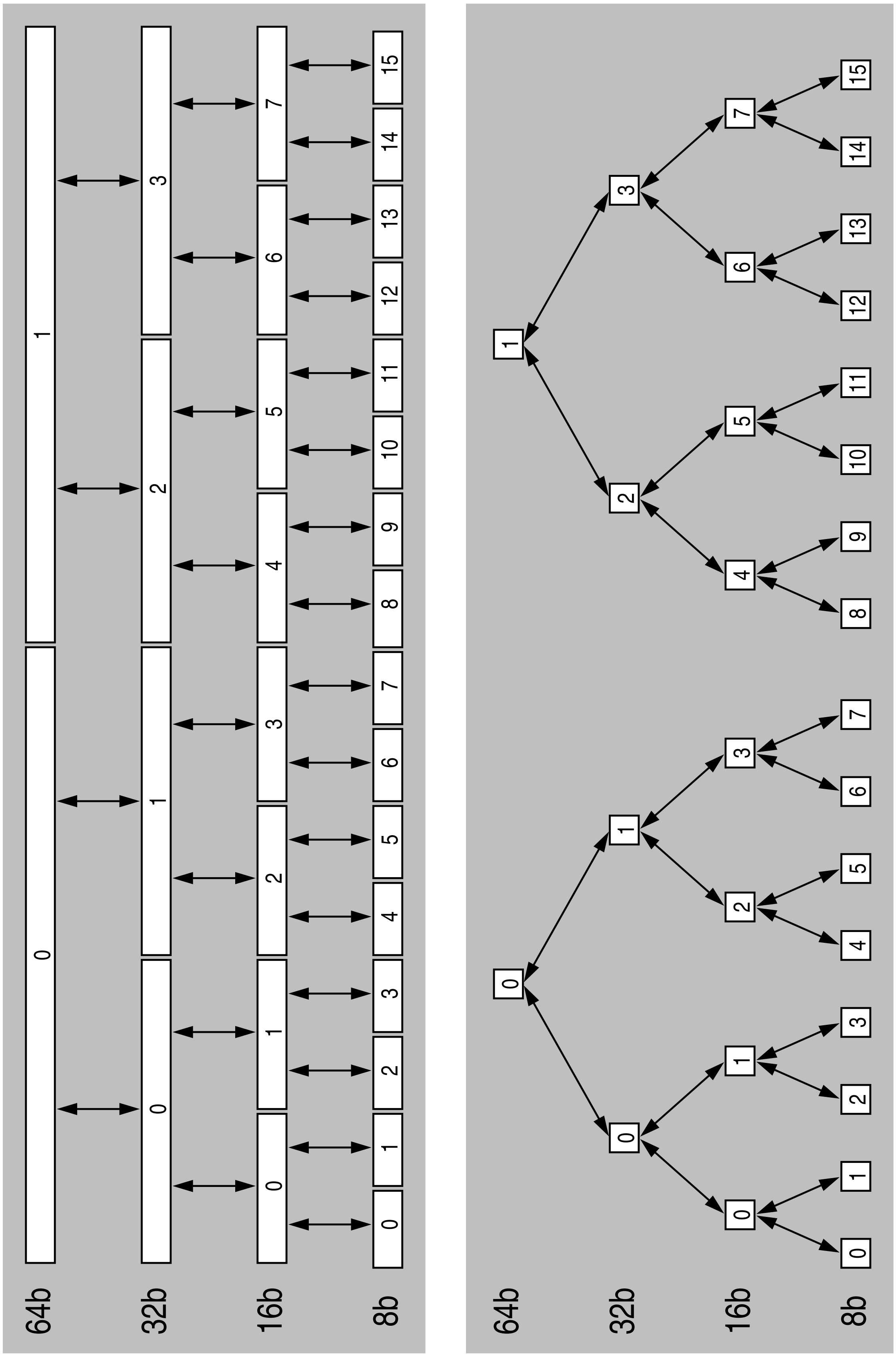

Figure 1.2: Data and Flag Register Element Mappings The upper portion of the figure shows the mapping of *data* elements when VP width changes. The lower portion of the figure shows the mapping of *flag* elements when VP width changes.

Hardware Name	Software Name	User Access	Kernel Access	Contents
`$vc0`	`vl`	rw	rw	Vector length
`$vc1`	`vpw`	rw	rw	Virtual processor width
`$vc2`	`vshamt`	rw	rw	Shift amount
`$vc3`	`vindex`	rw	rw	Element index
`$vc4`	`vcat`	rw	rw	Assembler temporary
`$vc5`	-	-	-	*Reserved*
`$vc6`	`vmode`	rw	rw	Execution Mode
`$vc7`	`vregstatus`	rw	rw	Vector register valid and dirty bits
`$vc8`	`vpsw`	-	rw	Vector processor status word
`$vc9`	`vconfig`	-	rw	I/O configuration
`$vc10`	`vcause`	-	rw	Vector exception cause
`$vc11`	`vbadvaddr`	-	rw	Bad virtual address
`$vc12`	`veiword`	-	r	Instruction word for exceptions
`$vc13`	`vaiword`	-	r	Instruction word for arithmetic exceptions
`$vc14`	`vwatch`	-	rw	Pysical watch address
`$vc15`	`vmemerror`	-	rw	Memory error and status
`$vc16`	`vtlbindex`	-	rw	Vector TLB index
`$vc17`	`vtlbrandom`	-	r	Vector TLB random index
`$vc18`	`vtlbentryhi`	-	rw	Vector TLB entry hi
`$vc19`	`vtlbentrylo`	-	rw	Vector TLB entry lo
`$vc20`	`vpagemask`	-	rw	Vector TLB entry mask
`$vc21`	`vtlbwired`	-	rw	Vector TLB wired entry index
`$vc22`	`vsafevl`	r	rw	Safe vector length
`$vc23`	`vrev`	r	r	Revision
`$vc24`	`mvl`	r	r	Maximum vector length
`$vc25`	`logmvl`	r	r	Base-2 logarithm of `mvl`
`$vc26`	`vcycle`	r	rw	Cycle counter
`$vc27`	-	-	-	*Reserved*
`$vc28`	-	-	-	*Reserved*
`$vc29`	-	-	-	*Reserved*
`$vc30`	-	-	-	*Reserved*
`$vc31`	-	-	-	*Reserved*
`$vc32`	`vbase0`	rw	rw	Base Register 0
...	...	...	...	...
`$vc47`	`vbase15`	rw	rw	Base Register 15
`$vc48`	`vinc0`	r	r	Auto-Increment Register 0
`$vc49`	`vinc1`	rw	rw	Auto-Increment Register 1
...	...	...	...	...
`$vc55`	`vinc7`	rw	rw	Auto-Increment Register 7
`$vc56`	`vstride0`	rw	rw	Stride Register 0
...	...	...	...	...
`$vc63`	`vstride7`	rw	rw	Stride Register 7

Figure 1.3: Vector Control Registers

Hardware Name	*Software Name*	*Contents*
`$vf0`	`vfmask0`	Primary mask
`$vf1`	`vfmask1`	Alternate mask
`$vf2`	`vfgr0`	General purpose
`$vf3`	`vfgr1`	General purpose
`$vf4`	`vfgr2`	General purpose
`$vf5`	`vfe0.L`	Speculative load fault
`$vf6`	`vfe1.L`	Speculative load fault
`$vf7`	`vfe2.L`	Speculative load fault
`$vf8`	`vfe0.F`	Integer overflow
`$vf9`	`vfe0.S`	Integer saturate
`$vf10`	`vfe0.I`	Floating-point inexact
`$vf11`	`vfe0.U`	Floating-point underflow
`$vf12`	`vfe0.O`	Floating-point overflow
`$vf13`	`vfe0.Z`	Floating-point divide-by-zero
`$vf14`	`vfe0.V`	Floating-point invalid
`$vf15`	`vfe0.E`	Floating-point unimplemented
`$vf16`	`vfe1.F`	Integer overflow
`$vf17`	`vfe1.S`	Integer saturate
`$vf18`	`vfe1.I`	Floating-point inexact
`$vf19`	`vfe1.U`	Floating-point underflow
`$vf20`	`vfe1.O`	Floating-point overflow
`$vf21`	`vfe1.Z`	Floating-point divide-by-zero
`$vf22`	`vfe1.V`	Floating-point invalid
`$vf23`	`vfe1.E`	Floating-point unimplemented
`$vf24`	`vfe2.F`	Integer overflow
`$vf25`	`vfe2.S`	Integer saturate
`$vf26`	`vfe2.I`	Floating-point inexact
`$vf27`	`vfe2.U`	Floating-point underflow
`$vf28`	`vfe2.O`	Floating-point overflow
`$vf29`	`vfe2.Z`	Floating-point divide-by-zero
`$vf30`	`vfe2.V`	Floating-point invalid
`$vf31`	`vfe2.E`	Floating-point unimplemented

Figure 1.4: Vector Flag Register Contents

Chapter 2

Instruction Set

2.1 Open Questions

This section contains a list of random open questions about the instruction set architecture.

1. There will probably be some instructions relating to fast user-level message passing.
2. It is not clear how relaxed the vector memory consistency model should be. In particular, should a virtual processor see the order of its own memory references? This is called intra-VP consistency, or simply VP consistency in this document. What implementaation could take advantage of relaxing this consistency guarantee?
3. Linkage conventions. MIPS registers have conventional uses for parameter passing, globals, etc. Such conventions will defined soon for the vector, flag, and vector-scalar registers.

2.2 Data Types

The integer data width for an integer vector operation is always equal to the VP width. Integers loaded from memory are either sign- or zero-extended to the VP width.

Both 32-bit and 64-bit IEEE-754 standard floating-point formats are supported when the VP width is 64 bits. The 32-bit floating-point format is supported when the VP width is 32 bits. A 32-bit floating-point value is always stored in the low-order half of a 64-bit register. In this case, the upper 32 bits aren't touched by floating-point operations.

2.3 Vector Operations

Nearly all vector operations follow the same operational idiom. They are goverened by a vector length and a vector mask. The vector length is taken from the vector length register. The vector mask is chosen from one of the two vector flag registers that are reserved for masking. Each virtual processor whose number is less than the vector length and whose mask bit is set performs the operation. If the vector length is zero, or if all VPs are masked out, then the instruction is a nop.

A VP takes its source operands from either a vector register or a scalar register. The destination is either a vector register or a scalar register, but is usually a vector register. Instructions whose source operands are both vectors are called *vector-vector* operations (qualified with `.vv`). Instructions may also be *vector-scalar* (qualified with `.vs`) if the second source is a scalar, or *scalar-vector* (qualified with `.sv`), if the first source is a scalar. Non-commutative operations are provided in both scalar-vector and vector-scalar forms.

2.4 Flag Register Use

Each VP has 32 1-bit flag registers. The flag registers have two important uses: They are used to store mask vectors to provide predicated execution, and they are used to store exception bits.

A flag register reserved for exception processing is reserved by convention only. It may be used as a general purpose flag register in the absense of the relevant exception. This must be done with care, as the register is implicitly set by any instruction that may produce the relevant exception.

2.5 Memory Consistency

The memory consistency model may be best described as guaranteeing processor consistency for each virtual processor. Thus, each VP observes the order of its own memory operations, but is does not observe any guaranteed order of other VPs' memory operations. There are no ordering guarantees between scalar and vector instructions. Memory ordering must be explicitly enforced with vector sync instructions. These instructions control memory consistency between the scalar and vector unit and between virtual processors within the vector unit.

There is also a memory ordering guraanteeed between virtual processors of different widths. Operations between a narrow and a wide VP are ordered if the narrow VP is contained within the wide VP. For example, the memory operations of the 64-bit wide VP_0, the 32-bit wide $\mathrm{VP}_0 \cdots \mathrm{VP}_1$, and the 16-bit wide $\mathrm{VP}_1 \cdots \mathrm{VP}_3$ are all ordered.

2.6 Exception Model

Coming soon...

2.7 Speculative Execution

Coming soon...

2.8 Assembly Language

Registers should be specified, wherever possible, using software names. The software names for vector flag registers are listed in Figure 1.4, and the software names for vector control registers are listed in Figure 1.3. There are currently no software names for vector or scalar data registers. The hardware names for the vector data registers are `$vr0`..`$vr31`. The hardware names for the scalar data registers are `$vs0`..`$vs31`.

2.9 Instructions

The vector instruction set is described in this document in three levels of detail. First, Figure 2.4 summarizes the entire instruction set in a single table. This table is divided into blocks of instructions of similar types. Eaach block is expanded into a more detailed table on the page named in the block's title. This collection of tables is the second level of detail, showing assembly formats and giving a short description of the instruction's operation.

The final level of detail is contained in Appendix B, where each instruction is defined in detail on a seperate page. The first column of Figure 2.4 lists the pages in Appendix B on which the instructions are formally defined.

The assembly language format of an instruction is written with a shorthand notation, since the cross-product of legal qualifiers can produce many distinct opcodes. For example, the format of the vector floating-point add instruction in Figure 2.9 on Page 32 is shown as:

$$\text{vadd}\left\{\begin{matrix}\texttt{.s}\\\texttt{.d}\end{matrix}\right\}\left\{\begin{matrix}\texttt{.vv[.1]}\ \texttt{vr}_{\texttt{dest}},\ \texttt{vr}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}}\\\texttt{.sv[.1]}\ \texttt{vr}_{\texttt{dest}},\ \texttt{vs}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}}\end{matrix}\right.$$

The brackets [] denote an optional qualifier. The braces {} contain all choices for a required qualifier. Note that in this case, the assembly language register specifiers are contained within a set of braces so that they match the `.vv/.vs` qualifier. Thus, the expanded list of formats for this instruction is:

```
vadd.s.vv vrdest, vrsrc1, vrsrc2
vadd.s.vv.1 vrdest, vrsrc1, vrsrc2
vadd.s.sv vrdest, vssrc1, vrsrc2
vadd.s.sv.1 vrdest, vssrc1, vrsrc2
vadd.d.vv vrdest, vrsrc1, vrsrc2
vadd.d.vv.1 vrdest, vrsrc1, vrsrc2
vadd.d.sv vrdest, vssrc1, vrsrc2
vadd.d.sv.1 vrdest, vssrc1, vrsrc2
```

2.9.1 Vector Integer Arithmetic Instructions

The vector integer arithmetic instructions are summarized in Figure 2.6 andFigure 2.7. All integer operations operate on VP-width operands and produce VP-width results.

Both signed and unsigned versions of some operations are provided. In most cases, this is because the computation is different depending on signedness (e.g. divide). However, both signed and unsigned add and subtract are provided, even though the computations are idential. The difference here is that signed add and subtract can overflow, while unsigned add and subtract never overflow under any circumstances.

2.9.2 Vector Logical Instructions

Figure 2.8 summarizes the vector logical operations. The usual set of logical operations is provided.

	Predicate	Predicate true if...				Invalid if
		Unordered	Less Than	Equal	Greater Than	Unordered
	(F)					
$\star$	LT		√			√
$\star$	EQ			√		
$\star$	GT				√	√
$\star$	LE		√	√		√
$\star$	GE			√	√	√
	LG		√		√	√
	LEG		√	√	√	√
†	UN	√				
	ULT	√	√			
	UEQ	√		√		
	UGT	√			√	
	ULE	√	√	√		
	UGE	√		√	√	
	(ULG)	√	√		√	
	(ULEG)	√	√	√	√	
	(T)	√	√	√	√	
‡	NLT	√		√	√	√
$\star$	NEQ	√	√		√	
‡	NGT	√	√	√		√
‡	NLE	√			√	√
‡	NGE	√	√			√
	NLG	√		√		√
	NLEG	√				√
‡	NUN		√	√	√	
	NULT			√	√	
	NUEQ		√		√	
	NUGT		√	√		
	NULE				√	
	NUGE		√			
	(NULG)			√		
	(NULEG)					

Figure 2.1: IEEE Floating-Point Comparisions
The full cross-product of the independent conditions *unordered*, *less than*, *equal*, and *greater than* produces 32 predicates, but only 26 are distinct. The 6 predicates in parentheses are either degenerate (F, ULEG, T, NULEG evalute to either true or false) or redundant (ULG is identical to NEQ, and NULG is identical to EQ).
The 6 predicates marked with $\star$ are "obligatory in any [IEEE 754-1985] conforming implementation." The predicte marked with † is "strongly recommended." An implementation should provide "a means of logically negating predicates." Negating all required and recommended predicates adds the 5 predictes marked with ‡.
Two operands are unordered if at least one is a NaN. The invalid operation exception is raised when unordered operands are compared using a predicate involving either *less than* or *greater than*, and not involving *unordered*. A predicate and its logical negation have the same exception behavior.

From Format \ To Format	W	L	S	D
W			†‡	
L			‡	‡
S	†‡	‡		‡
D	‡	‡	‡	

Figure 2.2: The Vector Convert Instruction Empty locations represent convertions not provided by the `vcvt` instruction either because they are redundant (e.g. `vcvt.d.w` can be accomplished by `vcvt.d.l`) or because they are not floating-point convertions (e.g. `vcvt.w.l`). Convertions marked with † are valid for 32-bit VPs. Convertions marked with ‡ are valid for 64-bit VPs.

2.9.3 Vector Floating-Point Arithmetic Instructions

Figure 2.9 shows the support for floating-point arithmetic operations. The support conforms to the IEEE 754-1985 standard for binary floating-point arithmetic. The 32-bit single format and the 64-bit double format are supported for all operations.

Figure 2.1 shows all possible floating-point comparison predicates and those that should be provided by an IEEE 754-1985 compliant implemention. This architecture provides the 7 required and recommended predicates. The EQ, NEQ, LT, LE, and UN predicates are provided directly. The GT and GE predicates may be synthesized by swapping operands and using either LT or LT, respectively.

2.9.4 Vector Floating-Point Convert Instructions

Figure 2.10 shows the support for floating-point arithmetic operations. The support conforms to the IEEE 754-1985 standard for binary floating-point arithmetic. The `vcvt.a.b` instruction uses the default floating-point rounding mode to convert *to* format *a*, *from* format *b*. The set of valid conversions is shown in Figure 2.2. Floating-point values may be converted to to signed integer format using an explicit rounding mode with the `vtrunc`, `vround`, `vceil`, and `vfloor` instructions.

2.9.5 Vector Fixed-Point Arithmetic Instructions

The fixed-point instructions are chosen to provide DSP functionality in a general purpose computing device. Many of the features of a DSP architecture are already provided by this vector architecture or are not needed at all:

- *Fixed* ↔ *Floating-Point Conversion* instructions already exist.
- *Multiple Loads and Stores Per Cycle* may be provided by an implementation with multiple memory units.
- *Auto Increment / Decrement* is built into strided vector memory operations.
- *Circular Addressing* is not needed. The cost of coding circular addressing is amortized over long vector operations.
- *Bit-Reverse Addressing* is not needed, as alternate FFT algorithms may be used.

Name	Action
Truncate	$x.y\cdots \Rightarrow x$
Round Up	$x.0y\cdots \Rightarrow x$ $x.1y\cdots \Rightarrow x+1$
Round to Nearest Even	if $xy.1$ [exactly] then $x0.1 \Rightarrow x0$ $x1.1 \Rightarrow x1+1$ else $x.0y\cdots \Rightarrow x$ $x.1y\cdots \Rightarrow x+1$ endif
Jam	$xy.z\cdots \Rightarrow xw$ where $w = y \mid\mid z \mid\mid \cdots$

Figure 2.3: Fixed-Point Rounding Modes

Some remaining DSP features need special architectural support, namely rounding, saturation, and special multiply instructions. Figure 2.11 and Figure 2.12 list the fixed-point instructions. Many of the operations involve shifting off low-order bits. For these instructions, this shift is accompanied by a rounding step. The rounding mode is one of the four modes described in Figure 2.3.

Since multiply-add is such a frequent operation in fixed-point codes, this operation (along with multipy-subtract) is provided as a basic operation, though it can be synthesized from more basic instructions. Note that fixed-point multiply instructions are different from normal integer multiply low or multiply high. Multiply low and high multiply their entire n-bit operands, and select either the lower or upper half of the 2n-bit result. In contrast, the fixed-point multiply instructions multiply either the lower or upper n/2-bit halves of the n-bit operands to produce an n-bit result.

2.9.6 Vector Logical Flag Instructions

Figure 2.13 shows the simple flag processing instructions. This includes the standard logical operations plus instructions to set and clear registers. The scalar-vector flag instructions take their scalar operand from a vector scalar register. This register is not treated as a bit vector; the operand is false if the register is zero, and true otherwise. The `vfmf8` and `vfmt8` move blocks of 8 flag registers to and from a vector register. These instructions exist in order to save and restore the flag register set efficiently for context switching. None of these instructions operate under mask, but they all operate under vector length.

2.9.7 Flag Processing Instructions

Figure 2.14 lists the more complex flag processing instructions. Some of these instructions – `vfpop`, `vfff1`, `vffl1` – place information about a flag register in a scalar register. Others – `vfsetbf`, `vfsetif`, `vfsetof` – perform complex operations directly on the flag registers. The remaining instructions – `viota` and `vciota` – are used to generate index vectors in a vector register from a flag register.

2.9.8 Vector Processing Instructions

Figure 2.15 and Figure 2.16 list the more complex vector processing instructions. All of these instructions are either not masked, or use the vector mask in an unusual manner.

Many of these instructions replicate functionality in order to gain performance. The `vcompress` and `vexpand` instructions can perform any operation that can be done with `vins.vv`, `vext.vv`, `vhalfup`, `vhalfdn`, and `vexthalf`. Their generality comes at a performance cost, so more restricted instructions are provided for specific applications. The `vhalfup` and `vhalfdn` can be used to perform butterfly permutations. The `vexthalf` instruction is designed specifically to accelerate reductions.

2.9.9 Vector Memory Instructions

The instructions to transfer data between vector registers and memory are listed in Figure 2.17 and Figure 2.18. There are two categories of vector memory instructions: *strided* and *indexed*. A strided load takes a base address and a signed stride, and loads a vector of values starting at the base address, where each element is seperated by the stride amount. The base address is taken from the vector control register `vbase`, and the stride is taken from the vector control register `vstride`. *Stride is in units of elements, not bytes.* A special case is *unit stride*, which loads a contiguous vector from memory. Unit stride loads and stores are provided as seperate instructions because they are the most common type of strided access.

