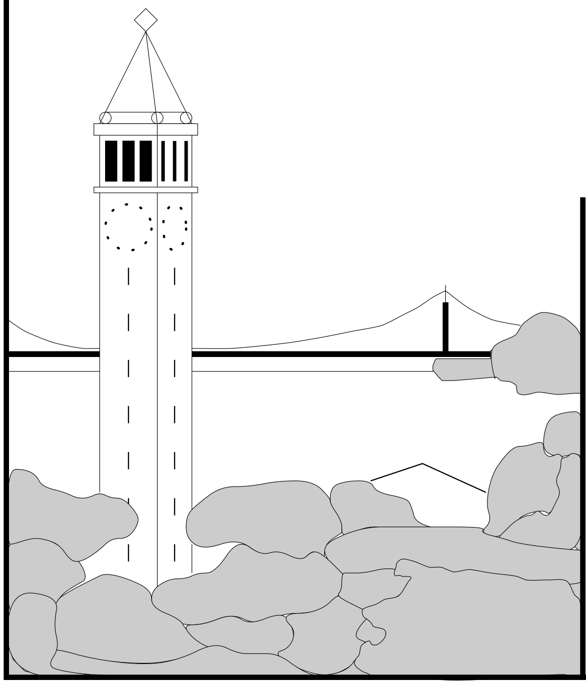
Torrent Architecture Manual

Krste Asanović
David Johnson



Report No. UCB/CSD-97-930

January 1997

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

This manual contains the specification of the Torrent Instruction Set Architecture (ISA). Torrent is a vector ISA designed for digital signal processing applications. Torrent is based on the 32-bit MIPS-II ISA, and this manual is intended to be read as a supplement to the book "MIPS RISC Architecture" by Kane and Heinrich. Torrent is the ISA of the T0 vector microprocessor which is described in the separate "T0 Engineering Data" technical report.

This work was supported by ONR URI Grant N00014-92-J-1617, ARPA contract number N0001493-C0249, NSF Grant No. MIP-9311980, and NSF PYI Award No. MIP-8958568NSF. Additional support was provided by ICSI.

1 Introduction

Torrent is a vector processor instruction set architecture (ISA) designed for digital signal processing applications. The Torrent ISA allows for a range of implementations performing differing numbers of operations per clock cycle, depending on available technology.

Torrent is based on the 32 bit MIPS-II ISA. The scalar unit is MIPS-compliant, with coprocessor instruction extensions for the vector unit. This manual is intended to be read as a supplement to the book "MIPS RISC Architecture" by Kane and Heinrich.

To date, the only implementation of Torrent is T0. T0 is described in detail in the "T0 Engineering Data" document.

2 CPU

The Torrent CPU runs the MIPS-II standard ISA. Future Torrent implementations may adopt further extensions to the MIPS ISA standard.

The MIPS-II SYNC instruction forces all previous memory operations to complete before allowing further memory operations. SYNC instructions may be required on some Torrent implementations to synchronize memory references between the vector and scalar processors, or between separate vector memory pipelines.

3 System Control Coprocessor (CP0)

The system control coprocessor is the standard location for registers dealing with memory management and exception handling. The contents of CP0 are implementation dependent.

4 Floating Point Coprocessor (CP1)

Torrent employs the standard MIPS floating point coprocessor instruction set. If an implementation does not provide hardware floating point, these instructions cause a trap to software emulation.

5 Vector Unit (CP2)

The Torrent vector unit (VU) is implemented as coprocessor 2 for the MIPS CPU.

Vector registers

Figure 1 shows the registers contained within the vector coprocessor. The VU ISA allows up to 32 vector registers, `$vr0`–`$vr31`[1]

The first implementation, T0, provides only 16 vector registers, `$vr0`–`$vr15`. All elements of vector register `$vr0` are defined to hold the constant zero. Writes to `$vr0` are ignored, and reads of `$vr0` return 0.

Each vector element is 32b wide. The first implementation, T0, provides 32 elements per vector. Future implementations may provide longer vectors.

Control registers

Five VU control registers are defined in the coprocessor control register space. These are accessed using the standard `ctc2`/`cfc2` instructions.

An implementation will typically include additional VU control registers to handle exceptions. These are implementation dependent and will normally only be accessible from kernel mode.

VU Instructions

The VU extends the MIPS-II instruction set by adding coprocessor instructions that perform vector operations. The VU is a load/store vector-register architecture. Vector instructions are divided into 2 major groups: vector load/store operations, and vector arithmetic operations. Vector load/store operations move vectors between vector registers and memory, while vector arithmetic instructions operate on vectors in registers.

Each vector register holds a vector of 32b values. A single vector instruction specifies a sequence of operations, and may run for many cycles. The maximum length of a vector is limited by the implementation, but shorter vectors can be specified using the vector length register.

Scalar operands for vector-scalar operations are obtained from the CPU general purpose registers.

There are three varieties of vector load/stores: unit stride, arbitrary stride, and indexed. Unit stride load/stores can specify a post-increment for the scalar address register. Arbitrary stride load/stores transfer elements stored at addresses that form an arithmetic progression. Indexed vector load/stores (gather/scatter) allow indirect memory accesses to be vectorized.

Vector memory operations can transfer bytes, halfwords, and words. Bytes and halfwords are optionally sign-extended to 32b when loaded, and the least significant byte or halfword of a vector element is used for a byte or halfword store.

Vector 32b integer and 32b logical operations are defined. In addition, fixed point instructions are defined to support scaled fixed-point arithmetic. These instructions allow the multiple steps required for a scaled, rounded, fixed-point addition or multiplication to be performed within a single vector instruction.

Conditional vector operations are supported with vector conditional move instructions. A set of vector flag instructions allow a vector condition to be represented as a bit vector that can be read into a scalar register for further processing.

[1] The Torrent Architecture extends the MIPS register naming scheme. The vector registers are defined by the assembler as `$vr0`–`$vr31`, but are usually referred to by the aliases `vv0`–`vv31` in user code.