Note that both negative and zero strides are legal. Zero stride is the most obvious case where the order of operations in a strided access can be observed. In fact, virtual memory can cause the order of operations of non-zero stride stores to be observed as well. If a strided store straddles a page boundary, then the indirection provided by virtual memory can create physical address aliases that are not virtual address aliases. Zero-stride is still a special case, since the address aliases existed in the virtual address space, while non-zero-stride aliases only exist in the physical address space. However, all strided stores are treated equally with respect to memory ordering: *All strided stores occur in VP order.*

An indexed memory operation takes a scalar base address, and a vector of byte (not element) offsets. A scalar-vector add is performed to yield a vector of addresses. The elements are loaded / stored using these addresses. Since aliases can exist, it is important to specify the order in which operations occur. The ordered indexed store (`vstxo`) stores elements in element order. The `vstx` instruction store elements in an undefined order.

Every vector memory operation specifies a data width in the opcode. This is the width of data in memory. This width must be less than or equal to the VP width. If it is less than the VP width, then the value is zero-extended for unsigned loads and sign-extended for signed loads. Floating-point values should be loaded with unsigned loads.

The instructions used to transfer flag registers directly to and from memory are also shown in Figure 2.17 and Figure 2.18. The vector flag store instruction stores a single flag register as a packed bit vector in memory, where the bytes are stored in a little-endian manner (i.e. the first 8 bits of the bit vector are at a lower address than the next 8 bits). It should be possible to strip-mine a loop in order to create a contiguous bit vector in memory. The bit vector always begins at the beginning of a half-word, but may end in the middle of another half-word. If a flag store does not have a multiple of 16 bits to store, then it may pad with zeros in order to store a multiple of 16 bits. Note that MINMVL is also 16, and this is not a coincidence. It is advantageous to an implementation to pad and align to larger boundaries, but the boundary should not be more than MINMVL bits.

2.9.10 Coprocessor Interface Instructions

The standard MIPS coprocessor move instructions are used in the vector unit to move data between the MIPS general purpose registers and the vector unit's control and scalar registers. The vector unit occupies coprocessor 2, so `cfc2`, `mtc2`, and `ctc2` are used. `mfc2` is not used because it causes problems with exceptions and the coprocessor interface in VIRAM-1.

2.9.11 Miscellaneous Vector Instructions

Figure 2.20 lists miscellanous user-mode vector instructions that do not fit other categories. This includes an instruction to aid strip-mining (`vsatvl`), instructions to move data between the vector unit's control and scalar register files, instructions to commit speculative operations, and instructions to enforce memory ordering.

2.9.12 Miscellaneous Kernel Vector Instructions

Figure 2.21 lists various instructions that may be executed only in kernel mode. These instructions are used to manipulate the TLB and to flush implementation-specific state when the vector unit is frozen.

Page	Mnemonic	Operation
Integer Arithmetic (29,30)		
51	`vabs`	Absolute Value
57,59	`vadd`	Add
253,255	`vsub`	Subtract
184	`vmullo`	Multiply Low
182,183	`vmulhi`	Multiply High
91,93	`vdiv`	Divide
169,171	`vmod`	Modulus
235	`vsra`	Arithmetic Right Shift
71,73	`vcmp`	Compare
167,168	`vmin`	Minimum
161,162	`vmax`	Maximum
Logical (31)		
64	`vand`	And
198	`vor`	Or
288	`vxor`	Xor
197	`vnor`	Nor
225	`vsll`	Logical Left Shift
237	`vsrl`	Logical Right Shift
Floating-Point Arithmetic (32,33)		
53	`vabs`	Absolute Value
60	`vadd`	Add
257	`vsub`	Subtract
178	`vmul`	Multiply
157	`vmadd`	Multiply Add
174	`vmsub`	Multiply Subtract
189	`vnmadd`	Negative Multiply Add
193	`vnmsub`	Negative Multiply Subtract
95	`vdiv`	Divide
231	`vsqrt`	Square Root
209	`vrsqrt`	Reciprocal Square Root
199	`vrecip`	Reciprocal
185	`vneg`	Negate
75	`vcmp`	Compare
80	`vcvt`	Convert
266	`vtrunc`	Truncate
203	`vround`	Round
65	`vceil`	Ceiling
119	`vfloor`	Floor
Fixed-Point Arithmetic (34,35)		
218,222,220	`vsat`	Saturate
214,216	`vsadd`	Saturating Add
241,243	`vssub`	Saturating Subtract
239,240	`vsrr`	Shift Right And Round
227,229	`vsls`	Saturating Left Shift
301,303	`vxumul`	Multiply Upper Halves
284,286	`vxlmul`	Multiply Lower Halves
289,292	`vxumadd`	Multiply Add Upper Halves
295,298	`vxumsub`	Multiply Subtract Upper Halves
272,275	`vxlmadd`	Multiply Add Lower Halves
278,281	`vxlmsub`	Multiply Subtract Lower Halves
Coprocessor Interface (42)		
48	`cfc2`	Control From Cop2
50	`mtc2`	Move To Cop2
49	`ctc2`	Control To Cop2

Page	Mnemonic	Operation
Flag Logical (36)		
110	`vfand`	And
128	`vfor`	Or
138	`vfxor`	Xor
127	`vfnor`	Nor
111	`vfclr`	Clear
132	`vfset`	Set
Flag Processing (37)		
142	`viota`	Iota
70	`vciota`	Continuous Iota
130	`vfpop`	Population Count
113	`vfff1`	Find First One
114	`vffl1`	Find Last One
133	`vfsetbf`	Set Before First One
134	`vfsetif`	Set Including First One
135	`vfsetof`	Set Only First One
126	`vfmt8`	Move To 8
125	`vfmf8`	Move From 8
112	`vfclr8`	Clear 8
129	`vfor8`	Or 8
Vector Processing (38,39)		
139,140	`vins`	Vector Insert
102,101,100	`vext`	Vector Extract
79	`vcompress`	Compress
99	`vexpand`	Expand
165	`vmerge`	Merge
115	`vfins`	Scalar Insert
103	`vexthalf`	Extract Half
104	`vhalf`	Half
108	`vhalfup`	Half Up
106	`vhalfdn`	Half Down
Miscellaneous User (43)		
164	`vmcts`	Move Control To Scalar
173	`vmstc`	Move Scalar To Control
224	`vsatvl`	Saturate Vector Length
56	`vacommit`	Commit Speculative Arithmetic
163	`vmcommit`	Commit Speculative Arithmetic
261	`vsync`	Sync
213	`vrsync`	Register Sync
Miscellaneous Kernel (44)		
262	`vtlbp`	TLB Probe
263	`vtlbr`	TLB Read
264	`vtlbwi`	TLB Write Indexed
265	`vtlbwr`	TLB Write Random
131	`vflush`	Flush Vector Unit
Load (40)		
116	`vfld`	Flag Load
143,146	`vld`	Unit Stride Load
148,151	`vlds`	Variable Stride Load
153,155	`vldx`	Indexed Load
Store (41)		
136	`vfst`	Flag Store
245	`vst`	Unit Stride Store
247	`vsts`	Variable Stride Store
249	`vstx`	Unordered Indexed Store
251	`vstxo`	Ordered Indexed Store

Figure 2.4: Vector Instruction Set Summary Each sub-block within this table expands into the more detailed table(s) on the page(s) specified in the sub-block title. The instructions are defined in full on the page(s) named in the first column.

Qualifier	*Meaning*	*Notes*
op.vv *op*.vs *op*.sv	Vector-Vector Vector-Scalar Scalar-Vector	Vector arithmetic instructions may take up to one source operand from a scalar register. A vector-vector operation takes two vector source operands; a vector-scalar opeation takes its second operand from the scalar register file; a scalar-vector operation takes its first operand from the scalar register file.
op.b *op*.h *op*.w *op*.l	1B Byte 2B Halfword 4B Word 8B Longword	The saturate instruction, floating-point convert instructions, and all vector memory instructions need to specify the width of integer data.
op.u	Unsigned	Where sensible, integer arithmetic operations are provided in both signed an unsigned versions. The unsigned versions are identified by applying this qualifier.
op.s *op*.d	4B Single FP Format 8B Double FP Format	Floating-point instructions need to specify data type.
op.1	Use $\mathtt{vf}_{\mathtt{mask1}}$ as the mask.	By default, the vector mask is taken from $\mathtt{vf}_{\mathtt{mask0}}$. This qualifier enables $\mathtt{vf}_{\mathtt{mask1}}$ as vector mask.
op.lt *op*.eq *op*.le *op*.neq *op*.un	Less Than Equal Greater Than Less Than Or Equal Greater Than Or Equal Not Equal Unordered	These are the prediates used by the integer and floating-point compare instructionss.

Figure 2.5: Instruction Qualifiers

Page	*Operation*	*Assembly*	*Summary*
51	Absolute Value	vabs[.1] vr$_{dest}$, vr$_{src}$	Each unmasked VP writes into vr$_{dest}$ the absolute value of vr$_{src}$.
57	Add	vadd.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vadd.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the signed integer sum of vs$_{src1}$/vr$_{src1}$ and vr$_{src2}$.
253	Subtract	vsub.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vsub.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$ vsub.vs[.1] vr$_{dest}$, vr$_{src1}$, vs$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the signed integer subtraction of vs$_{src1}$/vr$_{src1}$ and vs$_{src2}$/vr$_{src2}$, where at least one source is a vector.
184	Multiply Low	vmullo.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vmullo.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the lower half of the full-VP-width integer product of vs$_{src1}$/vr$_{src1}$ and vr$_{src2}$.
182	Multiply High	vmulhi.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vmulhi.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the upper half of the full-VP-width signed integer product of vs$_{src1}$/vr$_{src1}$ and vr$_{src2}$.
91	Divide	vdiv.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vdiv.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$ vdiv.vs[.1] vr$_{dest}$, vr$_{src1}$, vs$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the signed integer quotient of vs$_{src1}$/vr$_{src1}$ and vs$_{src2}$/vr$_{src2}$, where at least one source is a vector.
169	Modulus	vmod.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vmod.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$ vmod.vs[.1] vr$_{dest}$, vr$_{src1}$, vs$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the signed integer modulus of vs$_{src1}$/vr$_{src1}$ and vs$_{src2}$/vr$_{src2}$, where at least one source is a vector.
235	Arithmetic Right Shift	vsra.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vsra.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$ vsra.vs[.1] vr$_{dest}$, vr$_{src1}$, vs$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the signed integer contents of vs$_{src1}$/vr$_{src1}$ right-shifted by the number of bits specified by vs$_{src2}$/vr$_{src2}$, where at least one source is a vector. The shift amount is the b-bit unsigned integer taken from the low-order end of vs$_{src2}$/vr$_{src2}$, where 2^b equals the VP width in bits. The result is sign-extended.
71	Compare	vcmp{.eq, .neq}.vv[.1] vf$_{dest}$, vr$_{src1}$, vr$_{src2}$ vcmp{.eq, .neq}.sv[.1] vf$_{dest}$, vs$_{src1}$, vr$_{src2}$ vcmp{.lt, .le}.vv[.1] vf$_{dest}$, vr$_{src1}$, vr$_{src2}$ vcmp{.lt, .le}.sv[.1] vf$_{dest}$, vs$_{src1}$, vr$_{src2}$ vcmp{.lt, .le}.vs[.1] vf$_{dest}$, vr$_{src1}$, vs$_{src2}$	Each unmasked VP writes into vf$_{dest}$ the signed integer comparison of vs$_{src1}$/vr$_{src1}$ and vs$_{src2}$/vr$_{src2}$, where at least one source is a vector. A true predicate yields 1; a false predicate yields 0.
167	Minimum	vmin.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vmin.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the lesser of vs$_{src1}$/vr$_{src1}$ and vr$_{src2}$, treated as signed integers.
161	Maximum	vmax.vv[.1] vr$_{dest}$, vr$_{src1}$, vr$_{src2}$ vmax.sv[.1] vr$_{dest}$, vs$_{src1}$, vr$_{src2}$	Each unmasked VP writes into vr$_{dest}$ the greater of vs$_{src1}$/vr$_{src1}$ and vr$_{src2}$, treated as signed integers.

Figure 2.6: Signed Vector Integer Arithmetic Instructions

Page	*Operation*	*Assembly*	*Summary*
59	Add	$\texttt{vadd.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vr}_{dest}$ the unsigned integer sum of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vr}_{src2}$.
255	Subtract	$\texttt{vsub.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2} \\ \texttt{.vs[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vs}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vr}_{dest}$ the unsigned integer subtraction of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vs}_{src2}$/$\texttt{vr}_{src2}$, where at least one source is a vector.
183	Multiply High	$\texttt{vmulhi.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vr}_{dest}$ the upper half of the full-VP-width unsigned integer product of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vr}_{src2}$.
93	Divide	$\texttt{vdiv.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2} \\ \texttt{.vs[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vs}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vr}_{dest}$ the unsigned integer quotient of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vs}_{src2}$/$\texttt{vr}_{src2}$, where at least one source is a vector.
171	Modulus	$\texttt{vmod.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2} \\ \texttt{.vs[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vs}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vr}_{dest}$ the unsigned integer modulus of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vs}_{src2}$/$\texttt{vr}_{src2}$, where at least one source is a vector.
73	Compare	$\texttt{vcmp.u}\begin{Bmatrix}\texttt{.lt} \\ \texttt{.le}\end{Bmatrix}\begin{cases}\texttt{.vv[.1]} & \texttt{vf}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vf}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2} \\ \texttt{.vs[.1]} & \texttt{vf}_{dest}\texttt{, vr}_{src1}\texttt{, vs}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vf}_{dest}$ the unsigned integer comparison of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vs}_{src2}$/$\texttt{vr}_{src2}$, where at least one source is a vector. A true predicate yields 1; a false predicate yields 0.
168	Minimum	$\texttt{vmin.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vr}_{dest}$ the lesser of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vr}_{src2}$, treated as unsigned integers.
162	Maximum	$\texttt{vmax.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{, vr}_{src1}\texttt{, vr}_{src2} \\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{, vs}_{src1}\texttt{, vr}_{src2}\end{cases}$	Each unmasked VP writes into $\texttt{vr}_{dest}$ the greater of $\texttt{vs}_{src1}$/$\texttt{vr}_{src1}$ and $\texttt{vr}_{src2}$, treated as unsigned integers.

Figure 2.7: Unsigned Vector Integer Arithmetic Instructions

Page	*Operation*	*Assembly*	*Summary*
64	And	`vand` `.vv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vr}_{src2}$ `vand` `.sv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vs}_{src1}$, $\mathtt{vr}_{src2}$	Each VP writes into $\mathtt{vr}_{dest}$ the bit-wise logical *and* of $\mathtt{vs}_{src1}$/$\mathtt{vr}_{src1}$ and $\mathtt{vr}_{src2}$.
198	Or	`vor` `.vv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vr}_{src2}$ `vor` `.sv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vs}_{src1}$, $\mathtt{vr}_{src2}$	Each VP writes into $\mathtt{vr}_{dest}$ the bit-wise logical *or* of $\mathtt{vs}_{src1}$/$\mathtt{vr}_{src1}$ and $\mathtt{vr}_{src2}$.
288	Xor	`vxor` `.vv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vr}_{src2}$ `vxor` `.sv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vs}_{src1}$, $\mathtt{vr}_{src2}$	Each VP writes into $\mathtt{vr}_{dest}$ the bit-wise logical *xor* of $\mathtt{vs}_{src1}$/$\mathtt{vr}_{src1}$ and $\mathtt{vr}_{src2}$.
197	Nor	`vnor` `.vv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vr}_{src2}$ `vnor` `.sv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vs}_{src1}$, $\mathtt{vr}_{src2}$	Each VP writes into $\mathtt{vr}_{dest}$ the bit-wise logical *nor* of $\mathtt{vs}_{src1}$/$\mathtt{vr}_{src1}$ and $\mathtt{vr}_{src2}$.
225	Logical Left Shift	`vsll` `.vv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vr}_{src2}$ `vsll` `.sv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vs}_{src1}$, $\mathtt{vr}_{src2}$ `vsll` `.vs[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vs}_{src2}$	Each unmasked VP writes into $\mathtt{vr}_{dest}$ the signed integer contents of $\mathtt{vs}_{src1}$/$\mathtt{vr}_{src1}$ left-shifted by the number of bits specified by $\mathtt{vs}_{src2}$/$\mathtt{vr}_{src2}$, where at least one source is a vector. The shift amount is the b-bit unsigned integer taken from the low-order end of $\mathtt{vs}_{src2}$/$\mathtt{vr}_{src2}$, where 2^b equals the VP width in bits. The result is zero-filled.
237	Logical Right Shift	`vsrl` `.vv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vr}_{src2}$ `vsrl` `.sv[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vs}_{src1}$, $\mathtt{vr}_{src2}$ `vsrl` `.vs[.1]` $\mathtt{vr}_{dest}$, $\mathtt{vr}_{src1}$, $\mathtt{vs}_{src2}$	Each unmasked VP writes into $\mathtt{vr}_{dest}$ the signed integer contents of $\mathtt{vs}_{src1}$/$\mathtt{vr}_{src1}$ right-shifted by the number of bits specified by $\mathtt{vs}_{src2}$/$\mathtt{vr}_{src2}$, where at least one source is a vector. The shift amount is the b-bit unsigned integer taken from the low-order end of $\mathtt{vs}_{src2}$/$\mathtt{vr}_{src2}$, where 2^b equals the VP width in bits. The result is zero-extended.

Figure 2.8: Vector Logical Instructions

Page	*Operation*	*Assembly*	*Summary*
53	Absolute Value	$\mathtt{vabs}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src}$	Each unmasked VP places the floating-point absolute value of $\mathtt{vr}_{src}$ into $\mathtt{vr}_{dest}$.
60	Add	$\mathtt{vadd}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$	Each unmasked VP places the floating-point sum of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$ and $\mathtt{vr}_{src2}$ into $\mathtt{vr}_{dest}$.
257	Subtract	$\mathtt{vsub}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.vs}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vs}_{src2}\end{array}\right.$	Each unmasked VP places the floating-point difference of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$ and $\mathtt{vr}_{src2}/\mathtt{vf}_{src2}$ into $\mathtt{vr}_{dest}$, where at least one souce is a vector.
178	Multiply	$\mathtt{vmul}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$	Each unmasked VP places the floating-point product of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$ and $\mathtt{vr}_{src2}$ into $\mathtt{vr}_{dest}$.
157	Multiply Add	$\mathtt{vmadd}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$	Each unmasked VP adds the floating-point product of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$ and $\mathtt{vr}_{src2}$ into $\mathtt{vr}_{dest}$.
174	Multiply Subtract	$\mathtt{vmsub}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$	Each unmasked VP subtracts $\mathtt{vr}_{dest}$ from the floating-point product of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$.
189	Negative Multiply Add	$\mathtt{vnmadd}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$	Each unmasked VP adds the floating-point product of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$ and $\mathtt{vr}_{src2}$ into $\mathtt{vr}_{dest}$, and negates the result.
193	Negative Multiply Subtract	$\mathtt{vnmsub}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$	Each unmasked VP subtracts $\mathtt{vr}_{dest}$ from the floating-point product of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$, and negates the result.
95	Divide	$\mathtt{vdiv}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.vs}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vs}_{src2}\end{array}\right.$	Each unmasked VP places the floating-point quotient of $\mathtt{vr}_{src1}/\mathtt{vs}_{src1}$ and $\mathtt{vr}_{src2}/\mathtt{vf}_{src2}$ into $\mathtt{vr}_{dest}$, where at least one souce is a vector.
231	Square Root	$\mathtt{vsqrt}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src}$	Each unmasked VP places into $\mathtt{vr}_{dest}$ the floating-point square root of $\mathtt{vr}_{src}$.
209	Reciprocal Square Root	$\mathtt{vrsqrt}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src}$	Each unmasked VP places into $\mathtt{vr}_{dest}$ the floating-point reciprocal square root of $\mathtt{vr}_{src}$.
199	Reciprocal	$\mathtt{vrecip}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src}$	Each unmasked VP places into $\mathtt{vr}_{dest}$ the floating-point reciprocal of $\mathtt{vr}_{src}$.
185	Negate	$\mathtt{vneg}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}[.1]\ \mathtt{vr}_{dest},\ \mathtt{vr}_{src}$	Each unmasked VP places into $\mathtt{vr}_{dest}$ the floating-point square root of $\mathtt{vr}_{src}$.
75	Compare	$\mathtt{vcmp}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{c}\mathtt{.f}\\ \mathtt{.sf}\\ \mathtt{.un}\\ \mathtt{.ngle}\\ \mathtt{.eq}\\ \mathtt{.seq}\\ \mathtt{.ueq}\\ \mathtt{.ngl}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vf}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vf}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$ $\mathtt{vcmp}\left\{\begin{array}{l}\mathtt{.s}\\ \mathtt{.d}\end{array}\right\}\left\{\begin{array}{c}\mathtt{.olt}\\ \mathtt{.lt}\\ \mathtt{.ult}\\ \mathtt{.nge}\\ \mathtt{.ole}\\ \mathtt{.le}\\ \mathtt{.ule}\\ \mathtt{.ngt}\end{array}\right\}\left\{\begin{array}{l}\mathtt{.vv}[.1]\ \mathtt{vf}_{dest},\ \mathtt{vr}_{src1},\ \mathtt{vr}_{src2}\\ \mathtt{.sv}[.1]\ \mathtt{vf}_{dest},\ \mathtt{vs}_{src1},\ \mathtt{vr}_{src2}\end{array}\right.$	Each unmasked VP writes into $\mathtt{vf}_{dest}$ the floating-point comparison of $\mathtt{vs}_{src1}/\mathtt{vr}_{src1}$ and $\mathtt{vs}_{src2}/\mathtt{vr}_{src2}$, where at least one source is a vector. A true predicate yields 1; a false predicate yields 0.