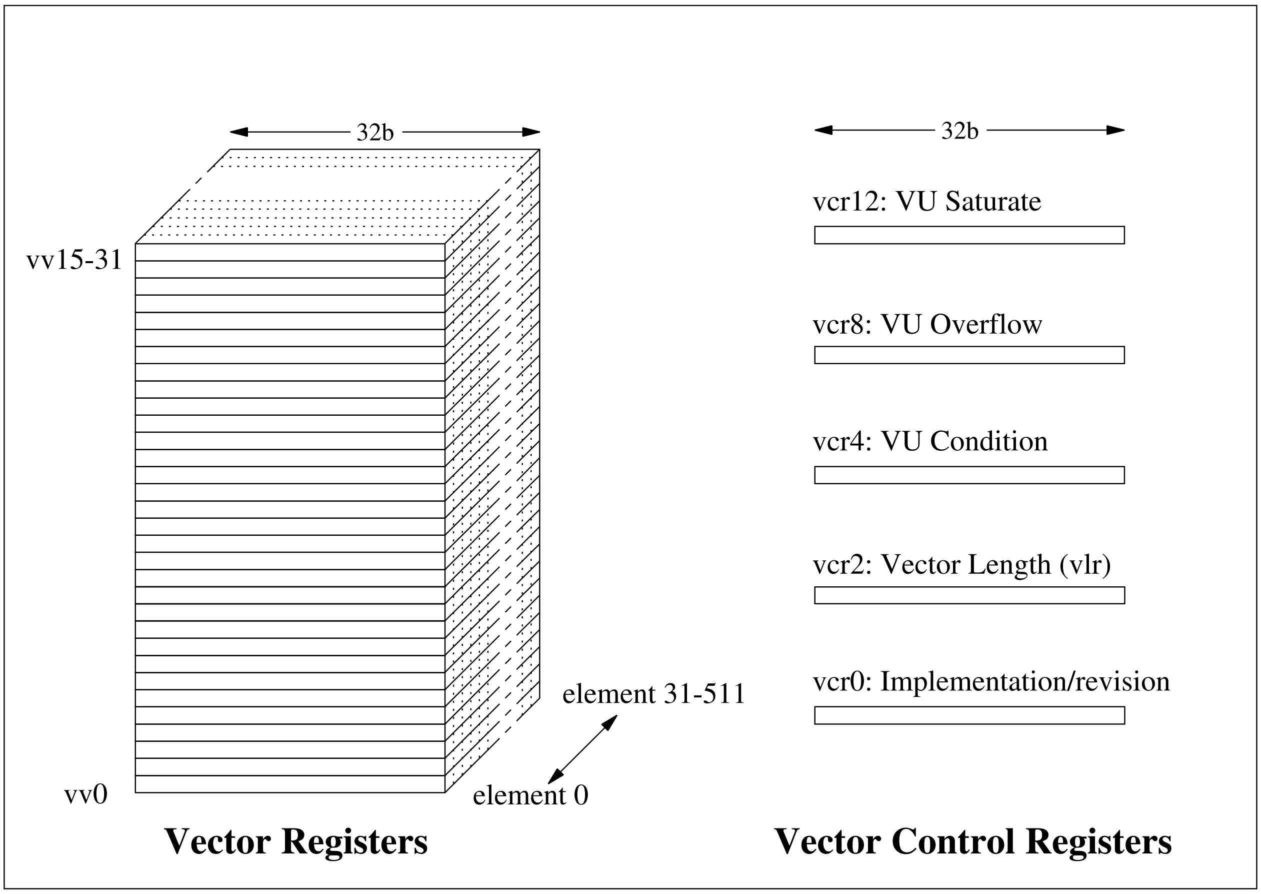


Figure 1: Vector unit registers.

5.1 Vector Unit Control Registers

The vector unit control registers are listed in Table 1.

Number	Register	Description
vcr0	`vrev`	Implementation/revision
vcr1	`vcount`	Free running counter
vcr2	`vlr`	Vector length
vcr4	`vcond`	Vector condition flags
vcr8	`vovf`	Vector overflow flags
vcr12	`vsat`	Vector saturation flags

Table 1: Vector unit control registers.

5.1.1 VU Implementation and Revision Number (VCR0)

31 16	15 8	7 0
0	Imp	Rev
16	8	8

Figure 2: VU Implementation and Revision Register Format

The `vrev` register is a 32b read only register that contains the implementation and revision number of the VU. These values can be used by configuration and diagnostic software.

The `vrev` register format is shown in Figure 2. Bits 15–8 define the implementation number, and bits 7–0 define the revision number. The implementation number can be used by user software to detect changes in instruction set or performance. The revision number identifies mask revisions of a given implementation.

Implementation field values are given in Table 2.

Imp. Number	Vector Unit
0	T0
1–255	*reserved*

Table 2: VU Implementation types.

5.1.2 Vector Count Register (VCR1)

31 … 0
`vcount`
32

Figure 3: Vector Count Register Format

The vector count register, `vcount`, is a 32b read only register formatted as shown in Figure 3.

The value in the `vcount` register is guaranteed to increase linearly with time, although the rate of increase is unspecified. When it reaches a maximum value of 0xffffffff, the count register will reset to 0 and continue incrementing. Its purpose is to allow relative comparison of small periods of elapsed time for performance analysis.

5.1.3 Vector Length Register (VCR2)

31 … 8	7 … 0
0	`vlr`
24	8

Figure 4: Vector Length Register Format

The length of a vector operation is specified in an 8b vector length register, `vlr`. If a vector instruction is issued when the value in `vlr` is 0, no operation is performed. If a vector instruction is issued when the value in `vlr` is greater than the implementation's maximum vector length, a vector operation exception is raised. Implementations provide at least 32 elements per vector.

Reads or writes of the vector length register do not affect vector instructions in progress.

5.1.4 VU Condition Register (VCR4)

31	30 … 2	1	0
vcond31	...	vcond1	vcond0
1	...	1	1

Figure 5: Vector Condition Register Format

The VU condition register, `vcond`, is a 32b read/write register formatted as shown in Figure 5.

The `vcond` register holds the VU condition flags, which reflect the result of the last conditional flag instruction. There is one flag bit corresponding to each vector element, with the least significant bit representing the condition for vector element zero. The register is only altered by conditional flag instructions (set less than, set less than unsigned, set equal) and writes to `vcond`.

Reads of `vcond` are interlocked and are guaranteed to return the most recent value. Writes to `vcond` are not affected by previously issued vector instructions which may still be executing.

Future implementations with greater than 32 elements per vector register may provide additional control registers to hold the extra conditional bits. Future implementations may also add further `vcond` registers to permit better scheduling of parallel conditional operations.