Figure 2.9: Vector Floating-Point Instructions

Page	*Operation*	*Assembly*	*Summary*
80	Convert	$\texttt{vcvt.s.}\begin{Bmatrix}\texttt{.d}\\ \texttt{.w}\\ \texttt{.l}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$ $\texttt{vcvt.d.}\begin{Bmatrix}\texttt{.s}\\ \texttt{.l}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$ $\texttt{vcvt.w.}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$ $\texttt{vcvt.l.}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$	Each unmasked VP places into $\texttt{vr}_{dest}$ the result of converting $\texttt{vr}_{src}$ from one integer or floating-point format to another.
266	Truncate	$\texttt{vtrunc}\begin{Bmatrix}\texttt{.w}\\ \texttt{.l}\end{Bmatrix}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$	Each unmasked VP places into $\texttt{vr}_{dest}$ the result of converting $\texttt{vr}_{src}$ from a floating-point format to a signed integer format, using the *truncate* rounding mode.
203	Round	$\texttt{vround}\begin{Bmatrix}\texttt{.w}\\ \texttt{.l}\end{Bmatrix}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$	Each unmasked VP places into $\texttt{vr}_{dest}$ the result of converting $\texttt{vr}_{src}$ from a floating-point format to a signed integer format, using the *round* rounding mode.
65	Ceiling	$\texttt{vceil}\begin{Bmatrix}\texttt{.w}\\ \texttt{.l}\end{Bmatrix}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$	Each unmasked VP places into $\texttt{vr}_{dest}$ the result of converting $\texttt{vr}_{src}$ from a floating-point format to a signed integer format, using the *ceiling* rounding mode.
119	Floor	$\texttt{vfloor}\begin{Bmatrix}\texttt{.w}\\ \texttt{.l}\end{Bmatrix}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}[.1]\ \texttt{vr}_{dest},\ \texttt{vr}_{src}$	Each unmasked VP places into $\texttt{vr}_{dest}$ the result of converting $\texttt{vr}_{src}$ from a floating-point format to a signed integer format, using the *floor* rounding mode.

Figure 2.10: Vector Floating-Point Convert Instructions

Page	*Operation*	*Assembly*	*Summary*
218	Saturate	$\texttt{vsat}\left\{\begin{array}{l}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\end{array}\right\}$[.1] vr_{dest}, vr_{src}	Each unmasked VP places into vr_{dest} the result of saturating vr_{src} to a signed integer narrower than the VP width. The result is sign-extended to the VP width.
220	Saturate	$\texttt{vsat.su}\left\{\begin{array}{l}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{array}\right\}$[.1] vr_{dest}, vr_{src}	Each unmasked VP places into vr_{dest} the result of saturating vr_{src} from a signed VP width value to an unsigned value that is as wide or narrower than the VP width. The result is zero-extended to the VP width.
214	Saturating Add	$\texttt{vsadd}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\end{array}\right.$	Each unmasked VP writes into vr_{dest} the signed integer sum of vs_{src1}/vr_{src1} and vr_{src2}. The sum saturates to the VP width instead of overflowing.
241	Saturating Subtract	$\texttt{vssub}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\\ \texttt{.vs[.1]}\ vr_{dest},\ vr_{src1},\ vs_{src2}\end{array}\right.$	Each unmasked VP writes into vr_{dest} the signed integer subtraction of vs_{src1}/vr_{src1} and vs_{src2}/vr_{src2}, where at least one source is a vector. The difference saturates to the VP width instead of overflowing.
239	Shift Right And Round	vsrr[.1] vr_{dest}, vr_{src}	Each unmasked VP writes into vr_{dest} the right arithmetic shift of vr_{src}. The result is rounded as per the fixed-point rounding mode. The shift amount is taken from vc_{vshamt}.
227	Saturating Left Shift	vsls[.1] vr_{dest}, vr_{src}	Each unmasked VP writes into vr_{dest} the signed saturating left shift of vr_{src}. The shift amount is taken from vc_{vshamt}.
301	Multiply Upper Halves	$\texttt{vxumul}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\end{array}\right.$	Each unmasked VP computes the signed integer product of the upper halves of vs_{src1}/vr_{src1} and vr_{src2}. This result is written into vr_{dest} after an arithmetic right shift and fixed-point round. The shift amount is taken from vc_{vshamt}.
284	Multiply Lower Halves	$\texttt{vxlmul}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\end{array}\right.$	Each unmasked VP computes the signed integer product of the lower halves of vs_{src1}/vr_{src1} and vr_{src2}. This result is written into vr_{dest} after an arithmetic right shift and fixed-point round. The shift amount is taken from vc_{vshamt}.
289	Multiply Add Upper Halves	$\texttt{vxumadd}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\end{array}\right.$	Each unmasked VP computes the signed integer product of the upper halves of vs_{src1}/vr_{src1} and vr_{src2}. This result is added into vr_{dest} after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from vc_{vshamt}.
295	Multiply Subtract Upper Halves	$\texttt{vxumsub}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\end{array}\right.$	Each unmasked VP computes the signed integer product of the upper halves of vs_{src1}/vr_{src1} and vr_{src2}. This result is subtracted from vr_{dest} after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from vc_{vshamt}.
272	Multiply Add Lower Halves	$\texttt{vxlmadd}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\end{array}\right.$	Each unmasked VP computes the signed integer product of the lower halves of vs_{src1}/vr_{src1} and vr_{src2}. This result is added into vr_{dest} after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from vc_{vshamt}.
278	Multiply Subtract Lower Halves	$\texttt{vxlmsub}\left\{\begin{array}{l}\texttt{.vv[.1]}\ vr_{dest},\ vr_{src1},\ vr_{src2}\\ \texttt{.sv[.1]}\ vr_{dest},\ vs_{src1},\ vr_{src2}\end{array}\right.$	Each unmasked VP computes the signed integer product of the lower halves of vs_{src1}/vr_{src1} and vr_{src2}. This result is subtracted from vr_{dest} after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from vc_{vshamt}.

Figure 2.11: Signed Vector Fixed-Point Instructions

Page	*Operation*	*Assembly*	*Summary*
222	Saturate	`vsat.u` {`.b`, `.h`, `.w`}[`.1`] `vr`$_{dest}$`, vr`$_{src}$	Each unmasked VP places into `vr`$_{dest}$ the result of saturating `vr`$_{src}$ to an unsigned integer narrower than the VP width. The result is zero-extended to the VP width.
216	Saturating Add	`vsadd.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$}	Each unmasked VP writes into `vr`$_{dest}$ the unsigned integer sum of `vs`$_{src1}$/`vr`$_{src1}$ and `vr`$_{src2}$. The sum saturates to the VP width instead of overflowing.
243	Saturating Subtract	`vssub.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$; `.vs`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vs`$_{src2}$}	Each unmasked VP writes into `vr`$_{dest}$ the unsigned integer subtraction of `vs`$_{src1}$/`vr`$_{src1}$ and `vs`$_{src2}$/`vr`$_{src2}$, where at least one source is a vector. The difference saturates to zero instead of underflowing.
240	Shift Right And Round	`vsrr.u`[`.1`] `vr`$_{dest}$`, vr`$_{src}$	Each unmasked VP writes into `vr`$_{dest}$ the right logical shift of `vr`$_{src}$. The result is rounded as per the fixed-point rounding mode. The shift amount is taken from `vc`$_{vshamt}$.
229	Saturating Left Shift	`vsls.u`[`.1`] `vr`$_{dest}$`, vr`$_{src}$	Each unmasked VP writes into `vr`$_{dest}$ the unsigned saturating left shift of `vr`$_{src}$. The shift amount is taken from `vc`$_{vshamt}$.
303	Multiply Upper Halves	`vxumul.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$}	Each unmasked VP computes the unsigned integer product of the upper halves of `vs`$_{src1}$/`vr`$_{src1}$ and `vr`$_{src2}$. This result is written into `vr`$_{dest}$ after a logical right shift and fixed-point round. The shift amount is taken from `vc`$_{vshamt}$.
286	Multiply Lower Halves	`vxlmul.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$}	Each unmasked VP computes the unsigned integer product of the lower halves of `vs`$_{src1}$/`vr`$_{src1}$ and `vr`$_{src2}$. This result is written into `vr`$_{dest}$ after a logical right shift and fixed-point round. The shift amount is taken from `vc`$_{vshamt}$.
292	Multiply Add Upper Halves	`vxumadd.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$}	Each unmasked VP computes the unsigned integer product of the upper halves of `vs`$_{src1}$/`vr`$_{src1}$ and `vr`$_{src2}$. This result is added into `vr`$_{dest}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from `vc`$_{vshamt}$.
298	Multiply Subtract Upper Halves	`vxumsub.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$}	Each unmasked VP computes the unsigned integer product of the upper halves of `vs`$_{src1}$/`vr`$_{src1}$ and `vr`$_{src2}$. This result is subtracted from `vr`$_{dest}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from `vc`$_{vshamt}$.
275	Multiply Add Lower Halves	`vxlmadd.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$}	Each unmasked VP computes the unsigned integer product of the lower halves of `vs`$_{src1}$/`vr`$_{src1}$ and `vr`$_{src2}$. This result is added into `vr`$_{dest}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from `vc`$_{vshamt}$.
281	Multiply Subtract Lower Halves	`vxlmsub.u` {`.vv`[`.1`] `vr`$_{dest}$`, vr`$_{src1}$`, vr`$_{src2}$; `.sv`[`.1`] `vr`$_{dest}$`, vs`$_{src1}$`, vr`$_{src2}$}	Each unmasked VP computes the unsigned integer product of the lower halves of `vs`$_{src1}$/`vr`$_{src1}$ and `vr`$_{src2}$. This result is subtracted from `vr`$_{dest}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from `vc`$_{vshamt}$.

Figure 2.12: Unsigned Fixed-Point Instructions

Page	*Operation*	*Assembly*	*Summary*
110	And	`vfand` $\begin{cases}\texttt{.vv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vf}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\\ \texttt{.sv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vs}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\end{cases}$	Each VP writes into $\mathbf{vf}_{dest}$ the logical *and* of $\mathbf{vs}_{src1}$/$\mathbf{vf}_{src1}$ and $\mathbf{vf}_{src2}$. This instruction is not masked.
128	Or	`vfor` $\begin{cases}\texttt{.vv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vf}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\\ \texttt{.sv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vs}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\end{cases}$	Each VP writes into $\mathbf{vf}_{dest}$ the logical *or* of $\mathbf{vs}_{src1}$/$\mathbf{vf}_{src1}$ and $\mathbf{vf}_{src2}$. This instruction is not masked.
138	Xor	`vfxor` $\begin{cases}\texttt{.vv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vf}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\\ \texttt{.sv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vs}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\end{cases}$	Each VP writes into $\mathbf{vf}_{dest}$ the logical *xor* of $\mathbf{vs}_{src1}$/$\mathbf{vf}_{src1}$ and $\mathbf{vf}_{src2}$. This instruction is not masked.
127	Nor	`vfnor` $\begin{cases}\texttt{.vv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vf}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\\ \texttt{.sv}\ \mathbf{vf}_{dest}\texttt{,}\ \mathbf{vs}_{src1}\texttt{,}\ \mathbf{vf}_{src2}\end{cases}$	Each VP writes into $\mathbf{vf}_{dest}$ the logical *nor* of $\mathbf{vs}_{src1}$/$\mathbf{vf}_{src1}$ and $\mathbf{vf}_{src2}$. This instruction is not masked.
111	Clear	`vfclr` $\mathbf{vf}_{dest}$	Each VP writes zero into $\mathbf{vf}_{dest}$. This instruction is not masked.
132	Set	`vfset` $\mathbf{vf}_{dest}$	Each VP writes one into $\mathbf{vf}_{dest}$. This instruction is not masked, but is subject to vector length.

Figure 2.13: Vector Flag Logical Instructions

Page	*Operation*	*Assembly*	*Summary*
142	Iota	`viota` $\mathbf{vr}_{dest}$, $\mathbf{vf}_{src}$	The list of VPs with bits set in $\mathbf{vf}_{src}$ is placed in $\mathbf{vr}_{dest}$. This is the compressed index vector of $\mathbf{vf}_{src}$. This instruction is not masked.
70	Continuous Iota	`vciota` $\mathbf{vr}_{dest}$, $\mathbf{vf}_{src}$	The continuous index vector of $\mathbf{vf}_{src1}$ is placed in register $\mathbf{vr}_{dest}$. The continuous index vector's value at vp_i is equal to the population count of flag values for $\mathrm{vp}_0 \cdots \mathrm{vp}_{i-1}$. This instruction is not masked.
130	Population Count	`vfpop` $\mathbf{vs}_{dest}$, $\mathbf{vf}_{src}$	The population count of $\mathbf{vf}_{src}$ is placed in $\mathbf{vs}_{dest}$. This instruction is not masked.
113	Find First One	`vfff1` $\mathbf{vs}_{dest}$, $\mathbf{vf}_{src}$	The location of the first set bit of $\mathbf{vf}_{src}$ is placed in $\mathbf{vs}_{dest}$. This instruction is not masked. If there is no set bit in $\mathbf{vf}_{src}$, then the vector length is placed in $\mathbf{vs}_{dest}$.
114	Find Last One	`vffl1` $\mathbf{vs}_{dest}$, $\mathbf{vf}_{src}$	The location of the last set bit of $\mathbf{vf}_{src}$ is placed in $\mathbf{vs}_{dest}$. This instruction is not masked. If there is no set bit in $\mathbf{vf}_{src}$, then the vector length is placed in $\mathbf{vs}_{dest}$.
133	Set Before First One	`vfsetbf` $\mathbf{vf}_{dest}$, $\mathbf{vf}_{src}$	Regiser $\mathbf{vf}_{dest}$ is filled with ones up to and not including the first set bit in $\mathbf{vf}_{src}$. Remaining positions in $\mathbf{vf}_{dest}$ are cleared. If $\mathbf{vf}_{src}$ contains no set bits, $\mathbf{vf}_{dest}$ is set to all ones. This instruction is not masked.
134	Set Including First One	`vfsetif` $\mathbf{vf}_{dest}$, $\mathbf{vf}_{src}$	Regiser $\mathbf{vf}_{dest}$ is filled with ones up to and including the first set bit in $\mathbf{vf}_{src}$. Remaining positions in $\mathbf{vf}_{dest}$ are cleared. If $\mathbf{vf}_{src}$ contains no set bits, $\mathbf{vf}_{dest}$ is set to all ones. This instruction is not masked.
135	Set Only First One	`vfsetof` $\mathbf{vf}_{dest}$, $\mathbf{vf}_{src}$	Regiser $\mathbf{vf}_{dest}$ is filled with zeros except for the position of the first set bit in $\mathbf{vf}_{src}$. If $\mathbf{vf}_{src}$ contains no set bits, $\mathbf{vf}_{dest}$ is set to all zeros. This instruction is not masked.
126	Move To 8	`vfmt8` $\mathbf{vf}_{dest}$, $\mathbf{vr}_{src}$	Registers $\mathbf{vf}_{dest} \cdots \mathbf{vf}_{dest+7}$ are set with the contents of $\mathbf{vr}_{src}$. This instruction is not masked, but is subject to vector length. The flag register specifier must be a multiple of 8.
125	Move From 8	`vfmf8` $\mathbf{vr}_{dest}$, $\mathbf{vf}_{src}$	Registers $\mathbf{vf}_{src} \cdots \mathbf{vf}_{src+7}$ are copied into $\mathbf{vr}_{dest}$. This instruction is not masked, but is subject to vector length. The flag register specifier must be a multiple of 8.
112	Clear 8	`vfclr8` $\mathbf{vf}_{dest}$	Each VP writes zero into $\mathbf{vf}_{dest} \cdots \mathbf{vf}_{dest+7}$. This instruction is not masked, but is subject to vector length. The flag register specifier must be a multiple of 8.
129	Or 8	`vfor8` $\mathbf{vf}_{dest}$, $\mathbf{vf}_{src}$	Each VP performs an OR-write of $\mathbf{vf}_{src} \cdots \mathbf{vf}_{src+7}$ into $\mathbf{vf}_{dest} \cdots \mathbf{vf}_{dest+7}$. This instruction is not masked, but is subject to vector length. Each flag register specifier must be a multiple of 8.

Figure 2.14: Vector Flag Processing Instructions

Page	*Operation*	*Assembly*	*Summary*
140	Vector Insert	vins.vv $\mathtt{vr_{dest}}$, $\mathtt{vr_{src2}}$	The leading portion of $\mathtt{vr_{src}}$ is inserted into $\mathtt{vr_{dest}}$. $\mathtt{vr_{dest}}$ must be different than $\mathtt{vr_{src}}$. Leading and trailing entries of $\mathtt{vr_{dest}}$ are not touched. The lower $\mathtt{vc_{logmvl}}$ bits of vector control register $\mathtt{vc_{vindex}}$ specifies the starting position in $\mathtt{vr_{dest}}$. The vector length specifies the number of elements to transfer. This instruction is not masked.
139	Scalar Insert	vins.sv $\mathtt{vr_{dest}}$, $\mathtt{vs_{src}}$	The contents of $\mathtt{vs_{src}}$ are written into $\mathtt{vr_{dest}}$ at position $\mathtt{vc_{vindex}}$. The lower $\mathtt{vc_{logmvl}}$ bits of $\mathtt{vc_{vindex}}$ are used. This instruction is not masked and does not use vector length.
102	Vector Extract	vext.vv $\mathtt{vr_{dest}}$, $\mathtt{vr_{src}}$	A portion of $\mathtt{vr_{src}}$ is extracted into the front of $\mathtt{vr_{dest}}$. $\mathtt{vr_{dest}}$ must be different than $\mathtt{vr_{src}}$. Trailing entries of $\mathtt{vr_{dest}}$ are not touched. The lower $\mathtt{vc_{logmvl}}$ bits of vector control register $\mathtt{vc_{vindex}}$ specifies the starting position in $\mathtt{vr_{src}}$. The vector length specifies the number of elements to transfer. This instruction is not masked.
100	Scalar Extract	vext.sv $\mathtt{vs_{dest}}$, $\mathtt{vr_{src2}}$	Element $\mathtt{vc_{vindex}}$ of $\mathtt{vr_{src}}$ is written into $\mathtt{vs_{dest}}$. The lower $\mathtt{vc_{logmvl}}$ bits of $\mathtt{vc_{vindex}}$ are used. The value is sign-extended. This instruction is not masked and does not use vector length.
101	Scalar Extract	vext.u.sv $\mathtt{vs_{dest}}$, $\mathtt{vr_{src}}$	Element $\mathtt{vc_{vindex}}$ of $\mathtt{vr_{src}}$ is written into $\mathtt{vs_{dest}}$. The lower $\mathtt{vc_{logmvl}}$ bits of $\mathtt{vc_{vindex}}$ are used. The value is zero-extended. This instruction is not masked and does not use vector length.
79	Compress	vcompress[.1] $\mathtt{vr_{dest}}$, $\mathtt{vr_{src}}$	All unmasked elements of $\mathtt{vr_{src}}$ are catenated to form a vector whose length is the population count of the mask (subject to vector length). The result is placed at the front of $\mathtt{vr_{dest}}$, leaving trailing elements untouched. $\mathtt{vr_{dest}}$ must be different than $\mathtt{vr_{src}}$.
99	Expand	vexpand[.1] $\mathtt{vr_{dest}}$, $\mathtt{vr_{src}}$	The first n elements of $\mathtt{vr_{src}}$ are written into the unmasked positions of $\mathtt{vr_{dest}}$, where n is the population count of the mask (subject to vector length). Masked positions in $\mathtt{vr_{dest}}$ are not touched. $\mathtt{vr_{dest}}$ must be different than $\mathtt{vr_{src}}$.
165	Merge	vmerge $\left\{\begin{array}{l}\mathtt{.vv[.1]\ vr_{dest},\ vr_{src1},\ vr_{src2}} \\ \mathtt{.sv[.1]\ vr_{dest},\ vs_{src1},\ vr_{src2}} \\ \mathtt{.vs[.1]\ vr_{dest},\ vr_{src1},\ vs_{src2}}\end{array}\right.$	Each VP copies into $\mathtt{vr_{dest}}$ either $\mathtt{vs_{src1}}$/$\mathtt{vr_{src1}}$ if the mask is 0, or $\mathtt{vs_{src2}}$/$\mathtt{vr_{src2}}$ if the mask is 1. At least one source is a vector. Scalar sources are truncated to the virtual processor width.
115	Scalar Insert	vfins $\mathtt{vf_{dest}}$, $\mathtt{vs_{src}}$	The boolean value of $\mathtt{vs_{src}}$ is written into $\mathtt{vf_{dest}}$ at position $\mathtt{vc_{vindex}}$. The lower $\mathtt{vc_{logmvl}}$ bits of $\mathtt{vc_{vindex}}$ are used. This instruction is not masked and does not use vector length.

Figure 2.15: Vector Processing Instructions I

Page	*Operation*	*Assembly*	*Summary*
103	Extract Half	`vexthalf` $\mathbf{vr}_{dest}$, $\mathbf{vr}_{src}$	The top half of $\mathbf{vr}_{src}$ is extracted into the front of $\mathbf{vr}_{dest}$. $\mathbf{vr}_{dest}$ must be different than $\mathbf{vr}_{src}$. Trailing entries of $\mathbf{vr}_{dest}$ are not touched. The vector length specifies the length of $\mathbf{vr}_{src}$. This instruction is not masked. The vector length must be a power of two, and must be at least two.
104	Half	`vhalf` $\mathbf{vr}_{dest}$, $\mathbf{vr}_{src}$	The top half of $\mathbf{vr}_{src}$ is extracted into the front of $\mathbf{vr}_{dest}$. $\mathbf{vr}_{dest}$ must be different than $\mathbf{vr}_{src}$. Trailing entries of $\mathbf{vr}_{dest}$ are not touched. The vector length specifies the length of $\mathbf{vr}_{src}$. This instruction is not masked. The vector length must be a power of two, and must be at least two. After execution, the vector length register is divided by two.
108	Half Up	`vhalfup` $\mathbf{vr}_{dest}$, $\mathbf{vr}_{src}$	This instruction does one half of one step of a butterfly permutation. Paired with `vhalfdn`, it can be used to perform an in-register butterfly permutation. $\mathbf{vr}_{dest}$ must be different than $\mathbf{vr}_{src}$. $\mathbf{vc}_{vindex}$ specifies the log of both the length and the separation of the sub-vectors that are moved from $\mathbf{vr}_{src}$ to $\mathbf{vr}_{dest}$. The sub-vectors are shifted up within each $2 * 2^{\mathbf{vc}_{vindex}}$ block by $2^{\mathbf{vc}_{vindex}}$ elements. This writes half of the elements within each block in $\mathbf{vr}_{dest}$; the remaining elements are not touched. This instruction is not masked. $\mathbf{vc}_{vl}$ must be at least 2 and must be a power of 2. $2^{\mathbf{vc}_{vindex}}$ must be less than $\mathbf{vc}_{vl}$.
106	Half Down	`vhalfdn` $\mathbf{vr}_{dest}$, $\mathbf{vr}_{src}$	This instruction does one half of one step of a butterfly permutation. Paired with `vhalfdn`, it can be used to perform an in-register butterfly permutation. $\mathbf{vr}_{dest}$ must be different than $\mathbf{vr}_{src}$. $\mathbf{vc}_{vindex}$ specifies the log of both the length and the separation of the sub-vectors that are moved from $\mathbf{vr}_{src}$ to $\mathbf{vr}_{dest}$. The sub-vectors are shifted down within each $2 * 2^{\mathbf{vc}_{vindex}}$ block by $2^{\mathbf{vc}_{vindex}}$ elements. This writes half of the elements within each block in $\mathbf{vr}_{dest}$; the remaining elements are not touched. This instruction is not masked. $\mathbf{vc}_{vl}$ must be at least 2 and must be a power of 2. $2^{\mathbf{vc}_{vindex}}$ must be less than $\mathbf{vc}_{vl}$.