5.1.5 VU Overflow Register (VCR8)

31	30 … 2	1	0
vovf31	...	vovf1	vovf0
1	...	1	1

Figure 6: Vector Overflow Register Format

The VU overflow register, `vovf`, is a 32 bit read/write register formatted as shown in Figure 6.

The `vovf` register holds the VU overflow flags. The overflow flags are sticky bits which are set when any vector arithmetic operation causes an overflow. There is one flag bit corresponding to each vector element, with the least significant bit representing the overflow status for vector element zero. The register is only altered by overflowing arithmetic operations (add, sub) and writes to `vovf`.

Reads of `vovf` are interlocked and are guaranteed to return the most recent value. Writes to `vovf` are not affected by previously issued vector instructions which may still be executing.

Implementations that have greater than 32 elements per vector register have additional overflow registers to hold the extra overflow bits.

5.1.6 VU Saturate Register (VCR12)

31	30 2	1	0
`vsat31`	...	`vsat1`	`vsat0`
1	...	1	1

Figure 7: Vector Saturate Register Format

The VU saturate register, **`vsat`**, is a 32 bit read/write register formatted as shown in Figure 7.

The **`vsat`** register holds the VU saturate flags. The saturate flags are sticky bits which are set when any vector arithmetic operation causes a saturated result value. There is one flag bit corresponding to each vector element, with the least significant bit representing the saturation status for vector element zero. The register is only altered by vector fixed-point operations and writes to **`vsat`**.

Reads of **`vsat`** are interlocked and are guaranteed to return the most recent value. Writes to **`vsat`** are not affected by previously issued vector instructions which may still be executing.

Implementations that have greater than 32 elements per vector register have additional saturation registers to hold the extra saturation bits.

5.2 Instruction Overview

VU Instruction Classes

Instructions affecting the vector unit are divided into several classes:

- **Control Register** instructions that read and write VU coprocessor control registers.
- **Move** instructions that move data between vector registers, and between the CPU general registers and the vector registers.
- **Load/Store** instructions that move vectors of data between vector registers and memory.
- **Integer Arithmetic** instructions that provide integer arithmetic, shift, logical, compare and conditional operations on vector register contents.
- **Fixed-point** instructions that provide scaled and rounded fixed point arithmetic operations on vector register contents.

VU Instruction Formats

The VU control register read/write instructions use the standard MIPS coprocessor instruction encodings.

The move, load/store, integer, and fixed-point arithmetic instructions are encoded using the standard MIPS coprocessor operate instruction. These all use the base instruction format shown in Figure 8

The *format* field is encoded as shown in Table 3. The *format* field encodes the class of instruction and also the operand sources, either vector-vector or vector-scalar.

The *opers* field defines the order of operands for non-commutative operations. If *opers* is zero, the operands are vector/vector or vector/scalar. If *opers* is one, the operands are scalar/vector.

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	format 1*xxxx*	rt/vt	rd	opers	vw	vd
6	5	5	5	1	5	5

Figure 8: Vector unit base instruction format.

Format	Operation type	Operands
10000	Insert/extract	Vector-Vector
10001	Insert/extract	Vector-Scalar
10010	Memory	Vector-Vector
10011	Memory	Vector-Scalar
10100	Integer/logical	Vector-Vector
10101	Integer/logical	Vector-Scalar
10110	*reserved*	
10111	*reserved*	
11000	Add fixed	Vector-Vector
11001	Add fixed	Vector-Scalar
11010	Sub fixed	Vector-Vector
11011	Sub fixed	Vector-Scalar
11100	Multiply fixed	Vector-Vector
11101	Multiply fixed	Vector-Scalar
11110	*reserved*	
11111	*reserved*	

Table 3: *Format* field decoding.

5.3 VU Control Register Instructions

The vector control register instructions move values between the scalar CPU registers and the vector control registers. These operations use the standard MIPS coprocessor control register operations.

These operations are unpredictable if the coprocessor control register field is not one of the valid coprocessor control register numbers as listed in Table 1.

CFVU Move Control Word From VU

31 26	25 21	20 16	15 11	10 0
COP2 010010	CF 00010	rt	cs	00000000000
6	5	5	5	11

Format:

CFVU rt, cs

Description:

The contents of vector unit control register *cs* are copied into scalar register *rt*.

This operation is only defined when *cs* is a valid coprocessor control register.

Operation:

```
r[rt] = vcr[cs];
```

Exceptions:

Coprocessor unusable exception.

CTVU Move Control Word To VU

31 26	25 21	20 16	15 11	10 0
COP2 010010	CT 00110	rt	cs	00000000000
6	5	5	5	11

Format:

CTVU rt, cs

Description:

The contents of scalar register *rt* are copied into vector unit control register *cs*.

This operation is only defined when *cs* is a valid coprocessor control register.

Operation:

```
vcr[cs] = r[rt];
```

Exceptions:

Coprocessor unusable exception.

5.4 Vector Insert/Extract Instructions

The insert and extract instructions are used to form vectors from scalars, or to break vectors down into scalars or smaller vectors.

The extract vector instruction transfers elements from the end of one vector register to the start of another. A scalar register gives a start index into the source register. This instruction can be used to speed reduction operations.

The scalar insert/extract instructions transfer an element between the scalar register file and a vector register. A scalar register gives the index.

The insert/extract instruction encoding is shown in Table 4.

Format	*vw*	Opers	Mnemonic	Description
10000	01011	0	vext.v	Extract from vector into vector.
10001	10011	0	vins.s	Insert into vector from scalar.
10001	11011	0	vext.s	Extract from vector into scalar.

Table 4: Insert/extract instruction encoding.

VEXT.V Extract Vector To Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IXV 10000	vt	rd	VS 0	EXTV 01011	vd
6	5	5	5	1	5	5

Format:

vext.v vd, vt, rd

Description:

The `vlr` register is read to give, n, the number of elements to be moved. Starting from the index in scalar register *rd*, n elements from vector register *vt* are copied into the first n elements of vector register *vd*.

If the lower 8 bits of the sum of `vlr` and *rd* are greater than the maximum vector length, a vector operation exception is raised.

Operation:

```
for (i=0; i<vlr; i++)
   v[vd][i] = v[vt][r[rd]+i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

VINS.S Insert Into Vector From Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IXS 10001	rt	rd	VS 0	INSS 10011	vd
6	5	5	5	1	5	5

Format:

vins.s rt, vd, rd

Description:

The value of scalar register *rt* is copied to element *rd* of vector register *vd*.

If the lower eight bits of scalar register *rd* are greater than or equal to the maximum vector length, a vector operation exception is raised.

Operation:

```
v[vd][r[rd]] = r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

VEXT.S Extract From Vector To Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IXS 10001	rt	rd	VS 0	EXTS 11011	vd
6	5	5	5	1	5	5

Format:

vext.s rt, vd, rd

Description:

The value of element *rd* of vector register *vd* is copied to scalar register *rt*.

If the lower 8 bits of scalar register *rd* are greater than or equal to the maximum vector length, a vector operation exception is raised.