Figure 2.16: Vector Processing Instructions II

Page	*Operation*	*Assembly*	*Summary*
116	Flag Load	$\mathtt{vfld}\ \mathtt{vf}_{\mathtt{dest}}[,\mathtt{vbase}[,\mathtt{vinc}]]$	The VPs perform a contiguous vector flag load into $\mathtt{vf}_{\mathtt{dest}}$. $\mathtt{vf}_{\mathtt{dest}}$ should be different than the speculative load fault flag register if the instruction is speculative. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be 16-bit aligned. The bytes are loaded in little-endian order. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. This instruction is not masked.
143	Unit Stride Load	$\mathtt{vld}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\\ \mathtt{.l}\end{array}\right\}[\mathtt{.1}]\ \mathtt{vr}_{\mathtt{dest}}[,\mathtt{vbase}[,\mathtt{vinc}]]$	The VPs perform a contiguous vector load into $\mathtt{vr}_{\mathtt{dest}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be aligned to the width of the data in memory. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The loaded value is sign-extended to the virtual processor width.
146	Unit Stride Load	$\mathtt{vld.u}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\end{array}\right\}[\mathtt{.1}]\ \mathtt{vr}_{\mathtt{dest}}[,\mathtt{vbase}[,\mathtt{vinc}]]$	Operation is identical to `vld`, except that loaded values are zero-extended to the virtual processor width instead of sign-extended.
148	Variable Stride Load	$\mathtt{vlds}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\\ \mathtt{.l}\end{array}\right\}[\mathtt{.1}]\ \mathtt{vr}_{\mathtt{dest}}[,\mathtt{vbase}[,\mathtt{vstride}[,\mathtt{vinc}]]]$	The VPs perform a strided vector load into $\mathtt{vr}_{\mathtt{dest}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be aligned to the width of the data in memory. The *signed* stride is given by `vstride` (default is $\mathtt{vstride}_0$). The stride is in terms of elements, not in terms of bytes. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The loaded value is sign-extended to the virtual processor width.
151	Variable Stride Load	$\mathtt{vlds.u}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\end{array}\right\}[\mathtt{.1}]\ \mathtt{vr}_{\mathtt{dest}}[,\mathtt{vbase}[,\mathtt{vstride}[,\mathtt{vinc}]]]$	Operation is identical to `vlds`, except that loaded values are zero-extended to the virtual processor width instead of sign-extended.
153	Indexed Load	$\mathtt{vldx}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\\ \mathtt{.l}\end{array}\right\}[\mathtt{.1}]\ \mathtt{vr}_{\mathtt{dest}},\ \mathtt{vr}_{\mathtt{offsets}}[,\mathtt{vbase}]$	The VPs perform an indexed vector load into $\mathtt{vr}_{\mathtt{dest}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$). The *signed* offsets are given by $\mathtt{vr}_{\mathtt{offsets}}$. The offsets are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data in memory. The width of each element in memory is given by the opcode. The loaded value is sign-extended to the virtual processor width.
155	Indexed Load	$\mathtt{vldx.u}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\end{array}\right\}[\mathtt{.1}]\ \mathtt{vr}_{\mathtt{dest}},\ \mathtt{vr}_{\mathtt{offsets}}[,\mathtt{vbase}]$	Operation is identical to `vldx`, except that loaded values are zero-extended to the virtual processor width instead of sign-extended.

Figure 2.17: Vector Load Instructions

Page	*Operation*	*Assembly*	*Summary*
136	Flag Store	`vfst` $\mathtt{vf}_{\mathtt{src}}$[, `vbase`[, `vinc`]]	The VPs perform a contiguous vector flag store of $\mathtt{vf}_{\mathtt{src}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be 16-bit aligned. A multiple of 16 bits is written, padding with zeros as necessary. The bytes are stored in little-endian order. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. This instruction is not masked.
245	Unit Stride Store	$\mathtt{vst}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\\ \mathtt{.l}\end{array}\right\}$[.1] $\mathtt{vr}_{\mathtt{src}}$[, `vbase`[, `vinc`]]	The VPs perform a contiguous vector store of $\mathtt{vr}_{\mathtt{src}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be aligned to the width of the data in memory. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The register value is truncated from the VP width to the memory width. The VPs access memory in order.
247	Variable Stride Store	$\mathtt{vsts}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\\ \mathtt{.l}\end{array}\right\}$[.1] $\mathtt{vr}_{\mathtt{src}}$[, `vbase`[, `vstride`[, `vinc`]]]	The VPs perform a contiguous vector store of $\mathtt{vr}_{\mathtt{src}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be aligned to the width of the data in memory. The *signed* stride is given by `vstride` (default is $\mathtt{vstride}_0$). The stride is in terms of elements, not in terms of bytes. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The register value is truncated from the VP width to the memory width. The VPs access memory in order.
249	Unordered Indexed Store	$\mathtt{vstx}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\\ \mathtt{.l}\end{array}\right\}$[.1] $\mathtt{vr}_{\mathtt{src}}$, $\mathtt{vr}_{\mathtt{offsets}}$[, `vbase`]	The VPs perform an indexed vector store of $\mathtt{vr}_{\mathtt{src}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$). The *signed* offsets are given by $\mathtt{vr}_{\mathtt{offsets}}$. The offsets are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data in memory. The register value is truncated from the VP width to the memory width. The stores may be performed in any order.
251	Ordered Indexed Store	$\mathtt{vstxo}\left\{\begin{array}{l}\mathtt{.b}\\ \mathtt{.h}\\ \mathtt{.w}\\ \mathtt{.l}\end{array}\right\}$[.1] $\mathtt{vr}_{\mathtt{src}}$, $\mathtt{vr}_{\mathtt{offsets}}$[, `vbase`]	Operation is identical to `vstx`, except that the VPs access memory in order.

Figure 2.18: Vector Store Instructions

Page	*Operation*	*Assembly*	*Summary*
48	Control From Cop2	`cfc2` $\mathtt{r}_{\mathtt{dest}}$, $\mathtt{vc}_{\mathtt{src}}$	$\mathtt{vc}_{\mathtt{src}}$ is copied into register $\mathtt{r}_{\mathtt{dest}}$.
50	Move To Cop2	`mtc2` $\mathtt{r}_{\mathtt{src}}$, $\mathtt{vs}_{\mathtt{dest}}$	$\mathtt{r}_{\mathtt{src}}$ is copied into register $\mathtt{vs}_{\mathtt{dest}}$.
49	Control To Cop2	`ctc2` $\mathtt{r}_{\mathtt{src}}$, $\mathtt{vc}_{\mathtt{dest}}$	$\mathtt{r}_{\mathtt{src}}$ is copied into register $\mathtt{vc}_{\mathtt{dest}}$. Writing $\mathtt{vc}_{\mathtt{vpw}}$ changes $\mathtt{vc}_{\mathtt{mvl}}$, $\mathtt{vc}_{\mathtt{logmvl}}$, and $\mathtt{vc}_{\mathtt{vsafevl}}$ as a side-effect.

Figure 2.19: Coprocessor Interface Instructions

Page	*Operation*	*Assembly*	*Summary*
164	Move Scalar To Control	`vmstc` $\mathtt{vc}_{\mathtt{dest}}$, $\mathtt{vs}_{\mathtt{src}}$	Register $\mathtt{vs}_{\mathtt{src}}$ is copied to $\mathtt{vc}_{\mathtt{dest}}$. Writing $\mathtt{vc}_{\mathtt{vpw}}$ changes $\mathtt{vc}_{\mathtt{mvl}}$, $\mathtt{vc}_{\mathtt{logmvl}}$, and $\mathtt{vc}_{\mathtt{vsafevl}}$ as a side-effect. This instruction may write only $\mathtt{vc}_{\mathtt{vl}}$, $\mathtt{vc}_{\mathtt{vpw}}$, $\mathtt{vc}_{\mathtt{vshamt}}$, $\mathtt{vc}_{\mathtt{vindex}}$, and $\mathtt{vc}_{\mathtt{vcat}}$.
173	Move Control To Scalar	`vmcts` $\mathtt{vs}_{\mathtt{dest}}$, $\mathtt{vc}_{\mathtt{src}}$	Register $\mathtt{vc}_{\mathtt{src}}$ is copied to $\mathtt{vs}_{\mathtt{dest}}$.
224	Saturate Vector Length	`vsatvl`	The vector length register is saturated to the maximum vector length.
56	Commit Speculative Arithmetic	`vacommit`[**.exact**][.1]	Any pending vector arithmetic exceptions generated by speculative operations are raised. This instruction operates under mask and vector length. The exceptions are precise if either `.exact` is specified, or if $\mathtt{vc}_{\mathtt{vmode}}$ specifies exact exceptions.
163	Commit Speculative Arithmetic	`vmcommit`[**.exact**][.1]	Any pending vector memory exceptions generated by speculative operations are raised. This instruction operates under mask and vector length. The exceptions are precise if either `.exact` is specified, or if $\mathtt{vc}_{\mathtt{vmode}}$ specifies exact exceptions.
261	Sync	$\mathtt{vsync}\left\{\begin{array}{c}\mathtt{.sav}\\\mathtt{.vas}\\\mathtt{.vav}\\\mathtt{.vp}\end{array}\right\}\left\{\begin{array}{c}\phi\\\mathtt{.raw}\\\mathtt{.war}\\\mathtt{.waw}\end{array}\right\}$	Enforces scalar-after-vector, vector-after-scalar, vector-after-vector, or intra-vp memory ordering. If none of raw/war/waw are specified, then all are in effect. All relevant memory references preceeding the sync must appear to execute before all relevant memory references following the sync. Order is defined as program order.
213	Register Sync	$\mathtt{vrsync}\left\{\begin{array}{c}\mathtt{.sav}\\\mathtt{.vav}\\\mathtt{.vp}\end{array}\right\}\left\{\begin{array}{c}\phi\\\mathtt{.raw}\\\mathtt{.war}\\\mathtt{.waw}\end{array}\right\}\ \mathtt{vr}_{\mathtt{dest}}$	Enforces scalar-after-vector, vector-after-scalar, vector-after-vector, or intra-vp memory ordering. If none of raw/war/waw are specified, then all are in effect. The most recent memory reference to $\mathtt{vr}_{\mathtt{dest}}$ preceeding the sync must appear to execute before all relevant memory references following the sync. Order is defined as program order.

Figure 2.20: Miscellaneous Instructions

Page	*Operation*	*Assembly*	*Summary*
262	TLB Probe	`vtlbp`	This instruction is not yet implemented.
263	TLB Read	`vtlbr`	This instruction is not yet implemented.
264	TLB Write Indexed	`vtlbwi`	This instruction is not yet implemented.
265	TLB Write Random	`vtlbwr`	This instruction is not yet implemented.
131	Flush Vector Unit	`vflush`	The vector unit must be frozen. All in-flight instructions are flushed. The register files are not touched.

Figure 2.21: Miscellaneous Kernel-Mode Instructions

Appendix A

Instruction Formats

The vector register format (VR-Type) specifies three registers and a 10-bit opcode in the coprocessor-2 opcode space. The 10-bit vector opcode is split into two pieces:

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Note that single-source VR-type instructions use field `src1` to specify the source register.

There are two instruction formats that are used by the coprocessor interface instructions to move data between the scalar and vector units:

	31–26	25–21	20–16	15–11	10–0
MXC-Type	cop2	op	gpr	vsr	0
	6	5	5	5	11

	31–26	25–21	20–16	15–10	9–0
CXC-Type	cop2	op	gpr	vcr	0
	6	5	5	6	10

There are several instruction formats used by the instructions that move data between the vector scalar and vector control register files:

	31–26	25	24–21	20–16	15–10	9–6	5–0
VMSTC-Type	cop2	1	op	src	dest	0	op
	6	1	4	5	6	4	6

	31–26	25	24–21	20–16	15–10	9–6	5–0
VMCTS-Type	cop2	1	op	dest	src	0	op
	6	1	4	5	6	4	6

Each class of vector memory access – unit-stride, constant-stride, and indexed – has its own instruction format:

VMEM-Type

31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
cop2	1	op	vr	base	inc	0	op
6	1	4	5	4	3	3	6

VMEMS-Type

31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
cop2	1	op	vr	base	inc	stride	op
6	1	4	5	4	3	3	6

VMEMX-Type

31–26	25	24–21	20–16	15–12	11–7	6	5–0
cop2	1	op	vr	base	offsets	0	op
6	1	4	5	4	5	1	6

NOTE: *The vsync instruction uses some special formats, to be defined soon...*

Appendix B

Instruction Definitions

CFC2 Control From Coprocessor 2

	31 26	25 21	20 16	15 10	9 0
CXC-Type	cop2	cf	dest	src	0
	6	5	5	6	10

Assembly

cfc2 $\mathtt{r}_{\mathtt{dest}}$, $\mathtt{vc}_{\mathtt{src}}$

Operation

```
if (!user_readable (VC[src])) { /* assume user mode for now */
  Raise vIUI;
}
x = pdinst->get_src_vc (src);
GPR[dest] = x;
```

Description

$\mathtt{vc}_{\mathtt{src}}$ is copied into register $\mathtt{r}_{\mathtt{dest}}$.

Exceptions

`vIUI` Illegal Use of Instruction Exception

Notes

None.

Control To Coprocessor 2 — CTC2

	31–26	25–21	20–16	15–10	9–0
CXC-Type	cop2	ct	src	dest	0
	6	5	5	6	10

Assembly

ctc2 $\mathtt{r}_{\mathtt{src}}$, $\mathtt{vc}_{\mathtt{dest}}$

Operation

```
if (!user_writable (VC[dest])) { /* assume user mode for now */
  Raise vIUI;
}
x = pdinst->get_src_gpr (src);
if (dest == VcVpw) {
  x &= 0x3; /* force vpw to a legal value */
  VC[vpw] = x;
  VC[mvl] = MVL[x];
  VC[logmvl] = LOGMVL[x];
} else {
  VC[dest] = x;
}
```

Description

$\mathtt{r}_{\mathtt{src}}$ is copied into register $\mathtt{vc}_{\mathtt{dest}}$. Writing $\mathtt{vc}_{\mathtt{vpw}}$ changes $\mathtt{vc}_{\mathtt{mvl}}$, $\mathtt{vc}_{\mathtt{logmvl}}$, and $\mathtt{vc}_{\mathtt{vsafevl}}$ as a side-effect.

Exceptions

`vInt`	Vector Integer Exception
`vFPE`	Vector Floating-Point Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

MTC2 Move To Coprocessor 2

	31–26	25–21	20–16	15–11	10–0
MXC-Type	cop2	mt	src	dest	0
	6	5	5	5	11

Assembly

mtc2 `r`$_{src}$, `vs`$_{dest}$

Operation

```
x = pdinst->get_src_gpr (src);
VS[dest] = x;
```

Description

`r`$_{src}$ is copied into register `vs`$_{dest}$.

Exceptions

None.

Notes

None.

Vector Integer Absolute Value VABS

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vabs[.1] $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
raise_F = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = abs (x);
    if (abs_overflow (x, z)) {
      f = true;
      if (!speculative && F_enabled) {
        raise_F = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_F][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_F][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_F][vp] = f;
}
if (raise_F) {
  Raise vAri;
}
```

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the absolute value of $\mathtt{vr}_{\mathtt{src}}$.

VABS(cont.) Vector Integer Absolute Value

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

This operation is only defined for signed operands.

Vector Floating-Point Absolute Value VABS.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vabs$\left\{\begin{matrix}\texttt{.s}\\ \texttt{.d}\end{matrix}\right\}$[.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = abs_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VABS.fmt(cont.) Vector Floating-Point Absolute Value

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = abs_d (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Description

Each unmasked VP places the floating-point absolute value of $\texttt{vr}_{\texttt{src}}$ into $\texttt{vr}_{\texttt{dest}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

VACOMMIT — Commit Speculative Arithmetic

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vacommit[`.exact`][`.1`]

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
F = S = I = U = O = Z = V = E = false;
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    F = F || (VF[vfe_F][vp] && F_enabled);
    S = S || (VF[vfe_S][vp] && S_enabled);
    I = I || (VF[vfe_I][vp] && I_enabled);
    U = U || (VF[vfe_U][vp] && U_enabled);
    O = O || (VF[vfe_O][vp] && O_enabled);
    Z = Z || (VF[vfe_Z][vp] && Z_enabled);
    V = V || (VF[vfe_V][vp] && V_enabled);
    E = E || VF[vfe_E][vp];
  }
}
if (F || S || I || U || O || Z || V || E) {
  Raise vAri;
}
```

Description

Any pending vector arithmetic exceptions generated by speculative operations are raised. This instruction operates under mask and vector length. The exceptions are precise if either `.exact` is specified, or if $\texttt{vc}_{\texttt{vmode}}$ specifies exact exceptions.

Exceptions

None.

Notes

None.

Signed Vector Integer Add — VADD

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vadd}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, } \texttt{vr}_{\texttt{src1}}\texttt{, } \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, } \texttt{vs}_{\texttt{src1}}\texttt{, } \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
raise_F = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x + y;
    if (add_overflow (x, y, z)) {
      f = true;
      if (!speculative && F_enabled) {
        raise_F = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_F][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_F][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_F][vp] = f;
}
if (raise_F) {
  Raise vAri;
}
```

VADD(cont.) Signed Vector Integer Add

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the signed integer sum of $\mathtt{vs}_{\mathtt{src1}}$/$\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Unsigned Vector Integer Add VADD.U

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vadd.u}\left\{\begin{array}{l}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vr}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vs}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\end{array}\right.$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x + y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the unsigned integer sum of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

The only difference between `vadd` and `vadd.u` is that `vadd` can overflow, while `vadd.u` cannot.

VADD.fmt Vector Floating-Point Add

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vadd}\begin{Bmatrix}\texttt{.s}\\\texttt{.d}\end{Bmatrix}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\\texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = add_s (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VADD.fmt(cont.) Vector Floating-Point Add

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = add_d (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Description

Each unmasked VP places the floating-point sum of $\mathtt{vr}_{\mathtt{src1}}/\mathtt{vs}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$ into $\mathtt{vr}_{\mathtt{dest}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

VAND — Vector And

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vand}\begin{cases}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x & y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each VP writes into $\texttt{vr}_{\texttt{dest}}$ the bit-wise logical *and* of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Vector Floating-Point Ceiling VCEIL

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vceil$\left\{\begin{array}{l}\texttt{.w}\\ \texttt{.l}\end{array}\right\}\left\{\begin{array}{l}\texttt{.s}\\ \texttt{.d}\end{array}\right\}$[.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation (Single to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = ceil_s_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCEIL(cont.) Vector Floating-Point Ceiling

Operation (Double to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = ceil_d_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Single to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = ceil_s_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCEIL(cont.) Vector Floating-Point Ceiling

Operation (Double to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = ceil_d_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the result of converting $\mathtt{vr}_{\mathtt{src}}$ from a floating-point format to a signed integer format, using the *ceiling* rounding mode.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

VCIOTA Vector Continuous Iota

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

vciota $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vf}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
count = 0;
for (vp = 0; vp < vl; vp++) {
  VR[dest][vp] = count;
  if (VF[src][vp]) {
    count++;
  }
}
for (vp = vl; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

The continuous index vector of $\mathtt{vf}_{\mathtt{src1}}$ is placed in register $\mathtt{vr}_{\mathtt{dest}}$. The continuous index vector's value at vp_i is equal to the population count of flag values for $\mathrm{vp}_0 \cdots \mathrm{vp}_{i-1}$. This instruction is not masked.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

The continuous and compressed index vectors of a flag register of all 1's are the same.

Signed Vector Integer Compare VCMP

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vcmp}\left\{\begin{array}{l}\texttt{.eq}\\ \texttt{.neq}\end{array}\right\}\left\{\begin{array}{l}\texttt{.vv[.1]}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{array}\right.$$

$$\text{vcmp}\left\{\begin{array}{l}\texttt{.lt}\\ \texttt{.le}\end{array}\right\}\left\{\begin{array}{l}\texttt{.vv[.1]}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs[.1]}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vs}_{\texttt{src2}}\end{array}\right.$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    b = x OP y;
  } else {
    b = VF[dest][vp];
  }
  VF[dest][vp] = b;
}
```

Description

Each unmasked VP writes into $\texttt{vf}_{\texttt{dest}}$ the signed integer comparison of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector. A true predicate yields 1; a false predicate yields 0.

VCMP(cont.) Signed Vector Integer Compare

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

Commutative operations are not provided in `.vs` form.

Unsigned Vector Integer Compare VCMP.U

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vcmp.u}\left\{\begin{array}{l}\texttt{.lt}\\ \texttt{.le}\end{array}\right\}\left\{\begin{array}{l}\texttt{.vv}[\texttt{.1}]\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv}[\texttt{.1}]\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs}[\texttt{.1}]\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vs}_{\texttt{src2}}\end{array}\right.$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    b = x OP y;
  } else {
    b = VF[dest][vp];
  }
  VF[dest][vp] = b;
}
```

Description

Each unmasked VP writes into $\texttt{vf}_{\texttt{dest}}$ the unsigned integer comparison of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vs}_{\texttt{src2}}$/$\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector. A true predicate yields 1; a false predicate yields 0.

VCMP.U(cont.) Unsigned Vector Integer Compare

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

Commutative operations are not provided in `.vs` form. The `.eq` and `.neq` are provided only as signed comparisons, since those operations are independent of signedness.

Vector Floating-Point Compare VCMP.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vcmp}\left\{\begin{array}{c}\texttt{.s}\\\texttt{.d}\end{array}\right\}\left\{\begin{array}{c}\texttt{.f}\\\texttt{.sf}\\\texttt{.un}\\\texttt{.ngle}\\\texttt{.eq}\\\texttt{.seq}\\\texttt{.ueq}\\\texttt{.ngl}\end{array}\right\}\left\{\begin{array}{l}\texttt{.vv}[\texttt{.1}]\ \ \mathbf{vf}_{\texttt{dest}}\texttt{,}\ \ \mathbf{vr}_{\texttt{src1}}\texttt{,}\ \ \mathbf{vr}_{\texttt{src2}}\\\texttt{.sv}[\texttt{.1}]\ \ \mathbf{vf}_{\texttt{dest}}\texttt{,}\ \ \mathbf{vs}_{\texttt{src1}}\texttt{,}\ \ \mathbf{vr}_{\texttt{src2}}\end{array}\right.$$

$$\text{vcmp}\left\{\begin{array}{c}\texttt{.s}\\\texttt{.d}\end{array}\right\}\left\{\begin{array}{c}\texttt{.olt}\\\texttt{.lt}\\\texttt{.ult}\\\texttt{.nge}\\\texttt{.ole}\\\texttt{.le}\\\texttt{.ule}\\\texttt{.ngt}\end{array}\right\}\left\{\begin{array}{l}\texttt{.vv}[\texttt{.1}]\ \ \mathbf{vf}_{\texttt{dest}}\texttt{,}\ \ \mathbf{vr}_{\texttt{src1}}\texttt{,}\ \ \mathbf{vr}_{\texttt{src2}}\\\texttt{.sv}[\texttt{.1}]\ \ \mathbf{vf}_{\texttt{dest}}\texttt{,}\ \ \mathbf{vs}_{\texttt{src1}}\texttt{,}\ \ \mathbf{vr}_{\texttt{src2}}\end{array}\right.$$

VCMP.fmt(cont.) Vector Floating-Point Compare

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    b = cmp_s_<op> (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        b = VF[dest][vp];
      }
    }
  } else {
    b = VF[dest][vp];
  }
  VF[dest][vp] = b;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    b = cmp_d_<op> (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        b = VF[dest][vp];
      }
    }
  } else {
    b = VF[dest][vp];
  }
  VF[dest][vp] = b;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCMP.fmt(cont.) Vector Floating-Point Compare

Description

Each unmasked VP writes into $\mathtt{vf}_{\mathtt{dest}}$ the floating-point comparison of $\mathtt{vs}_{\mathtt{src1}}$/$\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vs}_{\mathtt{src2}}$/$\mathtt{vr}_{\mathtt{src2}}$, where at least one source is a vector. A true predicate yields 1; a false predicate yields 0.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

Commutative operations are not provided in `.vs` form.

Vector Compress VCOMPRESS

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vcompress[.1] $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIUI;
}
if (src == dest) {
  Raise vIUI;
}
count = 0;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    VR[dest][count++] = x;
  }
}
for (vp = count; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

All unmasked elements of $\mathtt{vr}_{\mathtt{src}}$ are catenated to form a vector whose length is the population count of the mask (subject to vector length). The result is placed at the front of $\mathtt{vr}_{\mathtt{dest}}$, leaving trailing elements untouched. $\mathtt{vr}_{\mathtt{dest}}$ must be different than $\mathtt{vr}_{\mathtt{src}}$.

Exceptions

`vIUI`	Illegal Use of Instruction Exception
`vIVL`	Invalid Vector Length Exception

Notes

Compress is the opposite of expand. The vector length applies to $\mathtt{vr}_{\mathtt{src}}$ and the mask.

VCVT — Vector Convert

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vcvt.s.}\begin{Bmatrix}\texttt{.d}\\\texttt{.w}\\\texttt{.l}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{src}}$$

$$\text{vcvt.d.}\begin{Bmatrix}\texttt{.s}\\\texttt{.l}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{src}}$$

$$\text{vcvt.w.}\begin{Bmatrix}\texttt{.s}\\\texttt{.d}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{src}}$$

$$\text{vcvt.l.}\begin{Bmatrix}\texttt{.s}\\\texttt{.d}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{src}}$$

Operation (Double to Single)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_d_to_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCVT(cont.) Vector Convert

Operation (Word to Single)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_w_to_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Long to Single)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_l_to_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCVT(cont.) Vector Convert

Operation (Single to Double)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_s_to_d (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Long to Double)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_l_to_d (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCVT(cont.) Vector Convert

Operation (Single to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_s_to_w (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_d_to_w (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCVT(cont.) Vector Convert

Operation (Single to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_s_to_l (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = cvt_d_to_l (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VCVT(cont.) Vector Convert

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the result of converting $\mathtt{vr}_{\mathtt{src}}$ from one integer or floating-point format to another.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception

Notes

The difference between vcvt and the other floating-point conversion instructions (vtrunc, vfloor, vround, and vceil) is that vcvt implicitly uses the default rounding mode (specified in vector control register $\mathtt{vc}_{\mathtt{vmode}}$), where the other instructions use explicit rounding modes.

Signed Vector Integer Divide VDIV

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vdiv}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    if (y == 0) {
      z = VR[dest][vp];
    } else {
      z = x / y;
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the signed integer quotient of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector.

VDIV(cont.) Signed Vector Integer Divide

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

The result is undefined if the divisor is zero.

Unsigned Vector Integer Divide VDIVU

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vdiv.u}\begin{cases}\texttt{.vv}[.1]\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vr}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv}[.1]\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vs}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs}[.1]\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vr}_{\texttt{src1}},\ \ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    if (y == 0) {
      z = VR[dest][vp];
    } else {
      z = x / y;
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the unsigned integer quotient of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector.

VDIVU(cont.) Unsigned Vector Integer Divide

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

The result is undefined if the divisor is zero.

Vector Floating-Point Divide VDIV.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vdiv}\left\{\begin{matrix}\texttt{.s}\\ \texttt{.d}\end{matrix}\right\}\left\{\begin{matrix}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}, & \texttt{vr}_{\texttt{src1}}, & \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}, & \texttt{vs}_{\texttt{src1}}, & \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs[.1]} & \texttt{vr}_{\texttt{dest}}, & \texttt{vr}_{\texttt{src1}}, & \texttt{vs}_{\texttt{src2}}\end{matrix}\right.$$

VDIV.fmt(cont.) Vector Floating-Point Divide

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = div_s (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = div_d (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VDIV.fmt(cont.) Vector Floating-Point Divide

Description

Each unmasked VP places the floating-point quotient of $\mathtt{vr}_{\mathtt{src1}}$/$\mathtt{vs}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$/$\mathtt{vf}_{\mathtt{src2}}$ into $\mathtt{vr}_{\mathtt{dest}}$, where at least one souce is a vector.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception

Notes

None.

Vector Expand VEXPAND

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	0	src	dest	op
6	1	4	5	5	5	6

Assembly

vexpand[.1] $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIUI;
}
if (src == dest) {
  Raise vIUI;
}
count = 0;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][count++];
  } else {
    x = VR[dest][vp];
  }
  VR[dest][vp] = x;
}
```

Description

The first n elements of $\mathtt{vr}_{\mathtt{src}}$ are written into the unmasked positions of $\mathtt{vr}_{\mathtt{dest}}$, where n is the population count of the mask (subject to vector length). Masked positions in $\mathtt{vr}_{\mathtt{dest}}$ are not touched. $\mathtt{vr}_{\mathtt{dest}}$ must be different than $\mathtt{vr}_{\mathtt{src}}$.

Exceptions

vIUI	Illegal Use of Instruction Exception
vIVL	Invalid Vector Length Exception

Notes

Expand is the opposite of compress. The vector length applies to $\mathtt{vr}_{\mathtt{dest}}$ and the mask.

VEXT.SV Signed Scalar-Vector Extract

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vext.sv $\mathtt{vs}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src2}}$

Operation

```
vindex &= mvl - 1; /* force vindex into valid range */
x = VR[src][vindex];
VS[dest] = x;
```

Description

Element $\mathtt{vc}_{\mathtt{vindex}}$ of $\mathtt{vr}_{\mathtt{src}}$ is written into $\mathtt{vs}_{\mathtt{dest}}$. The lower $\mathtt{vc}_{\mathtt{logmvl}}$ bits of $\mathtt{vc}_{\mathtt{vindex}}$ are used. The value is sign-extended. This instruction is not masked and does not use vector length.

Exceptions

None.

Notes

None.

Unsigned Scalar-Vector Extract — VEXT.U.SV

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vext.u.sv $\texttt{vs}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation

```
vindex &= mvl - 1; /* force vindex into valid range */
x = VR[src][vindex];
VS[dest] = x;
```

Description

Element $\texttt{vc}_{\texttt{vindex}}$ of $\texttt{vr}_{\texttt{src}}$ is written into $\texttt{vs}_{\texttt{dest}}$. The lower $\texttt{vc}_{\texttt{logmvl}}$ bits of $\texttt{vc}_{\texttt{vindex}}$ are used. The value is zero-extended. This instruction is not masked and does not use vector length.

Exceptions

None.

Notes

Floating-point extraction should be done with this instruction.

VEXT.VV — Vector-Vector Extract

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vext.vv $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
if (src == dest) {
  Raise vIUI;
}
vindex &= mvl - 1; /* force vindex into valid range */
for (vp = 0; vp < vl && vp + vindex < mvl; vp++) {
  x = VR[src][vp + vindex];
  VR[dest][vp] = x;
}
for (; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

A portion of $\mathtt{vr}_{\mathtt{src}}$ is extracted into the front of $\mathtt{vr}_{\mathtt{dest}}$. $\mathtt{vr}_{\mathtt{dest}}$ must be different than $\mathtt{vr}_{\mathtt{src}}$. Trailing entries of $\mathtt{vr}_{\mathtt{dest}}$ are not touched. The lower $\mathtt{vc}_{\mathtt{logmvl}}$ bits of vector control register $\mathtt{vc}_{\mathtt{vindex}}$ specifies the starting position in $\mathtt{vr}_{\mathtt{src}}$. The vector length specifies the number of elements to transfer. This instruction is not masked.

Exceptions

`vIUI`	Illegal Use of Instruction Exception
`vIVL`	Invalid Vector Length Exception

Notes

If vindex + vl ¿ mvl, then the only elements copied are those whose source vp number is legal. Vector-vector extract is the opposite of vector-vector insert.

Vector Extract Half VEXTHALF

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vexthalf vr_{dest}, vr_{src}

Operation

```
if (vl > mvl || vl == 1 || !is_pow2 (vl)) {
  Raise vIVL;
}
if (src == dest) {
  Raise vIUI;
}
for (vp = 0; vp < vl / 2; vp++) {
  x = VR[src][vl / 2 + vp];
  VR[dest][vp] = x;
}
for (vp = vl / 2; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

The top half of vr_{src} is extracted into the front of vr_{dest}. vr_{dest} must be different than vr_{src}. Trailing entries of vr_{dest} are not touched. The vector length specifies the length of vr_{src}. This instruction is not masked. The vector length must be a power of two, and must be at least two.

Exceptions

`vIUI`	Illegal Use of Instruction Exception
`vIVL`	Invalid Vector Length Exception

Notes

This instruction is a special-case of the `vext.vv` instruction. This instruction may be used to implement reductions efficiently.

VHALF Vector Half

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vhalf $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation

```
if (vl > mvl || vl == 1 || !is_pow2 (vl)) {
  Raise vIVL;
}
if (src == dest) {
  Raise vIUI;
}
for (vp = 0; vp < vl / 2; vp++) {
  x = VR[src][vl / 2 + vp];
  VR[dest][vp] = x;
}
for (vp = vl / 2; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
VC[vc_vl] = vl / 2;
```

Description

The top half of $\texttt{vr}_{\texttt{src}}$ is extracted into the front of $\texttt{vr}_{\texttt{dest}}$. $\texttt{vr}_{\texttt{dest}}$ must be different than $\texttt{vr}_{\texttt{src}}$. Trailing entries of $\texttt{vr}_{\texttt{dest}}$ are not touched. The vector length specifies the length of $\texttt{vr}_{\texttt{src}}$. This instruction is not masked. The vector length must be a power of two, and must be at least two. After execution, the vector length register is divided by two.

Exceptions

vIUI	Illegal Use of Instruction Exception
vIVL	Invalid Vector Length Exception

Notes

This instruction is identical to the `vexthalf` instruction, except that this instruction also halves the vector length register. This instruction may be used to implement reductions efficiently.

VHALFDN Vector Half Down

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	dest	op
	6	1	4	5	5	5	6

Assembly

vhalfdn $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation

```
if (vl > mvl || vl == 1 || !is_pow2 (vl)) {
  Raise vIVL;
}
if (src == dest) {
  Raise vIUI;
}
L = 1 << vindex;
if (L >= vl) {
  vl = 0; /* operation not defined */
}
for (vp = 0; vp < vl; vp += 2 * L) {
  for (i = 0; i < L; i++) {
    x = VR[src][vp + L + i];
    VR[dest][vp + i] = x;
    x = VR[dest][vp + L + i];
    VR[dest][vp + L + i] = x;
  }
}
for (vp = vl; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

This instruction does one half of one step of a butterfly permutation. Paired with `vhalfdn`, it can be used to perform an in-register butterfly permutation. $\mathtt{vr}_{\mathtt{dest}}$ must be different than $\mathtt{vr}_{\mathtt{src}}$. $\mathtt{vc}_{\mathtt{vindex}}$ specifies the log of both the length and the separation of the sub-vectors that are moved from $\mathtt{vr}_{\mathtt{src}}$ to $\mathtt{vr}_{\mathtt{dest}}$. The sub-vectors are shifted down within each $2 * 2^{\mathtt{vc}_{\mathtt{vindex}}}$ block by $2^{\mathtt{vc}_{\mathtt{vindex}}}$ elements. This writes half of the elements within each block in $\mathtt{vr}_{\mathtt{dest}}$; the remaining elements are not touched. This instruction is not masked. $\mathtt{vc}_{\mathtt{vl}}$ must be at least 2 and must be a power of 2. $2^{\mathtt{vc}_{\mathtt{vindex}}}$ must be less than $\mathtt{vc}_{\mathtt{vl}}$.

Exceptions

`vIUI`	Illegal Use of Instruction Exception
`vIVL`	Invalid Vector Length Exception

Notes

This instruction may be used to implement multiple reductions efficiently.

VHALFUP Vector Half Up

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	dest	op
	6	1	4	5	5	5	6

Assembly

vhalfup $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src}}$

Operation

```
if (vl > mvl || vl == 1 || !is_pow2 (vl)) {
  Raise vIVL;
}
if (src == dest) {
  Raise vIUI;
}
L = 1 << vindex;
if (L >= vl) {
  vl = 0; /* operation not defined */
}
for (vp = 0; vp < vl; vp += 2 * L) {
  for (i = 0; i < L; i++) {
    x = VR[src][vp + i];
    VR[dest][vp + L + i] = x;
    x = VR[dest][vp + i];
    VR[dest][vp + i] = x;
  }
}
for (vp = vl; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

This instruction does one half of one step of a butterfly permutation. Paired with `vhalfdn`, it can be used to perform an in-register butterfly permutation. $\mathtt{vr}_{\mathtt{dest}}$ must be different than $\mathtt{vr}_{\mathtt{src}}$. $\mathtt{vc}_{\mathtt{vindex}}$ specifies the log of both the length and the separation of the sub-vectors that are moved from $\mathtt{vr}_{\mathtt{src}}$ to $\mathtt{vr}_{\mathtt{dest}}$. The sub-vectors are shifted up within each $2 * 2^{\mathtt{vc}_{\mathtt{vindex}}}$ block by $2^{\mathtt{vc}_{\mathtt{vindex}}}$ elements. This writes half of the elements within each block in $\mathtt{vr}_{\mathtt{dest}}$; the remaining elements are not touched. This instruction is not masked. $\mathtt{vc}_{\mathtt{vl}}$ must be at least 2 and must be a power of 2. $2^{\mathtt{vc}_{\mathtt{vindex}}}$ must be less than $\mathtt{vc}_{\mathtt{vl}}$.

Exceptions

`vIUI`	Illegal Use of Instruction Exception
`vIVL`	Invalid Vector Length Exception

Notes

If $\mathtt{vc}_{\mathtt{vindex}}$ is $\mathtt{vc}_{\mathtt{vl}}/2$, this instruction is identical to the `vexthalf` instruction. This instruction may be used to implement multiple reductions efficiently.

VFAND — Vector Flag And

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vfand}\left\{\begin{array}{l}\texttt{.vv}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vf}_{\texttt{src1}}\texttt{,}\ \ \texttt{vf}_{\texttt{src2}}\\ \texttt{.sv}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vf}_{\texttt{src2}}\end{array}\right.$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl) {
    if (vv) {
      a = VF[src1][vp];
      b = VF[src2][vp];
    } else {
      a = (VS[src1] != 0);
      b = VF[src2][vp];
    }
    c = a && b;
  } else {
    c = VF[dest][vp];
  }
  VF[dest][vp] = c;
}
```

Description

Each VP writes into $\texttt{vf}_{\texttt{dest}}$ the logical *and* of $\texttt{vs}_{\texttt{src1}}/\texttt{vf}_{\texttt{src1}}$ and $\texttt{vf}_{\texttt{src2}}$. This instruction is not masked.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

A scalar source used as a boolean operand is false when zero and true when non-zero.

Vector Flag Clear VFCLR

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	dest	op
	6	1	4	5	5	5	6

Assembly

vfclr $\texttt{vf}_{\texttt{dest}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl) {
    c = false;
  } else {
    c = VF[dest][vp];
  }
  VF[dest][vp] = c;
}
```

Description

Each VP writes zero into $\texttt{vf}_{\texttt{dest}}$. This instruction is not masked.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

VFCLR8 Vector Flag Clear 8

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	0	0	dest	op
	6	1	4	5	5	5	6

Assembly

vfclr8 $\mathtt{vf}_{\mathtt{dest}}$

Operation

```
assert ((dest % 8) == 0);
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  for (i = 0; i < 8; i++) {
    VF[dest + i][vp] = false;
  }
}
for (vp = vl; vp < mvl; vp++) {
  for (i = 0; i < 8; i++) {
    c = VF[dest + i][vp];
    VF[dest + i][vp] = c;
  }
}
```

Description

Each VP writes zero into $\mathtt{vf}_{\mathtt{dest}} \cdots \mathtt{vf}_{\mathtt{dest}+7}$. This instruction is not masked, but is subject to vector length. The flag register specifier must be a multiple of 8.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

Vector Flag Find First One VFFF1

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfff1 $\texttt{vs}_{\texttt{dest}}$, $\texttt{vf}_{\texttt{src}}$

Operation

```
count = 0;
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[src][vp]) {
    break;
  }
}
VS[dest] = vp;
```

Description

The location of the first set bit of $\texttt{vf}_{\texttt{src}}$ is placed in $\texttt{vs}_{\texttt{dest}}$. This instruction is not masked. If there is no set bit in $\texttt{vf}_{\texttt{src}}$, then the vector length is placed in $\texttt{vs}_{\texttt{dest}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

VFFL1 Vector Flag Find Last One

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	0	src	dest	op
6	1	4	5	5	5	6

Assembly

vffl1 $\texttt{vs}_{\texttt{dest}}$, $\texttt{vf}_{\texttt{src}}$

Operation

```
last = vl;
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[src][vp]) {
    last = vp;
  }
}
VS[dest] = last;
```

Description

The location of the last set bit of $\texttt{vf}_{\texttt{src}}$ is placed in $\texttt{vs}_{\texttt{dest}}$. This instruction is not masked. If there is no set bit in $\texttt{vf}_{\texttt{src}}$, then the vector length is placed in $\texttt{vs}_{\texttt{dest}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Scalar-Vector Insert VFINS

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfins $\mathtt{vf}_{\mathtt{dest}}$, $\mathtt{vs}_{\mathtt{src}}$

Operation

```
vindex &= mvl - 1; /* force vindex into valid range */
for (vp = 0; vp < vindex && vp < mvl; vp++) {
  b = VF[dest][vp];
  VF[dest][vp] = b;
}
b = (VS[src] != 0);
VF[dest][vindex] = b;
for (vp = vindex + 1; vp < mvl; vp++) {
  b = VF[dest][vp];
  VF[dest][vp] = b;
}
```

Description

The boolean value of $\mathtt{vs}_{\mathtt{src}}$ is written into $\mathtt{vf}_{\mathtt{dest}}$ at position $\mathtt{vc}_{\mathtt{vindex}}$. The lower $\mathtt{vc}_{\mathtt{logmvl}}$ bits of $\mathtt{vc}_{\mathtt{vindex}}$ are used. This instruction is not masked and does not use vector length.

Exceptions

None.

Notes

A scalar source used as a boolean operand is false when zero and true when non-zero.

VFLD Vector Flag Load

VMEM-Type

31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
cop2	1	op	dest	base	inc	0	op
6	1	4	5	4	3	3	6

Assembly

vfld $\texttt{vf}_{\texttt{dest}}$[, `vbase`[, `vinc`]]

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
/* load 16-bit chunks; not 8-bit chunks! */
for (vp = 0; vp < vl; vp += 16) {
  va = base + vp / 8;
  if ((va & 0x1) != 0) { /* Address must be 16-bit aligned */
    fault[vp] = vAdEL;
  } else {
    fault[vp] = translate (READ, 2, va, &pa[vp]);
  }
  if (!speculative && fault[vp] != Exc_none) {
    Raise fault[vp];
  }
}
for (vp = 0; vp < mvl; vp++) {
  if (speculative) {
    a = VF[vfe_L][vp];
    VF[vfe_L][vp] = a;
  }
  b = VF[dest][vp];
  VF[dest][vp] = b;
}
for (vp = 0; vp < vl; vp += 16) {
  if (fault[vp] != Exc_none) {
    assert (speculative);
    for (i = 0; i < 16; i++) {
      VF[vfe_L][vp + i] = true;
    }
  } else {
    x = unsigned_load (pa[vp], 2);
    for (i = 0; i < 16 && vp + i < vl; i++) {
      loc = i < 8 ? i + 8 : i - 8; /* bytes are little-endian */
      b = (x >> loc) & 0x1 ;
      VF[dest][vp + i] = b;
    }
  }
}
VC[vc_base] = base + inc;
```

Description

The VPs perform a contiguous vector flag load into $\texttt{vf}_{\texttt{dest}}$. $\texttt{vf}_{\texttt{dest}}$ should be different than the speculative load fault flag register if the instruction is speculative. The base address is given by `vbase` (default is $\texttt{vbase}_0$), and must be 16-bit aligned. The bytes are loaded in little-endian order. The signed increment in `vinc` (default is $\texttt{vinc}_0$) is added to `vbase` as a side-effect. This instruction is not masked.