Operation:

```
r[rt] = v[vd][r[rd]];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

5.5 Vector Load/Store Instructions

Vector loads and stores transfer bytes, halfwords, and words between vector register elements and memory. Bytes and halfwords are sign-extended when loaded into vector elements. Stores always transfer the least significant bits of an element to memory. Addresses for the memory transfers are taken from the scalar registers.

There are three classes of vector loads and stores: unit-stride, arbitrary stride, and vector indexed. The unit-stride operations transfer vectors whose elements are held in contiguous locations in memory. The unit-stride operations allow a post-increment of the base address register. The arbitrary stride operations transfer vectors to or from memory at addresses that form an arithmetic progression. The vector indexed operations transfer vectors whose elements are located at offsets from a base address, with the offsets specified by the contents of an index vector.

Table 5 shows the encoding for vector load/store operations.

Format	*vw*	Opers	Mnemonic	Description
10010	0xxxx	0	*reserved*	
10010	10000	0	lbx.v	Load signed byte vector indexed.
10010	10001	0	lhx.v	Load signed halfword vector indexed.
10010	10010	0	*reserved*	
10010	10011	0	lwx.v	Load word vector indexed.
10010	10100	0	lbux.v	Load unsigned byte vector indexed.
10010	10101	0	lhux.v	Load unsigned halfword vector indexed.
10010	1011x	0	*reserved*	
10010	11000	0	sbx.v	Store byte vector indexed.
10010	11001	0	shx.v	Store halfword vector indexed.
10010	11010	0	*reserved*	
10010	11011	0	swx.v	Store word vector indexed.
10010	111xx	0	*reserved*	
10011	00000	0	lbai.v	Load unit-stride signed byte vector with auto-increment.
10011	00001	0	lhai.v	Load unit-stride signed halfword vector with auto-increment.
10011	00010	0	*reserved*	
10011	00011	0	lwai.v	Load unit-stride word vector with auto-increment.
10011	00100	0	lbuai.v	Load unit-stride unsigned byte vector with auto-increment.
10011	00101	0	lhuai.v	Load unit-stride unsigned halfword vector with auto-increment.
10011	0011x	0	*reserved*	
10011	01000	0	sbai.v	Store unit-stride byte vector with auto-increment.
10011	01001	0	shai.v	Store unit-stride halfword vector with auto-increment.
10011	01010	0	*reserved*	
10011	01011	0	swai.v	Store unit-stride word vector with auto-increment.
10011	011xx	0	*reserved*	
10011	10000	0	lbst.v	Load signed byte vector with stride.
10011	10001	0	lhst.v	Load signed halfword vector with stride.
10011	10010	0	*reserved*	
10011	10011	0	lwst.v	Load word vector with stride.
10011	10100	0	lbust.v	Load unsigned byte vector with stride.
10011	10101	0	lhust.v	Load unsigned halfword vector with stride.
10011	1011x	0	*reserved*	
10011	11000	0	sbst.v	Store byte vector with stride.
10011	11001	0	shst.v	Store halfword vector with stride.
10011	11010	0	*reserved*	
10011	11011	0	swst.v	Store word vector with stride.
10011	111xx	0	*reserved*	
1001x	xxxxx	1	*reserved*	

Table 5: Vector load/store instruction encoding.

LxAI.V Load Auto-Increment Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LBAI 00000	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LHAI 00001	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LWAI 00011	vd
6	5	5	5	1	5	5

Format:

lbai.v vd, rd, rt
lhai.v vd, rd, rt
lwai.v vd, rd, rt

Description:

The `vlr` register is read to give, n, the number of elements to be loaded. Starting from the base address in scalar register *rd*, n contiguous elements are loaded from memory, sign-extended to 32b (if necessary), and placed in the first n consecutive elements of the vector register *vd*. The value of *rd* is post-incremented by the value of *rt*. This post-increment is treated as unsigned addition and does not generate an overflow. The result of the instruction is undefined if *rd* is the same as *rt*.

A vector address exception occurs if the instruction loads halfwords and the least significant bit of the *rd* register is non-zero. A vector address exception occurs if the instruction loads words and either of the two least significant bits of the *rd* register are non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vd][i] = extend(m[r[rd]+i*elsize]);
r[rd] += r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

LxUAI.V Load Unsigned Auto-Increment Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LBUAI 00100	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LHUAI 00101	vd
6	5	5	5	1	5	5

Format:

lbuai.v vd, rd, rt
lhuai.v vd, rd, rt

Description:

The `vlr` register is read to give, n, the number of elements to be loaded. Starting from the base address in scalar register *rd*, n contiguous elements are loaded from memory, zero extended to 32b, and placed in the first n consecutive elements of the vector register *vd*. The value of *rd* is post-incremented by the value of *rt*. This post-increment is treated as unsigned addition and does not generate an overflow. The result of the instruction is undefined if *rd* is the same as *rt*.

A vector address exception occurs if the instruction loads halfwords and the least significant bit of the *rd* register is non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vd][i] = m[r[rd]+i*elsize];
r[rd] += r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

LxX.V Load Indexed Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	LBX 10000	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	LHX 10001	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	LWX 10011	vd
6	5	5	5	1	5	5

Format:

lbx.v vd, rd, vt
lhx.v vd, rd, vt
lwx.v vd, rd, vt

Description:

This is a gather operation. The `vlr` register is read to give, n, the number of elements to be loaded. The scalar register *rd* is read to give the base address. The first n elements of *vt* are then added to *rd* to give n effective addresses. The vector of effective addresses is used to load n elements from memory which are then sign-extended to 32b, and placed into the first n elements of *vd*.

A vector address exception occurs if the instruction loads halfwords and the least significant bit of any effective address is zero. A vector address exception occurs if the instruction loads words and either of the two least significant bits of any effective address are non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vd][i] = extend(m[r[rd]+v[vt][i]]);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

LxUX.V Load Unsigned Indexed Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	LBUX 10100	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	LHUX 10101	vd
6	5	5	5	1	5	5

Format:

lbux.v vd, rd, vt
lhux.v vd, rd, vt

Description:

This is a gather operation. The `vlr` register is read to give, n, the number of elements to be loaded. The scalar register *rd* is read to give the base address. The first n elements of *vt* are then added to *rd* to give n effective addresses. The vector of effective addresses is used to load n elements from memory which are zero extended to 32b, and placed into the first n elements of *vd*.

A vector address exception occurs if the instruction loads halfwords and the least significant bit of any effective address is zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vd][i] = m[r[rd]+v[vt][i]];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

LxST.V **Load Strided Vector**

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LBST 10000	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LHST 10001	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LWST 10011	vd
6	5	5	5	1	5	5

Format:

lbst.v vd, rd, rt
lhst.v vd, rd, rt
lwst.v vd, rd, rt

Description:

The `vlr` register is read to give, n, the number of elements to be loaded. The scalar register *rt* is read to give the byte stride of the accesses. The first operand is loaded from the address given in *rd*, sign-extended to the vector element width (if necessary), and placed in the first element of *vd*. The k^{th} element of *vd* is loaded from address

$$rd + rt \times (k-1)$$

A vector address exception occurs if the instruction loads halfwords and the least significant bit of any effective address is non-zero. A vector address exception occurs if the instruction loads words and either of the two least significant bits of any effective address is non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vd][i] = extend(m[r[rd]+r[rt]*i]);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

LxUST.V Load Unsigned Strided Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LBUST 10100	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	LHUST 10101	vd
6	5	5	5	1	5	5

Format:

lbust.v vd, rd, rt
lhust.v vd, rd, rt

Description:

The `vlr` register is read to give, n, the number of elements to be loaded. The scalar register *rt* is read to give the byte stride of the accesses. The first operand is loaded from the address given in *rd*, zero extended to the vector element width (if necessary), and placed in the first element of *vd*. The k^{th} element of *vd* is loaded from address

$$rd + rt \times (k-1)$$

A vector address error exception occurs if the instruction loads halfwords and the least significant bit of any effective address is non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vd][i] = m[r[rd]+r[rt]*i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

SxAI.V Store Auto-Increment Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	SBAI 01000	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	SHAI 01001	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	SWAI 01011	vd
6	5	5	5	1	5	5

Format:

sbai.v vd, rd, rt
shai.v vd, rd, rt
swai.v vd, rd, rt

Description:

The `vlr` register is read to give, n, the number of elements to be stored. The first n consecutive elements of the vector register *vd* are stored in consecutive memory locations starting at the base address in scalar register *rd*. The *rd* register is post-incremented by the contents of the *rt* register. This post-increment is treated as unsigned addition and does not generate an overflow. The result of the instruction is undefined if *rd* is the same as *rt*. Implementations do not guarantee the order in which vector elements are written within a single vector store instruction.

A vector address exception occurs if the instruction stores halfwords and the least significant bit of the *rd* register is non-zero. A vector address exception occurs if the instruction stores words and either of the two least significant bits of the *rd* register is non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   m[r[rd]+i*elsize] = v[vd][i];
r[rd] += r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

SxX.V Store Indexed Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	SBX 11000	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	SHX 11001	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMV 10010	vt	rd	VS 0	SWX 11011	vd
6	5	5	5	1	5	5

Format:

sbx.v vd, rd, vt
shx.v vd, rd, vt
swx.v vd, rd, vt

Description:

This is a scatter operation. The `vlr` register is read to give, n, the number of elements to be stored. The scalar register *rd* is read to give the base address. The first n elements of *vt* are then added to *rd* to give n effective addresses. The first n elements of *vd* are written to memory using the vector of effective addresses. Implementations do not guarantee the order in which individual vector elements are written within a single vector store instruction.