VFLD(cont.) Vector Flag Load

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdEL`	Vector Address Error (Load)
`vMod`	Vector TLB Modification Exception
`vTLBL`	Vector TLB Exception (Load)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

Exactly VL bits are loaded into the destination register. All addresses are checked and translated before any values are loaded. A load fault causes an exception only if the instruction is non-speculative. A speculative faulting load does not load any value.

Vector Floating-Point Floor — VFLOOR

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vfloor}\left\{\begin{matrix}\texttt{.w}\\\texttt{.l}\end{matrix}\right\}\left\{\begin{matrix}\texttt{.s}\\\texttt{.d}\end{matrix}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{src}}$$

VFLOOR(cont.) Vector Floating-Point Floor

Operation (Single to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = floor_s_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = floor_d_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VFLOOR(cont.) Vector Floating-Point Floor

Operation (Single to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = floor_s_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = floor_d_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VFLOOR(cont.) Vector Floating-Point Floor

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the result of converting $\mathtt{vr}_{\mathtt{src}}$ from a floating-point format to a signed integer format, using the *floor* rounding mode.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception

Notes

None.

Vector Flag Move From 8 — VFMF8

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfmf8 $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vf}_{\mathtt{src}}$

Operation

```
assert ((src % 8) == 0);
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  x = 0;
  for (i = 0; i < 8; i++) {
    if (VF[src+i][vp]) {
      x |= 1 << i;
    }
  }
  VR[dest][vp] = x;
}
for (vp = vl; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

Registers $\mathtt{vf}_{\mathtt{src}} \cdots \mathtt{vf}_{\mathtt{src}+7}$ are copied into $\mathtt{vr}_{\mathtt{dest}}$. This instruction is not masked, but is subject to vector length. The flag register specifier must be a multiple of 8.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

The mapping of flag registers into $\mathtt{vr}_{\mathtt{dest}}$ is not defined, but must be consistent with the `vfmt8` instruction. The result of this instruction is only meaningful to the `vfmt8` instruction.

VFMT8 Vector Flag Move To 8

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfmt8 $\texttt{vf}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation

```
assert ((dest % 8) == 0);
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  x = VR[src][vp];
  for (i = 0; i < 8; i++) {
    b = (x >> i) & 0x1;
    VF[dest + i][vp] = b;
  }
}
for (vp = vl; vp < mvl; vp++) {
  for (i = 0; i < 8; i++) {
    b = VF[dest + i][vp];
    VF[dest + i][vp] = b;
  }
}
```

Description

Registers $\texttt{vf}_{\texttt{dest}} \cdots \texttt{vf}_{\texttt{dest}+7}$ are set with the contents of $\texttt{vr}_{\texttt{src}}$. This instruction is not masked, but is subject to vector length. The flag register specifier must be a multiple of 8.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

The mapping of flag registers in $\texttt{vr}_{\texttt{src}}$ is not defined, but is consistent with the `vfmf8` instruction. Register $\texttt{vr}_{\texttt{src}}$ must contain the ressult of a previous `vfmf8` instruction.

Vector Flag Nor — VFNOR

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vfnor}\begin{cases}\texttt{.vv}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vf}_{\texttt{src1}}\texttt{,}\ \ \texttt{vf}_{\texttt{src2}} \\ \texttt{.sv}\ \ \texttt{vf}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vf}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl) {
    if (vv) {
      a = VF[src1][vp];
      b = VF[src2][vp];
    } else {
      a = (VS[src1] != 0);
      b = VF[src2][vp];
    }
    c = !(a || b);
  } else {
    c = VF[dest][vp];
  }
  VF[dest][vp] = c;
}
```

Description

Each VP writes into $\texttt{vf}_{\texttt{dest}}$ the logical *nor* of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vf}_{\texttt{src1}}$ and $\texttt{vf}_{\texttt{src2}}$. This instruction is not masked.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

A scalar source used as a boolean operand is false when zero and true when non-zero.

VFOR Vector Flag Or

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vfor}\begin{cases}\texttt{.vv}\ \ \texttt{vf}_{\texttt{dest}},\ \ \texttt{vf}_{\texttt{src1}},\ \ \texttt{vf}_{\texttt{src2}}\\ \texttt{.sv}\ \ \texttt{vf}_{\texttt{dest}},\ \ \texttt{vs}_{\texttt{src1}},\ \ \texttt{vf}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl) {
    if (vv) {
      a = VF[src1][vp];
      b = VF[src2][vp];
    } else {
      a = (VS[src1] != 0);
      b = VF[src2][vp];
    }
    c = a || b;
  } else {
    c = VF[dest][vp];
  }
  VF[dest][vp] = c;
}
```

Description

Each VP writes into $\texttt{vf}_{\texttt{dest}}$ the logical *or* of $\texttt{vs}_{\texttt{src1}}/\texttt{vf}_{\texttt{src1}}$ and $\texttt{vf}_{\texttt{src2}}$. This instruction is not masked.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

A scalar source used as a boolean operand is false when zero and true when non-zero.

Vector Flag Or 8 VFOR8

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfor8 $\mathtt{vf}_{\mathtt{dest}}$, $\mathtt{vf}_{\mathtt{src}}$

Operation

```
assert ((src % 8) == 0);
assert ((dest % 8) == 0);
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  for (i = 0; i < 8; i++) {
    a = VF[dest + i][vp];
    b = VF[src + i][vp];
    VF[dest + i][vp] = a || b;
  }
}
for (vp = vl; vp < mvl; vp++) {
  for (i = 0; i < 8; i++) {
    c = VF[dest + i][vp];
    VF[dest + i][vp] = c;
  }
}
```

Description

Each VP performs an OR-write of $\mathtt{vf}_{\mathtt{src}} \cdots \mathtt{vf}_{\mathtt{src}+7}$ into $\mathtt{vf}_{\mathtt{dest}} \cdots \mathtt{vf}_{\mathtt{dest}+7}$. This instruction is not masked, but is subject to vector length. Each flag register specifier must be a multiple of 8.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

VFPOP — Vector Flag Population Count

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfpop $\texttt{vs}_{\texttt{dest}}$, $\texttt{vf}_{\texttt{src}}$

Operation

```
count = 0;
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[src][vp]) {
    count++;
  }
}
VS[dest] = count;
```

Description

The population count of $\texttt{vf}_{\texttt{src}}$ is placed in $\texttt{vs}_{\texttt{dest}}$. This instruction is not masked.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Flush Vector Unit VFLUSH

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vflush

Operation

```
fprintf (stderr, "vflush not yet implementedn");
```

Description

The vector unit must be frozen. All in-flight instructions are flushed. The register files are not touched.

Exceptions

None.

Notes

None.

VFSET — Vector Flag Set

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	dest	op
	6	1	4	5	5	5	6

Assembly

vfset $\mathtt{vf}_{\mathtt{dest}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl) {
    c = true;
  } else {
    c = VF[dest][vp];
  }
  VF[dest][vp] = c;
}
```

Description

Each VP writes one into $\mathtt{vf}_{\mathtt{dest}}$. This instruction is not masked, but is subject to vector length.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Vector Flag Set Before First One — VFSETBF

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfsetbf $\mathtt{vf}_{\mathtt{dest}}$, $\mathtt{vf}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[src][vp]) {
    break;
  }
  VF[dest][vp] = true;
}
for (; vp < vl; vp++) {
  VF[dest][vp] = false;
}
for (vp = vl; vp < mvl; vp++) {
  b = VF[dest][vp];
  VF[dest][vp] = b;
}
```

Description

Regiser $\mathtt{vf}_{\mathtt{dest}}$ is filled with ones up to and not including the first set bit in $\mathtt{vf}_{\mathtt{src}}$. Remaining positions in $\mathtt{vf}_{\mathtt{dest}}$ are cleared. If $\mathtt{vf}_{\mathtt{src}}$ contains no set bits, $\mathtt{vf}_{\mathtt{dest}}$ is set to all ones. This instruction is not masked.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VFSETIF Vector Flag Set Including First One

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfsetif $\mathtt{vf}_{\mathtt{dest}}$, $\mathtt{vf}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  b = VF[src][vp];
  VF[dest][vp] = true;
  if (b) {
    break;
  }
}
for (++vp; vp < vl; vp++) {
  VF[dest][vp] = false;
}
for (vp = vl; vp < mvl; vp++) {
  b = VF[dest][vp];
  VF[dest][vp] = b;
}
```

Description

Regiser $\mathtt{vf}_{\mathtt{dest}}$ is filled with ones up to and including the first set bit in $\mathtt{vf}_{\mathtt{src}}$. Remaining positions in $\mathtt{vf}_{\mathtt{dest}}$ are cleared. If $\mathtt{vf}_{\mathtt{src}}$ contains no set bits, $\mathtt{vf}_{\mathtt{dest}}$ is set to all ones. This instruction is not masked.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

Vector Flag Set Only First One — VFSETOF

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vfsetof $\texttt{vf}_{\texttt{dest}}$, $\texttt{vf}_{\texttt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[src][vp]) {
    break;
  }
  VF[dest][vp] = false;
}
if (vp < vl) {
  VF[dest][vp] = true;
}
for (++vp; vp < vl; vp++) {
  VF[dest][vp] = false;
}
for (vp = vl; vp < mvl; vp++) {
  b = VF[dest][vp];
  VF[dest][vp] = b;
}
```

Description

Regiser $\texttt{vf}_{\texttt{dest}}$ is filled with zeros except for the position of the first set bit in $\texttt{vf}_{\texttt{src}}$. If $\texttt{vf}_{\texttt{src}}$ contains no set bits, $\texttt{vf}_{\texttt{dest}}$ is set to all zeros. This instruction is not masked.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

VFST Vector Flag Store

	31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
VMEM-Type	cop2	1	op	src	base	inc	0	op
	6	1	4	5	4	3	3	6

Assembly

vfst $\mathtt{vf}_{\mathtt{src}}$[, vbase[, vinc]]

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < vl; vp += 16) {
  va = base + vp / 8;
  if ((va & 0x1) != 0) { /* Address must be 16-bit aligned */
    fault[vp] = vAdEL;
  } else {
    fault[vp] = translate (WRITE, 2, va, &pa[vp]);
  }
  if (fault[vp] != Exc_none) {
    Raise fault[vp];
  }
}
for (vp = 0; vp < vl; vp += 16) {
  addr = pa[vp];
  x = 0;
  for (i = 0; i < 16; i++) {
    loc = i < 8 ? i + 8 : i - 8; /* bytes are little-endian */
    b = vp + i > vl ? false : VF[src][vp + i];
    x |= b << loc;
  }
  store (addr, 2, x);
}
VC[vc_base] = base + inc;
```

Description

The VPs perform a contiguous vector flag store of $\mathtt{vf}_{\mathtt{src}}$. The base address is given by vbase (default is $\mathtt{vbase}_0$), and must be 16-bit aligned. A multiple of 16 bits is written, padding with zeros as necessary. The bytes are stored in little-endian order. The signed increment in vinc (default is $\mathtt{vinc}_0$) is added to vbase as a side-effect. This instruction is not masked.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vAdES`	Vector Address Error (Store)
`vMod`	Vector TLB Modification Exception
`vTLBS`	Vector TLB Exception (Store)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

All addresses are checked and translated before any values are stored.

VFXOR Vector Flag Xor

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vfxor}\begin{cases}\texttt{.vv vf}_{\texttt{dest}}\texttt{, vf}_{\texttt{src1}}\texttt{, vf}_{\texttt{src2}}\\ \texttt{.sv vf}_{\texttt{dest}}\texttt{, vs}_{\texttt{src1}}\texttt{, vf}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl) {
    if (vv) {
      a = VF[src1][vp];
      b = VF[src2][vp];
    } else {
      a = (VS[src1] != 0);
      b = VF[src2][vp];
    }
    c = a ^ b;
  } else {
    c = VF[dest][vp];
  }
  VF[dest][vp] = c;
}
```

Description

Each VP writes into $\texttt{vf}_{\texttt{dest}}$ the logical *xor* of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vf}_{\texttt{src1}}$ and $\texttt{vf}_{\texttt{src2}}$. This instruction is not masked.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

A scalar source used as a boolean operand is false when zero and true when non-zero.

Scalar-Vector Insert VINS.SV

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vins.sv $\texttt{vr}_{\texttt{dest}}$, $\texttt{vs}_{\texttt{src}}$

Operation

```
vindex &= mvl - 1; /* force vindex into valid range */
for (vp = 0; vp < vindex; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
x = VS[src];
VR[dest][vindex] = x;
for (vp = vindex + 1; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

The contents of $\texttt{vs}_{\texttt{src}}$ are written into $\texttt{vr}_{\texttt{dest}}$ at position $\texttt{vc}_{\texttt{vindex}}$. The lower $\texttt{vc}_{\texttt{logmvl}}$ bits of $\texttt{vc}_{\texttt{vindex}}$ are used. This instruction is not masked and does not use vector length.

Exceptions

None.

Notes

The scalar source is truncated to the virtual processor width.

VINS.VV — Vector-Vector Insert

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vins.vv $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src2}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
if (src == dest) {
  Raise vIUI;
}
vindex &= mvl - 1; /* force vindex into valid range */
for (vp = 0; vp < vindex; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
for (vp = vindex; vp < vindex + vl && vp < mvl; vp++) {
  x = VR[src][vp - vindex];
  VR[dest][vp] = x;
}
for (vp = vindex + vl; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

The leading portion of $\mathtt{vr}_{\mathtt{src}}$ is inserted into $\mathtt{vr}_{\mathtt{dest}}$. $\mathtt{vr}_{\mathtt{dest}}$ must be different than $\mathtt{vr}_{\mathtt{src}}$. Leading and trailing entries of $\mathtt{vr}_{\mathtt{dest}}$ are not touched. The lower $\mathtt{vc}_{\mathtt{logmvl}}$ bits of vector control register $\mathtt{vc}_{\mathtt{vindex}}$ specifies the starting position in $\mathtt{vr}_{\mathtt{dest}}$. The vector length specifies the number of elements to transfer. This instruction is not masked.

Exceptions

vIUI	Illegal Use of Instruction Exception
vIVL	Invalid Vector Length Exception

Notes

If vindex + vl ¿ mvl, then the only elements copied are those whose destination vp number is legal. Vector-vector insert is the opposite of vector-vector extract.

VIOTA Vector Iota

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

viota $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vf}_{\mathtt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
count = 0;
for (vp = 0; vp < vl; vp++) {
  if (VF[src][vp]) {
    VR[dest][count++] = vp;
  }
}
for (vp = count; vp < mvl; vp++) {
  x = VR[dest][vp];
  VR[dest][vp] = x;
}
```

Description

The list of VPs with bits set in $\mathtt{vf}_{\mathtt{src}}$ is placed in $\mathtt{vr}_{\mathtt{dest}}$. This is the compressed index vector of $\mathtt{vf}_{\mathtt{src}}$. This instruction is not masked.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Unit Stride Signed Vector Load — VLD

	31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
VMEM-Type	cop2	1	op	dest	base	inc	0	op
	6	1	4	5	4	3	3	6

Assembly

$$\mathrm{vld}\left\{\begin{matrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{matrix}\right\}[.1]\ \texttt{vr}_{\texttt{dest}}[, \texttt{vbase}[, \texttt{vinc}]]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + nbytes * vp;
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (READ, nbytes, va, &pa[vp]);
    }
    if (!speculative && fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (fault[vp] != Exc_none) {
      if (speculative) b = true;
      x = VR[dest][vp];
    } else {
      if (speculative) b = VF[vfe_L][vp];
      x = signed_load (pa[vp], nbytes);
    }
  } else {
    if (speculative) b = VF[vfe_L][vp];
    x = VR[dest][vp];
  }
  if (speculative) VF[vfe_L][vp] = b;
  VR[dest][vp] = x;
}
VC[vc_base] = base + inc;
```

Unit Stride Signed Vector Load VLD(cont.)

Description

The VPs perform a contiguous vector load into $\texttt{vr}_{\texttt{dest}}$. The base address is given by `vbase` (default is $\texttt{vbase}_0$), and must be aligned to the width of the data in memory. The signed increment in `vinc` (default is $\texttt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The loaded value is sign-extended to the virtual processor width.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdEL`	Vector Address Error (Load)
`vMod`	Vector TLB Modification Exception
`vTLBL`	Vector TLB Exception (Load)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

All addresses are checked and translated before any values are loaded. A load fault causes an exception only if the instruction is non-speculative. A speculative faulting load does not load any value. The only difference between `vld` and `vld.u` is that `vld` sign-extends loaded data, and `vld.u` zero-extends loaded data.

VLD.U Unit Stride Unsigned Vector Load

VMEM-Type

31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
cop2	1	op	dest	base	inc	0	op
6	1	4	5	4	3	3	6

Assembly

$$\text{vld.u}\left\{\begin{matrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\end{matrix}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}}[,\texttt{vbase}[,\texttt{vinc}]]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + nbytes * vp;
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (READ, nbytes, va, &pa[vp]);
    }
    if (!speculative && fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (fault[vp] != Exc_none) {
      if (speculative) b = true;
      x = VR[dest][vp];
    } else {
      if (speculative) b = VF[vfe_L][vp];
      x = unsigned_load (pa[vp], nbytes);
    }
  } else {
    if (speculative) b = VF[vfe_L][vp];
    x = VR[dest][vp];
  }
  if (speculative) VF[vfe_L][vp] = b;
  VR[dest][vp] = x;
}
VC[vc_base] = base + inc;
```

Description

Operation is identical to `vld`, except that loaded values are zero-extended to the virtual processor width instead of sign-extended.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdEL`	Vector Address Error (Load)
`vMod`	Vector TLB Modification Exception
`vTLBL`	Vector TLB Exception (Load)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

`vld.u.l` is not provided, as it would be functionally identical to `vld.l`.

VLDS — Variable Stride Signed Vector Load

	31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
VMEMS-Type	cop2	1	op	dest	base	inc	stride	op
	6	1	4	5	4	3	3	6

Assembly

$$\text{vlds}\begin{Bmatrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}}[,\texttt{vbase}[,\texttt{vstride}[,\texttt{vinc}]]]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + nbytes * stride * vp;
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (READ, nbytes, va, &pa[vp]);
    }
    if (!speculative && fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (fault[vp] != Exc_none) {
      if (speculative) b = true;
      x = VR[dest][vp];
    } else {
      if (speculative) b = VF[vfe_L][vp];
      x = signed_load (pa[vp], nbytes);
    }
  } else {
    if (speculative) b = VF[vfe_L][vp];
    x = VR[dest][vp];
  }
  if (speculative) VF[vfe_L][vp] = b;
  VR[dest][vp] = x;
}
VC[vc_base] = base + inc;
```

VLDS(cont.) Variable Stride Signed Vector Load

Description

The VPs perform a strided vector load into $\mathtt{vr}_{\mathtt{dest}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be aligned to the width of the data in memory. The *signed* stride is given by `vstride` (default is $\mathtt{vstride}_0$). The stride is in terms of elements, not in terms of bytes. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The loaded value is sign-extended to the virtual processor width.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdEL`	Vector Address Error (Load)
`vMod`	Vector TLB Modification Exception
`vTLBL`	Vector TLB Exception (Load)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

Stride may be negative or zero. All addresses are checked and translated before any values are loaded. A load fault causes an exception only if the instruction is non-speculative. A speculative faulting load does not load any value. The only difference between `vlds` and `vlds.u` is that `vlds` sign-extends loaded data, and `vlds.u` zero-extends loaded data.

Variable Stride Unsigned Vector Load — VLDS.U

	31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
VMEMS-Type	cop2	1	op	dest	base	inc	stride	op
	6	1	4	5	4	3	3	6

Assembly

$$\text{vlds.u}\left\{\begin{array}{l}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\end{array}\right\}[.1]\ \texttt{vr}_{\texttt{dest}}[,\texttt{vbase}[,\texttt{vstride}[,\texttt{vinc}]]]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + nbytes * stride * vp;
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (READ, nbytes, va, &pa[vp]);
    }
    if (!speculative && fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (fault[vp] != Exc_none) {
      if (speculative) b = true;
      x = VR[dest][vp];
    } else {
      if (speculative) b = VF[vfe_L][vp];
      x = unsigned_load (pa[vp], nbytes);
    }
  } else {
    if (speculative) b = VF[vfe_L][vp];
    x = VR[dest][vp];
  }
  if (speculative) VF[vfe_L][vp] = b;
  VR[dest][vp] = x;
}
VC[vc_base] = base + inc;
```

VLDS.U(cont.) Variable Stride Unsigned Vector Load

Description

Operation is identical to `vlds`, except that loaded values are zero-extended to the virtual processor width instead of sign-extended.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdEL`	Vector Address Error (Load)
`vMod`	Vector TLB Modification Exception
`vTLBL`	Vector TLB Exception (Load)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

`vlds.u.l` is not provided, as it would be functionally identical to `vlds.l`.

Indexed Signed Vector Load — VLDX

VMEMX-Type	31–26	25	24–21	20–16	15–12	11–7	6	5–0
Field	cop2	1	op	dest	base	offsets	0	op
Width	6	1	4	5	4	5	1	6

Assembly

$$\text{vldx}\begin{Bmatrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{offsets}}[,\texttt{vbase}]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + VR[offsets][vp];
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (READ, nbytes, va, &pa[vp]);
    }
    if (!speculative && fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (fault[vp] != Exc_none) {
      if (speculative) b = true;
      x = VR[dest][vp];
    } else {
      if (speculative) b = VF[vfe_L][vp];
      x = signed_load (pa[vp], nbytes);
    }
  } else {
    if (speculative) b = VF[vfe_L][vp];
    x = VR[dest][vp];
  }
  if (speculative) VF[vfe_L][vp] = b;
  VR[dest][vp] = x;
}
```

VLDX(cont.) Indexed Signed Vector Load

Description

The VPs perform an indexed vector load into $\texttt{vr}_{\texttt{dest}}$. The base address is given by `vbase` (default is $\texttt{vbase}_0$). The *signed* offsets are given by $\texttt{vr}_{\texttt{offsets}}$. The offsets are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data in memory. The width of each element in memory is given by the opcode. The loaded value is sign-extended to the virtual processor width.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdEL`	Vector Address Error (Load)
`vMod`	Vector TLB Modification Exception
`vTLBL`	Vector TLB Exception (Load)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

All addresses are checked and translated before any values are loaded. A load fault causes an exception only if the instruction is non-speculative. A speculative faulting load does not load any value. The only difference between `vldx` and `vldx.u` is that `vldx` sign-extends loaded data, and `vldx.u` zero-extends loaded data.

Indexed Unsigned Vector Load — VLDX.U

VMEMX-Type

31–26	25	24–21	20–16	15–12	11–7	6	5–0
cop2	1	op	dest	base	offsets	0	op
6	1	4	5	4	5	1	6

Assembly

$$\text{vldx.u}\left\{\begin{array}{l}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\end{array}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{offsets}}[,\texttt{vbase}]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + VR[offsets][vp];
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (READ, nbytes, va, &pa[vp]);
    }
    if (!speculative && fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (fault[vp] != Exc_none) {
      if (speculative) b = true;
      x = VR[dest][vp];
    } else {
      if (speculative) b = VF[vfe_L][vp];
      x = unsigned_load (pa[vp], nbytes);
    }
  } else {
    if (speculative) b = VF[vfe_L][vp];
    x = VR[dest][vp];
  }
  if (speculative) VF[vfe_L][vp] = b;
  VR[dest][vp] = x;
}
```

VLDX.U(cont.) Indexed Unsigned Vector Load

Description

Operation is identical to `vldx`, except that loaded values are zero-extended to the virtual processor width instead of sign-extended.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdEL`	Vector Address Error (Load)
`vMod`	Vector TLB Modification Exception
`vTLBL`	Vector TLB Exception (Load)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

`vldx.u.l` is not provided, as it would be functionally identical to `vldx.l`.

Vector Floating-Point Multiply Add VMADD.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmadd}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

VMADD.fmt(cont.) Vector Floating-Point Multiply Add

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = mul_add_s (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = mul_add_d (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VMADD.fmt(cont.) Vector Floating-Point Multiply Add

Description

Each unmasked VP adds the floating-point product of $\mathtt{vr}_{\mathtt{src1}}/\mathtt{vs}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$ into $\mathtt{vr}_{\mathtt{dest}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

Signed Vector Integer Maximum VMAX

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmax}\begin{cases}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vr}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vs}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x > y ? x : y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the greater of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$, treated as signed integers.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VMAX.U Unsigned Vector Integer Maximum

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmax.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x > y ? x : y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the greater of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$, treated as unsigned integers.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Commit Speculative Arithmetic VMCOMMIT

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vmcommit[.exact][.1]

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
L = false;
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    L = L || VF[vfe_L][vp];
  }
}
if (L) {
  Raise vSLE;
}
```

Description

Any pending vector memory exceptions generated by speculative operations are raised. This instruction operates under mask and vector length. The exceptions are precise if either `.exact` is specified, or if $\texttt{vc}_{\texttt{vmode}}$ specifies exact exceptions.

Exceptions

None.

Notes

None.

VMCTS Vector Move Control To Scalar

VMCTS-Type

31–26	25	24–21	20–16	15–10	9–6	5–0
cop2	1	op	dest	src	0	op
6	1	4	5	6	4	6

Assembly

vmcts vs_{dest}, vc_{src}

Operation

```
if (!reg_user_readable (src)) { /* assume user mode for now */
  Raise vIUI;
}
x = pdinst->get_src_vc (src);
VS[dest] = x;
```

Description

Register vc_{src} is copied to vs_{dest}.

Exceptions

vIUI	Illegal Use of Instruction Exception

Notes

None.

Vector Merge VMERGE

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vmerge}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    if (VF[mask][vp]) {
      z = y;
    } else {
      z = x;
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each VP copies into $\texttt{vr}_{\texttt{dest}}$ either $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ if the mask is 0, or $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$ if the mask is 1. At least one source is a vector. Scalar sources are truncated to the virtual processor width.

VMERGE(cont.) Vector Merge

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

This instruction can be used to efficiently replicate a scalar value into a vector register.

Signed Vector Integer Minimum VMIN

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmin}\begin{cases}\texttt{.vv}[.1] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv}[.1] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x < y ? x : y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the lesser of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$, treated as signed integers.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VMIN.U Unsigned Vector Integer Minimum

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmin.u}\left\{\begin{array}{l}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{array}\right.$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x < y ? x : y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the lesser of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$, treated as unsigned integers.

Exceptions

vIVL	Invalid Vector Length Exception

Notes

None.

Signed Vector Integer Modulus VMOD

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmod}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    if (y == 0) {
      z = VR[dest][vp];
    } else {
      z = x % y;
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the signed integer modulus of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector.

VMOD(cont.) Signed Vector Integer Modulus

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

The result is undefined if the divisor is zero.

Unsigned Vector Integer Modulus — VMODU

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmod.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{dest}\texttt{,}\ \texttt{vr}_{src1}\texttt{,}\ \texttt{vr}_{src2}\\ \texttt{.sv[.1]} & \texttt{vr}_{dest}\texttt{,}\ \texttt{vs}_{src1}\texttt{,}\ \texttt{vr}_{src2}\\ \texttt{.vs[.1]} & \texttt{vr}_{dest}\texttt{,}\ \texttt{vr}_{src1}\texttt{,}\ \texttt{vs}_{src2}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    if (y == 0) {
      z = VR[dest][vp];
    } else {
      z = x % y;
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{dest}$ the unsigned integer modulus of $\texttt{vs}_{src1}/\texttt{vr}_{src1}$ and $\texttt{vs}_{src2}/\texttt{vr}_{src2}$, where at least one source is a vector.

VMODU(cont.) Unsigned Vector Integer Modulus

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

The result is undefined if the divisor is zero.

Vector Move Scalar To Control — VMSTC

VMSTC-Type

31–26	25	24–21	20–16	15–10	9–6	5–0
cop2	1	op	src	dest	0	op
6	1	4	5	6	4	6

Assembly

vmstc $\texttt{vc}_{\texttt{dest}}$, $\texttt{vs}_{\texttt{src}}$

Operation

```
if (dest != VcVl && dest != VcVpw && dest != VcShamt &&
    dest != VcIndex && dest != VcAt) {
  Raise vIUI;
}
x = VS[src];
if (dest == VcVpw) {
  assert (x <= 3);
  VC[vpw] = x;
  VC[mvl] = MVL[x];
  VC[logmvl] = LOGMVL[x];
} else {
  VC[dest] = x;
}
```

Description

Register $\texttt{vs}_{\texttt{src}}$ is copied to $\texttt{vc}_{\texttt{dest}}$. Writing $\texttt{vc}_{\texttt{vpw}}$ changes $\texttt{vc}_{\texttt{mvl}}$, $\texttt{vc}_{\texttt{logmvl}}$, and $\texttt{vc}_{\texttt{vsafevl}}$ as a side-effect. This instruction may write only $\texttt{vc}_{\texttt{vl}}$, $\texttt{vc}_{\texttt{vpw}}$, $\texttt{vc}_{\texttt{vshamt}}$, $\texttt{vc}_{\texttt{vindex}}$, and $\texttt{vc}_{\texttt{vcat}}$.

Exceptions

vInt	Vector Integer Exception
vFPE	Vector Floating-Point Exception
vIUI	Illegal Use of Instruction Exception

Notes

None.

VMSUB.fmt — Vector Floating-Point Multiply Subtract

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmsub}\left\{\begin{matrix}\texttt{.s}\\ \texttt{.d}\end{matrix}\right\}\left\{\begin{matrix}\texttt{.vv[.1]} & \mathbf{vr}_{\texttt{dest}}\texttt{,} & \mathbf{vr}_{\texttt{src1}}\texttt{,} & \mathbf{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \mathbf{vr}_{\texttt{dest}}\texttt{,} & \mathbf{vs}_{\texttt{src1}}\texttt{,} & \mathbf{vr}_{\texttt{src2}}\end{matrix}\right.$$

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = mul_sub_s (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VMSUB.fmt(cont.) Vector Floating-Point Multiply Subtract

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = mul_sub_d (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Description

Each unmasked VP subtracts $\mathtt{vr}_{\mathtt{dest}}$ from the floating-point product of $\mathtt{vr}_{\mathtt{src1}}$/$\mathtt{vs}_{\mathtt{src1}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

VMUL.fmt — Vector Floating-Point Multiply

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmul}\begin{Bmatrix}\texttt{.s}\\ \texttt{.d}\end{Bmatrix}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = mul_s (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VMUL.fmt(cont.) Vector Floating-Point Multiply

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = mul_d (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Description

Each unmasked VP places the floating-point product of $\mathtt{vr}_{\mathtt{src1}}$/$\mathtt{vs}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$ into $\mathtt{vr}_{\mathtt{dest}}$.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception

Notes

None.

VMULHI Vector Integer Multiply High

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmulhi}\begin{cases}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = signed_multiply_high (x, y);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the upper half of the full-VP-width signed integer product of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

Vector Integer Multiply High VMULHI.U

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vmulhi.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = unsigned_multiply_high (x, y);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the upper half of the full-VP-width unsigned integer product of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VMULLO — Vector Integer Multiply Low

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vmullo}\begin{cases} \texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x * y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the lower half of the full-VP-width integer product of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

There are not seperate signed and unsigned multiply low instructions, because signed and unsiged multiply low are identical operations.

Vector Floating-Point Negate VNEG.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vneg}\left\{\begin{array}{l}\texttt{.s}\\ \texttt{.d}\end{array}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\ \texttt{vr}_{\texttt{src}}$$

VNEG.fmt(cont.) Vector Floating-Point Negate

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = neg_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = neg_d (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VNEG.fmt(cont.) Vector Floating-Point Negate

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the floating-point square root of $\mathtt{vr}_{\mathtt{src}}$.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception

Notes

None.

Vector Floating-Point Negative Multiply Add — VNMADD.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vnmadd}\left\{\begin{array}{l}\texttt{.s}\\ \texttt{.d}\end{array}\right\}\left\{\begin{array}{l}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{array}\right.$$

VNMADD.fmt(cont.) Vector Floating-Point Negative Multiply Add

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = neg_mul_add_s (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = neg_mul_add_d (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VNMADD.fmt(cont.) Vector Floating-Point Negative Multiply Add

Description

Each unmasked VP adds the floating-point product of $\mathtt{vr}_{\mathtt{src1}}/\mathtt{vs}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$ into $\mathtt{vr}_{\mathtt{dest}}$, and negates the result.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

Vector Floating-Point Negative Multiply Subtract VNMSUB.fmt

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vnmsub}\begin{Bmatrix}\texttt{.s}\\\texttt{.d}\end{Bmatrix}\begin{Bmatrix}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, } \texttt{vr}_{\texttt{src1}}\texttt{, } \texttt{vr}_{\texttt{src2}}\\\texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, } \texttt{vs}_{\texttt{src1}}\texttt{, } \texttt{vr}_{\texttt{src2}}\end{Bmatrix}$$

VNMSUB.fmt(cont.) Vector Floating-Point Negative Multiply Subtract

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = neg_mul_sub_s (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = VR[src3][vp];
    z = neg_mul_sub_d (x, y, z, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VNMSUB.fmt(cont.) Vector Floating-Point Negative Multiply Subtract

Description

Each unmasked VP subtracts $\texttt{vr}_{\texttt{dest}}$ from the floating-point product of $\texttt{vr}_{\texttt{src1}}/\texttt{vs}_{\texttt{src1}}$, and negates the result.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

Vector Nor VNOR

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\texttt{vnor}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, } \texttt{vr}_{\texttt{src1}}\texttt{, } \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, } \texttt{vs}_{\texttt{src1}}\texttt{, } \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = ~(x | y);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each VP writes into $\texttt{vr}_{\texttt{dest}}$ the bit-wise logical *nor* of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VOR — Vector Or

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vor}\begin{cases}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vr}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vs}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x | y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each VP writes into $\texttt{vr}_{\texttt{dest}}$ the bit-wise logical *or* of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

Vector Floating-Point Reciprocal — VRECIP.fmt

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	0	src	dest	op
6	1	4	5	5	5	6

Assembly

vrecip $\left\{\begin{matrix}\texttt{.s}\\ \texttt{.d}\end{matrix}\right\}$ [.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

VRECIP.fmt(cont.) Vector Floating-Point Reciprocal

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = recip_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = recip_d (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VRECIP.fmt(cont.) Vector Floating-Point Reciprocal

Description

Each unmasked VP places into vr_{dest} the floating-point reciprocal of vr_{src}.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception

Notes

None.

Vector Floating-Point Round VROUND

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vround}\left\{\begin{array}{l}\texttt{.w}\\\texttt{.l}\end{array}\right\}\left\{\begin{array}{l}\texttt{.s}\\\texttt{.d}\end{array}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\ \texttt{vr}_{\texttt{src}}$$

VROUND(cont.) Vector Floating-Point Round

Operation (Single to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = round_s_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = round_d_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VROUND(cont.) Vector Floating-Point Round

Operation (Single to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = round_s_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = round_d_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VROUND(cont.) Vector Floating-Point Round

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the result of converting $\mathtt{vr}_{\mathtt{src}}$ from a floating-point format to a signed integer format, using the *round* rounding mode.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

Vector Floating-Point Reciprocal Square Root — VRSQRT.fmt

VR-Type

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vrsqrt}\left\{\begin{matrix}\texttt{.s}\\ \texttt{.d}\end{matrix}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}}, \texttt{vr}_{\texttt{src}}$$

VRSQRT.fmt(cont.) Vector Floating-Point Reciprocal Square Root

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = rsqrt_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = rsqrt_d (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VRSQRT.fmt(cont.) Vector Floating-Point Reciprocal Square Root

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the floating-point reciprocal square root of $\mathtt{vr}_{\mathtt{src}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

Vector Register Sync VRSYNC

	31–26	25	24–21	20–6	5–0
VMB-Type	cop2	1	op	???	op
	6	1	4	15	6

Assembly

$$\text{vrsync}\left\{\begin{array}{c}\texttt{.sav}\\ \texttt{.vav}\\ \texttt{.vp}\end{array}\right\}\left\{\begin{array}{c}\phi\\ \texttt{.raw}\\ \texttt{.war}\\ \texttt{.waw}\end{array}\right\}\ \texttt{vr}_{\texttt{dest}}$$

Operation

```
/* operation is a nop in a simple implementation */
```

Description

Enforces scalar-after-vector, vector-after-scalar, vector-after-vector, or intra-vp memory ordering. If none of raw/war/waw are specified, then all are in effect. The most recent memory reference to $\texttt{vr}_{\texttt{dest}}$ preceeding the sync must appear to execute before all relevant memory references following the sync. Order is defined as program order.

Exceptions

None.

Notes

None.

VSADD — Signed Saturating Add

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

vsadd
- `.vv[.1]` $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src1}}$, $\mathtt{vr}_{\mathtt{src2}}$
- `.sv[.1]` $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vs}_{\mathtt{src1}}$, $\mathtt{vr}_{\mathtt{src2}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = signed_saturating_add (x, y, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the signed integer sum of $\mathtt{vs}_{\mathtt{src1}}$/$\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$. The sum saturates to the VP width instead of overflowing.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VSADD.U — Unsigned Saturating Add

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsadd.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = unsigned_saturating_add (x, y, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the unsigned integer sum of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. The sum saturates to the VP width instead of overflowing.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VSAT Signed Vector Saturate

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsat}\left\{\begin{matrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\end{matrix}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{src}}$$

Operation

```
assert (sat_width == B || sat_width == H || sat_width == W);
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2 && sat_width == W || vpw < 1 && sat_width == B) {
  Raise vIUI;
}
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = signed_saturate (x, sat_width, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP places into $\texttt{vr}_{\texttt{dest}}$ the result of saturating $\texttt{vr}_{\texttt{src}}$ to a signed integer narrower than the VP width. The result is sign-extended to the VP width.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

It is illegal to saturate to wider than the VP width.

VSAT.SU Signed to Unsigned Vector Saturate

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsat.su}\begin{Bmatrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\ \texttt{vr}_{\texttt{src}}$$

Operation

```
assert (sat_width == B || sat_width == H || sat_width == W || sat_width == L);
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3 && sat_width == L || vpw < 2 && sat_width == W || vpw < 1 && sat_width == B) {
  Raise vIUI;
}
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = signed_to_unsigned_saturate (x, sat_width, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the result of saturating $\mathtt{vr}_{\mathtt{src}}$ from a signed VP width value to an unsigned value that is as wide or narrower than the VP width. The result is zero-extended to the VP width.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

It is illegal to saturate to wider than the VP width.

VSAT.U Unsigned Vector Saturate

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsat.u}\begin{Bmatrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\end{Bmatrix}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}}, \texttt{vr}_{\texttt{src}}$$

Operation

```
assert (sat_width == B || sat_width == H || sat_width == W);
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2 && sat_width == W || vpw < 1 && sat_width == B) {
  Raise vIUI;
}
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = unsigned_saturate (x, sat_width, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP places into $\texttt{vr}_{\texttt{dest}}$ the result of saturating $\texttt{vr}_{\texttt{src}}$ to an unsigned integer narrower than the VP width. The result is zero-extended to the VP width.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

It is illegal to saturate to wider than the VP width.

VSATVL Saturate Vector Length

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vsatvl

Operation

```
VC[vc_vl] = vl > mvl ? mvl : vl;
```

Description

The vector length register is saturated to the maximum vector length.

Exceptions

None.

Notes

None.

Vector Shift Left Logical — VSLL

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsll}\begin{cases}\texttt{.vv[.1]}\ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs[.1]}\ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    y &= shamt_mask;
    z = x << y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the signed integer contents of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ left-shifted by the number of bits specified by $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector. The shift amount is the b-bit unsigned integer taken from the low-order end of $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where 2^b equals the VP width in bits. The result is zero-filled.

VSLL(cont.) Vector Shift Left Logical

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

Only the lower b bits of the register from which the shift amount is taken are used (2^b equals the VP width in bits).

Signed Saturating Left Shift — VSLS

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vsls[.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = signed_shift_left_and_sat (x, shamt, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the signed saturating left shift of $\texttt{vr}_{\texttt{src}}$. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

VSLS(cont.) Signed Saturating Left Shift

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

Unsigned Saturating Left Shift — VSLS.U

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vsls.u[.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = unsigned_shift_left_and_sat (x, shamt, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the unsigned saturating left shift of $\texttt{vr}_{\texttt{src}}$. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

VSLS.U(cont.) Unsigned Saturating Left Shift

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Vector Floating-Point Square Root VSQRT.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vsqrt$\left\{\begin{matrix}\texttt{.s}\\ \texttt{.d}\end{matrix}\right\}$[.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src}}$

VSQRT.fmt(cont.) Vector Floating-Point Square Root

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = sqrt_s (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = sqrt_d (x, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VSQRT.fmt(cont.) Vector Floating-Point Square Root

Description

Each unmasked VP places into $\mathtt{vr}_{\mathtt{dest}}$ the floating-point square root of $\mathtt{vr}_{\mathtt{src}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

Vector Arithmetic Right Shift — VSRA

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsra}\begin{cases}\texttt{.vv[.1]}\ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]}\ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.vs[.1]}\ \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    y &= shamt_mask;
    z = shift_right_arith (x, y);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the signed integer contents of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ right-shifted by the number of bits specified by $\texttt{vs}_{\texttt{src2}}$/$\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector. The shift amount is the b-bit unsigned integer taken from the low-order end of $\texttt{vs}_{\texttt{src2}}$/$\texttt{vr}_{\texttt{src2}}$, where 2^b equals the VP width in bits. The result is sign-extended.

VSRA(cont.) Vector Arithmetic Right Shift

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

Only the lower b bits of the register from which the shift amount is taken are used (2^b equals the VP width in bits).

Vector Shift Right Logical VSRL

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsrl}\begin{cases}\texttt{.vv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    y &= shamt_mask;
    z = x >> y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\texttt{vr}_{\texttt{dest}}$ the signed integer contents of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ right-shifted by the number of bits specified by $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where at least one source is a vector. The shift amount is the b-bit unsigned integer taken from the low-order end of $\texttt{vs}_{\texttt{src2}}/\texttt{vr}_{\texttt{src2}}$, where 2^b equals the VP width in bits. The result is zero-extended.

VSRL(cont.) Vector Shift Right Logical

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

VSRLnotes

Signed Shift Right And Round VSRR

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vsrr[.1] $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src}}$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = signed_shift_right_and_round (x, shamt, RM);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the right arithmetic shift of $\mathtt{vr}_{\mathtt{src}}$. The result is rounded as per the fixed-point rounding mode. The shift amount is taken from $\mathtt{vc}_{\mathtt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VSRR.U Unsigned Shift Right And Round

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

vsrr.u[.1] $\mathtt{vr}_{\mathtt{dest}}$, $\mathtt{vr}_{\mathtt{src}}$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = unsigned_shift_right_and_round (x, shamt, RM);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the right logical shift of $\mathtt{vr}_{\mathtt{src}}$. The result is rounded as per the fixed-point rounding mode. The shift amount is taken from $\mathtt{vc}_{\mathtt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Signed Saturating Subtract VSSUB

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vssub}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}} \\ \texttt{.vs[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = signed_saturating_sub (x, y, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

VSSUB(cont.) Signed Saturating Subtract

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the signed integer subtraction of $\mathtt{vs}_{\mathtt{src1}}$/$\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vs}_{\mathtt{src2}}$/$\mathtt{vr}_{\mathtt{src2}}$, where at least one source is a vector. The difference saturates to the VP width instead of overflowing.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Signed Saturating Subtract VSSUB.U

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vssub.u}\begin{cases}\texttt{.vv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.vs}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = unsigned_saturating_sub (x, y, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

VSSUB.U(cont.) Signed Saturating Subtract

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the unsigned integer subtraction of $\mathtt{vs}_{\mathtt{src1}}/\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vs}_{\mathtt{src2}}/\mathtt{vr}_{\mathtt{src2}}$, where at least one source is a vector. The difference saturates to zero instead of underflowing.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Unit Stride Vector Store — VST

	31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
VMEM-Type	cop2	1	op	src	base	inc	0	op
	6	1	4	5	4	3	3	6

Assembly

$$\text{vst}\left\{\begin{array}{l}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{array}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{src}}[,\texttt{vbase}[,\texttt{vinc}]]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + nbytes * vp;
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (WRITE, nbytes, va, &pa[vp]);
    }
    if (fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    x = VR[src][vp];
    store (pa[vp], nbytes, x);
  }
}
VC[vc_base] = base + inc;
```

VST(cont.) Unit Stride Vector Store

Description

The VPs perform a contiguous vector store of $\mathtt{vr}_{\mathtt{src}}$. The base address is given by `vbase` (default is $\mathtt{vbase}_0$), and must be aligned to the width of the data in memory. The signed increment in `vinc` (default is $\mathtt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The register value is truncated from the VP width to the memory width. The VPs access memory in order.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception
vAdES	Vector Address Error (Store)
vMod	Vector TLB Modification Exception
vTLBS	Vector TLB Exception (Store)
vDBEA	Vector Bus Error Exception (Address)
vDBED	Vector Bus Error Exception (Data)
vWatch	Watch Address Exception

Notes

All addresses are checked and translated before any values are stored.

Variable Stride Vector Store VSTS

	31–26	25	24–21	20–16	15–12	11–9	8–6	5–0
VMEMS-Type	cop2	1	op	src	base	inc	stride	op
	6	1	4	5	4	3	3	6

Assembly

$$\text{vsts}\left\{\begin{array}{l}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{array}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{src}}[,\texttt{vbase}[,\texttt{vstride}[,\texttt{vinc}]]]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + nbytes * stride * vp;
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (WRITE, nbytes, va, &pa[vp]);
    }
    if (fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    x = VR[src][vp];
    store (pa[vp], nbytes, x);
  }
}
VC[vc_base] = base + inc;
```

VSTS(cont.) Variable Stride Vector Store

Description

The VPs perform a contiguous vector store of $\texttt{vr}_{\texttt{src}}$. The base address is given by `vbase` (default is $\texttt{vbase}_0$), and must be aligned to the width of the data in memory. The *signed* stride is given by `vstride` (default is $\texttt{vstride}_0$). The stride is in terms of elements, not in terms of bytes. The signed increment in `vinc` (default is $\texttt{vinc}_0$) is added to `vbase` as a side-effect. The width of each element in memory is given by the opcode. The register value is truncated from the VP width to the memory width. The VPs access memory in order.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdES`	Vector Address Error (Store)
`vMod`	Vector TLB Modification Exception
`vTLBS`	Vector TLB Exception (Store)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

All addresses are checked and translated before any values are stored.

Unordered Indexed Vector Store VSTX

	31–26	25	24–21	20–16	15–12	11–7	6	5–0
VMEMX-Type	cop2	1	op	src	base	offsets	0	op
	6	1	4	5	4	5	1	6

Assembly

$$\text{vstx}\begin{Bmatrix}\texttt{.b}\\ \texttt{.h}\\ \texttt{.w}\\ \texttt{.l}\end{Bmatrix}[.1]\ \texttt{vr}_{\texttt{src}}, \texttt{vr}_{\texttt{offsets}}\,[, \texttt{vbase}]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + VR[offsets][vp];
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (WRITE, nbytes, va, &pa[vp]);
    }
    if (fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    x = VR[src][vp];
    store (pa[vp], nbytes, x);
  }
}
```

VSTX(cont.) Unordered Indexed Vector Store

Description

The VPs perform an indexed vector store of $\texttt{vr}_{\texttt{src}}$. The base address is given by `vbase` (default is $\texttt{vbase}_0$). The *signed* offsets are given by $\texttt{vr}_{\texttt{offsets}}$. The offsets are in units of bytes, not in units of elements. The effective addresses must be aligned to the width of the data in memory. The register value is truncated from the VP width to the memory width. The stores may be performed in any order.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdES`	Vector Address Error (Store)
`vMod`	Vector TLB Modification Exception
`vTLBS`	Vector TLB Exception (Store)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

All addresses are checked and translated before any values are stored. An implementation is free to perform the stores in any order. The implementation shown performs them in order.

Ordered Indexed Vector Store VSTXO

	31–26	25	24–21	20–16	15–12	11–7	6	5–0
VMEMX-Type	cop2	1	op	src	base	offsets	0	op
	6	1	4	5	4	5	1	6

Assembly

$$\text{vstxo}\left\{\begin{array}{l}\texttt{.b}\\\texttt{.h}\\\texttt{.w}\\\texttt{.l}\end{array}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{src}},\ \texttt{vr}_{\texttt{offsets}}[,\ \texttt{vbase}]$$

Operation

```
assert (nbytes == 1 || nbytes == 2 || nbytes == 4 || nbytes == 8);
if (vl > mvl) {
  Raise vIVL;
}
if (nbytes > (1 << vpw)) {
  Raise vIUI;
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    va = base + VR[offsets][vp];
    if (va & (nbytes - 1)) {
      fault[vp] = vAdEL;
    } else {
      fault[vp] = translate (WRITE, nbytes, va, &pa[vp]);
    }
    if (fault[vp] != Exc_none) {
      Raise fault[vp];
    }
  }
}
for (vp = 0; vp < vl; vp++) {
  if (VF[mask][vp]) {
    x = VR[src][vp];
    store (pa[vp], nbytes, x);
  }
}
```

VSTXO(cont.) Ordered Indexed Vector Store

Description

Operation is identical to `vstx`, except that the VPs access memory in order.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception
`vAdES`	Vector Address Error (Store)
`vMod`	Vector TLB Modification Exception
`vTLBS`	Vector TLB Exception (Store)
`vDBEA`	Vector Bus Error Exception (Address)
`vDBED`	Vector Bus Error Exception (Data)
`vWatch`	Watch Address Exception

Notes

Signed Vector Integer Subtract — VSUB

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsub}\begin{cases}\texttt{.vv}[.1] & \texttt{vr}_{\text{dest}}\texttt{,}\ \texttt{vr}_{\text{src1}}\texttt{,}\ \texttt{vr}_{\text{src2}}\\ \texttt{.sv}[.1] & \texttt{vr}_{\text{dest}}\texttt{,}\ \texttt{vs}_{\text{src1}}\texttt{,}\ \texttt{vr}_{\text{src2}}\\ \texttt{.vs}[.1] & \texttt{vr}_{\text{dest}}\texttt{,}\ \texttt{vr}_{\text{src1}}\texttt{,}\ \texttt{vs}_{\text{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
raise_F = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = x - y;
    if (sub_overflow (x, y, z)) {
      f = true;
      if (!speculative && F_enabled) {
        raise_F = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_F][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_F][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_F][vp] = f;
}
if (raise_F) {
  Raise vAri;
}
```

VSUB(cont.) Signed Vector Integer Subtract

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the signed integer subtraction of $\mathtt{vs}_{\mathtt{src1}}/\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vs}_{\mathtt{src2}}/\mathtt{vr}_{\mathtt{src2}}$, where at least one source is a vector.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Unsigned Vector Integer Subtract VSUB.U

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsub.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vs}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = x - y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

VSUB.U(cont.) Unsigned Vector Integer Subtract

Description

Each unmasked VP writes into $\mathtt{vr}_{\mathtt{dest}}$ the unsigned integer subtraction of $\mathtt{vs}_{\mathtt{src1}}/\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vs}_{\mathtt{src2}}/\mathtt{vr}_{\mathtt{src2}}$, where at least one source is a vector.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

The only difference between `vsub` and `vsub.u` is that `vsub` can overflow, while `vsub.u` cannot.

Vector Floating-Point Subtract VSUB.fmt

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vsub}\left\{\begin{matrix}\texttt{.s}\\ \texttt{.d}\end{matrix}\right\}\left\{\begin{matrix}\texttt{.vv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,} & \texttt{vr}_{\texttt{src1}}\texttt{,} & \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,} & \texttt{vs}_{\texttt{src1}}\texttt{,} & \texttt{vr}_{\texttt{src2}}\\ \texttt{.vs}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,} & \texttt{vr}_{\texttt{src1}}\texttt{,} & \texttt{vs}_{\texttt{src2}}\end{matrix}\right.$$

VSUB.fmt(cont.) Vector Floating-Point Subtract

Operation (Single Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = sub_s (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Double Precision)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else if (sv) {
      x = VS[src1];
      y = VR[src2][vp];
    } else {
      x = VR[src1][vp];
      y = VS[src2];
    }
    z = sub_d (x, y, FS, RM, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VSUB.fmt(cont.) Vector Floating-Point Subtract

Description

Each unmasked VP places the floating-point difference of $\mathtt{vr}_{\mathtt{src1}}$/$\mathtt{vs}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$/$\mathtt{vf}_{\mathtt{src2}}$ into $\mathtt{vr}_{\mathtt{dest}}$, where at least one souce is a vector.

Exceptions

vIVL	Invalid Vector Length Exception
vIUI	Illegal Use of Instruction Exception

Notes

None.

Vector Sync VSYNC

	31–26	25	24–21	20–6	5–0
VMB-Type	cop2	1	op	???	op
	6	1	4	15	6

Assembly

$$\text{vsync}\left\{\begin{array}{l}\texttt{.sav}\\ \texttt{.vas}\\ \texttt{.vav}\\ \texttt{.vp}\end{array}\right\}\left\{\begin{array}{c}\phi\\ \texttt{.raw}\\ \texttt{.war}\\ \texttt{.waw}\end{array}\right\}$$

Operation

```
/* operation is a nop in a simple implementation */
```

Description

Enforces scalar-after-vector, vector-after-scalar, vector-after-vector, or intra-vp memory ordering. If none of raw/war/waw are specified, then all are in effect. All relevant memory references preceeding the sync must appear to execute before all relevant memory references following the sync. Order is defined as program order.

Exceptions

None.

Notes

None.

VTLBP Vector TLB Probe

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vtlbp

Operation

```
fprintf (stderr, "vtlbp not yet implementedn");
```

Description

This instruction is not yet implemented.

Exceptions

None.

Notes

None.

Vector TLB Read VTLBR

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vtlbr

Operation

```
fprintf (stderr, "vtlbr not yet implementedn");
```

Description

This instruction is not yet implemented.

Exceptions

None.

Notes

None.

VTLBWI Vector TLB Write Indexed

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vtlbwi

Operation

```
fprintf (stderr, "vtlbwi not yet implementedn");
```

Description

This instruction is not yet implemented.

Exceptions

None.

Notes

None.

Vector TLB Write Random — VTLBWR

VR-Type	31–26	25	24–21	20–16	15–11	10–6	5–0
	cop2	1	op	0	0	0	op
	6	1	4	5	5	5	6

Assembly

vtlbwr

Operation

```
fprintf (stderr, "vtlbwr not yet implementedn");
```

Description

This instruction is not yet implemented.

Exceptions

None.

Notes

None.

VTRUNC — Vector Floating-Point Truncate

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	0	src	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vtrunc}\left\{\begin{matrix}\texttt{.w}\\\texttt{.l}\end{matrix}\right\}\left\{\begin{matrix}\texttt{.s}\\\texttt{.d}\end{matrix}\right\}[\texttt{.1}]\ \texttt{vr}_{\texttt{dest}},\texttt{vr}_{\texttt{src}}$$

Operation (Single to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 2) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = trunc_s_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VTRUNC(cont.) Vector Floating-Point Truncate

Operation (Double to Word)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = trunc_d_to_w (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Operation (Single to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = trunc_s_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

VTRUNC(cont.) Vector Floating-Point Truncate

Operation (Double to Long)

```
if (vl > mvl) {
  Raise vIVL;
}
if (vpw < 3) {
  Raise vIUI;
}
raise_I = raise_U = raise_O = raise_Z = raise_V = raise_E = false;
for (vp = 0; vp < mvl; vp++) {
  I = U = O = Z = V = E = false;
  if (vp < vl && VF[mask][vp]) {
    x = VR[src][vp];
    z = trunc_d_to_l (x, FS, &I, &U, &O, &Z, &V, &E);
    if (!speculative) {
      write = true;
      if (I && I_enabled) { raise_I = true; write = false; }
      if (U && U_enabled) { raise_U = true; write = false; }
      if (O && O_enabled) { raise_O = true; write = false; }
      if (Z && Z_enabled) { raise_Z = true; write = false; }
      if (V && V_enabled) { raise_V = true; write = false; }
      if (E) { raise_E = true; write = false; }
      if (!write) {
        z = VR[dest][vp];
      }
    }
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_I][vp] = I || VF[vfe_I][vp];
  VF[vfe_U][vp] = U || VF[vfe_U][vp];
  VF[vfe_O][vp] = O || VF[vfe_O][vp];
  VF[vfe_Z][vp] = Z || VF[vfe_Z][vp];
  VF[vfe_V][vp] = V || VF[vfe_V][vp];
  VF[vfe_E][vp] = E || VF[vfe_E][vp];
}
if (raise_I || raise_U || raise_O || raise_Z || raise_V || raise_E) {
  Raise vAri;
}
```

Description

Each unmasked VP places into $\texttt{vr}_{\texttt{dest}}$ the result of converting $\texttt{vr}_{\texttt{src}}$ from a floating-point format to a signed integer format, using the *truncate* rounding mode.

Exceptions

`vIVL`	Invalid Vector Length Exception
`vIUI`	Illegal Use of Instruction Exception

Notes

None.

VXLMADD Signed Multiply Add Lower Halves

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vxlmadd}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = signed_multiply_lower_halves (x, y);
    z = signed_shift_right_and_round (z, shamt, RM);
    z = signed_saturating_add (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

VXLMADD(cont.) Signed Multiply Add Lower Halves

Description

Each unmasked VP computes the signed integer product of the lower halves of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. This result is added into $\texttt{vr}_{\texttt{dest}}$ after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

If the source registers do not contain valid half-width signed integers, then the operation is not sensible.

Unsigned Multiply Add Lower Halves VXLMADD.U

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vxlmadd.u}\begin{cases}\texttt{.vv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv}[\texttt{.1}] & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

VXLMADD.U(cont.) Unsigned Multiply Add Lower Halves

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = unsigned_multiply_lower_halves (x, y);
    z = unsigned_shift_right_and_round (z, shamt, RM);
    z = unsigned_saturating_add (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP computes the unsigned integer product of the lower halves of $\mathtt{vs}_{\mathtt{src1}}$/$\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$. This result is added into $\mathtt{vr}_{\mathtt{dest}}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from $\mathtt{vc}_{\mathtt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VXLMSUB Signed Multiply Subtract Lower Halves

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vxlmsub}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = signed_multiply_lower_halves (x, y);
    z = signed_shift_right_and_round (z, shamt, RM);
    z = signed_saturating_sub (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

VXLMSUB(cont.) Signed Multiply Subtract Lower Halves

Description

Each unmasked VP computes the signed integer product of the lower halves of vs_{src1}/vr_{src1} and vr_{src2}. This result is subtracted from vr_{dest} after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from vc_{vshamt}.

Exceptions

vIVL	Invalid Vector Length Exception

Notes

If the source registers do not contain valid half-width signed integers, then the operation is not sensible.

Unsigned Multiply Subtract Lower Halves VXLMSUB.U

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vxlmsub.u}\begin{cases} \texttt{.vv[.1]}\ \texttt{vr}_{dest},\ \texttt{vr}_{src1},\ \texttt{vr}_{src2} \\ \texttt{.sv[.1]}\ \texttt{vr}_{dest},\ \texttt{vs}_{src1},\ \texttt{vr}_{src2} \end{cases}$$

VXLMSUB.U(cont.) Unsigned Multiply Subtract Lower Halves

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = unsigned_multiply_lower_halves (x, y);
    z = unsigned_shift_right_and_round (z, shamt, RM);
    z = unsigned_saturating_sub (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Unsigned Multiply Subtract Lower Halves VXLMSUB.U(cont.)

Description

Each unmasked VP computes the unsigned integer product of the lower halves of $\mathtt{vs}_{\mathtt{src1}}$/$\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$. This result is subtracted from $\mathtt{vr}_{\mathtt{dest}}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from $\mathtt{vc}_{\mathtt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VXLMUL — Signed Multiply Lower Halves

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vxlmul}\begin{cases}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \texttt{vr}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \texttt{vs}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = signed_multiply_lower_halves (x, y);
    z = signed_shift_right_and_round (z, shamt, RM);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP computes the signed integer product of the lower halves of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. This result is written into $\texttt{vr}_{\texttt{dest}}$ after an arithmetic right shift and fixed-point round. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

If the source registers do not contain valid half-width signed integers, then the operation is not sensible.

VXLMUL.U Unsigned Multiply Lower Halves

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vxlmul.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = unsigned_multiply_lower_halves (x, y);
    z = unsigned_shift_right_and_round (z, shamt, RM);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP computes the unsigned integer product of the lower halves of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. This result is written into $\texttt{vr}_{\texttt{dest}}$ after a logical right shift and fixed-point round. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

Exceptions

`vIVL`	Invalid Vector Length Exception

Notes

None.

VXOR Vector Xor

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vxor}\begin{cases}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vr}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \ \texttt{vs}_{\texttt{src1}},\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
if (vl > mvl) {
  Raise vIVL;
}
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = x ^ y;
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each VP writes into $\texttt{vr}_{\texttt{dest}}$ the bit-wise logical *xor* of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Signed Multiply Add Upper Halves VXUMADD

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vxumadd}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

VXUMADD(cont.) Signed Multiply Add Upper Halves

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = signed_multiply_upper_halves (x, y);
    z = signed_shift_right_and_round (z, shamt, RM);
    z = signed_saturating_add (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP computes the signed integer product of the upper halves of $\mathtt{vs}_{\mathtt{src1}}$/$\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$. This result is added into $\mathtt{vr}_{\mathtt{dest}}$ after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from $\mathtt{vc}_{\mathtt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VXUMADD.U Unsigned Multiply Add Upper Halves

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vxumadd.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, vr}_{\texttt{src1}}\texttt{, vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{, vs}_{\texttt{src1}}\texttt{, vr}_{\texttt{src2}}\end{cases}$$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = unsigned_multiply_upper_halves (x, y);
    z = unsigned_shift_right_and_round (z, shamt, RM);
    z = unsigned_saturating_add (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

VXUMADD.U(cont.) Unsigned Multiply Add Upper Halves

Description

Each unmasked VP computes the unsigned integer product of the upper halves of $\mathtt{vs}_{\mathtt{src1}}/\mathtt{vr}_{\mathtt{src1}}$ and $\mathtt{vr}_{\mathtt{src2}}$. This result is added into $\mathtt{vr}_{\mathtt{dest}}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from $\mathtt{vc}_{\mathtt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Signed Multiply Subtract Upper Halves — VXUMSUB

	31 26	25	24 21	20 16	15 11	10 6	5 0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vxumsub}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \ \texttt{vr}_{\texttt{src2}}\end{cases}$$

VXUMSUB(cont.) Signed Multiply Subtract Upper Halves

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = signed_multiply_upper_halves (x, y);
    z = signed_shift_right_and_round (z, shamt, RM);
    z = signed_saturating_sub (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

Description

Each unmasked VP computes the signed integer product of the upper halves of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. This result is subtracted from $\texttt{vr}_{\texttt{dest}}$ after an arithmetic right shift and fixed-point round. The final result is saturated. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

VXUMSUB.U Unsigned Multiply Subtract Upper Halves

VR-Type

31–26	25	24–21	20–16	15–11	10–6	5–0
cop2	1	op	src2	src1	dest	op
6	1	4	5	5	5	6

Assembly

$$\text{vxumsub.u}\begin{cases}\texttt{.vv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vr}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\\ \texttt{.sv[.1]} & \texttt{vr}_{\texttt{dest}}\texttt{,}\ \texttt{vs}_{\texttt{src1}}\texttt{,}\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
raise_S = false;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    w = VR[src3][vp];
    z = unsigned_multiply_upper_halves (x, y);
    z = unsigned_shift_right_and_round (z, shamt, RM);
    z = unsigned_saturating_sub (w, z, &saturated);
    if (saturated) {
      f = true;
      if (!speculative && S_enabled) {
        raise_S = true;
        z = VR[dest][vp]; /* don't write result */
      }
    } else {
      f = VF[vfe_S][vp];
    }
  } else {
    z = VR[dest][vp];
    f = VF[vfe_S][vp];
  }
  VR[dest][vp] = z;
  VF[vfe_S][vp] = f;
}
if (raise_S) {
  Raise vAri;
}
```

VXUMSUB.U(cont.) Unsigned Multiply Subtract Upper Halves

Description

Each unmasked VP computes the unsigned integer product of the upper halves of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. This result is subtracted from $\texttt{vr}_{\texttt{dest}}$ after a logical right shift and fixed-point round. The final result is saturated. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Signed Multiply Upper Halves VXUMUL

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

$$\text{vxumul}\begin{cases}\texttt{.vv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \texttt{vr}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}} \\ \texttt{.sv[.1]}\ \ \texttt{vr}_{\texttt{dest}},\ \texttt{vs}_{\texttt{src1}},\ \texttt{vr}_{\texttt{src2}}\end{cases}$$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = signed_multiply_upper_halves (x, y);
    z = signed_shift_right_and_round (z, shamt, RM);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP computes the signed integer product of the upper halves of $\texttt{vs}_{\texttt{src1}}/\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. This result is written into $\texttt{vr}_{\texttt{dest}}$ after an arithmetic right shift and fixed-point round. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

VXUMUL(cont.) Signed Multiply Upper Halves

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.

Unsigned Multiply Upper Halves VXUMUL.U

	31–26	25	24–21	20–16	15–11	10–6	5–0
VR-Type	cop2	1	op	src2	src1	dest	op
	6	1	4	5	5	5	6

Assembly

vxumul.u { .vv[.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vr}_{\texttt{src1}}$, $\texttt{vr}_{\texttt{src2}}$
         { .sv[.1] $\texttt{vr}_{\texttt{dest}}$, $\texttt{vs}_{\texttt{src1}}$, $\texttt{vr}_{\texttt{src2}}$

Operation

```
assert (RM == TRC || RM == UP || RM == NRE || RM == JAM);
if (vl > mvl) {
  Raise vIVL;
}
shamt_mask = (1 << (vpw + 3)) - 1;
shamt = vshamt & shamt_mask;
for (vp = 0; vp < mvl; vp++) {
  if (vp < vl && VF[mask][vp]) {
    if (vv) {
      x = VR[src1][vp];
      y = VR[src2][vp];
    } else {
      x = VS[src1];
      y = VR[src2][vp];
    }
    z = unsigned_multiply_upper_halves (x, y);
    z = unsigned_shift_right_and_round (z, shamt, RM);
  } else {
    z = VR[dest][vp];
  }
  VR[dest][vp] = z;
}
```

Description

Each unmasked VP computes the unsigned integer product of the upper halves of $\texttt{vs}_{\texttt{src1}}$/$\texttt{vr}_{\texttt{src1}}$ and $\texttt{vr}_{\texttt{src2}}$. This result is written into $\texttt{vr}_{\texttt{dest}}$ after a logical right shift and fixed-point round. The shift amount is taken from $\texttt{vc}_{\texttt{vshamt}}$.

VXUMUL.U(cont.) Unsigned Multiply Upper Halves

Exceptions

`vIVL` Invalid Vector Length Exception

Notes

None.