A vector address exception occurs if the instruction stores halfwords and the least significant bit any effective address is non-zero. A vector address exception occurs if the instruction stores words and either of the two least significant bits of any effective address is non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   m[r[rd]+v[vt][i]] = v[vd][i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

SxST.V — Store Strided Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	SBST 11000	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	SHST 11001	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	MEMS 10011	rt	rd	VS 0	SWST 11011	vd
6	5	5	5	1	5	5

Format:

sbst.v vd, rd, rt
shst.v vd, rd, rt
swst.v vd, rd, rt

Description:

The `vlr` register is read to give, n, the number of elements to be stored. The scalar register *rt* is read to give the byte stride of the accesses. The first element of *vd* is stored to the address given in the scalar register *rd*. The k^{th} element of *vd* is stored at address

$$rd + rt \times (k-1)$$

Implementations do not guarantee the order in which vector elements are written within a single vector store instruction.

A vector address exception occurs if the instruction stores halfwords and the least significant bit of any effective address is non-zero. A vector address exception occurs if the instruction stores words and either of the two least significant bits of any effective address is non-zero. A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   m[r[rd]+r[rt]*i] = v[vd][i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.
Vector address exception.

5.6 Vector Integer ALU Operations

All vector integer and logical computational instructions are available in both vector-vector and vector-scalar forms. The *rd* field is used to encode the integer function.

The vector-vector instructions specify a vector of binary operations, with the first operand taken from vector register *vd*, the second operand from vector register *vt*, and the result placed in vector register *vw*. Vector-scalar instructions specify a vector of binary operations, with the first operand taken from vector register *vd*, the second operand from scalar register *rt*, and the result placed in vector register *vd*.

A few non-commutative operations have scalar-vector forms where the first operand is *rt* and the second operand is *vs*.

Table 6 shows the encoding of the available integer computational operations.

Vector arithmetic and logical instructions

The vector unit implements numerous arithmetic and logical operations, including variable displacement shifts. Signed and unsigned addition and subtraction are provided — the fixed point instructions (see Section 5.7) can be used to perform signed and unsigned 16 bit multiplication.

An overflow status register, `vovf`, is updated by integer arithmetic operations and can be accessed in coprocessor 2. One "sticky" overflow bit is provided for each vector element. Overflowing operations set these bits, but they can only be cleared by explicit writes to the overflow register.

Vector conditional instructions

To compare vectors, there are conditional set instructions "set less than" and "set equal". These instructions produce a vector of boolean results, and in the set less than case the comparison can be signed or unsigned. Both are available in vector-vector and vector-scalar forms, with set less than also having a scalar-vector form.

To compare vectors and produce results for manipulation in the scalar unit, there are "flag less than" and "flag equal" instructions. The vector unit maintains a condition flag register, with one condition bit for each vector register element. This register is updated by the flag instructions, and is accessed as a coprocessor 2 control register. To perform conditional branches on vector operations, this register is copied to a scalar CPU register where any MIPS-II conditional branch can be used. Implementations with greater than 32 elements per vector register provide additional condition flag registers.

The vector unit implements vector conditional moves, in both vector-vector and vector-scalar forms. The first operand is compared against zero; if the comparison succeeds, the destination element is updated with the second operand, otherwise the destination element is unaffected. Note that a scalar-vector form is not required, as a scalar condition can always be replaced with a branch in the CPU.

The conditional moves can be made unconditional if the condition vector register is `$vr0`. In this way, scalar register values can be broadcast into a vector register, and vectors of values can be copied between vector registers.

Format	*rd*	Opers	Mnemonic	Description
1010v	00000	w	sub.*yy*	Subtract signed (flag overflow).
1010v	00001	w	subu.*yy*	Subtract unsigned (no overflow).
1010x	00010	x		*reserved*
1010x	00011	x		*reserved*
1010v	00100	w	flt.*yy*	Flag less than (update condition).
1010v	00101	w	fltu.*yy*	Flag less than unsigned (update condition).
1010v	00110	0	feq.*yy*	Flag equal (update condition).
1010x	00111	x		*reserved*
1010v	01000	w	sllv.*yy*	Shift left logical variable.
1010v	01001	w	srlv.*yy*	Shift right logical variable.
1010x	01010	x		*reserved*
1010v	01011	w	srav.*yy*	Shift right arithmetic variable.
1010v	01100	w	slt.*yy*	Set less than.
1010v	01101	w	sltu.*yy*	Set less than unsigned.
1010v	01110	0	seq.*yy*	Set equal.
1010x	01111	x		*reserved*
1010v	10000	0	add.*yy*	Add signed (flag overflow).
1010v	10001	0	addu.*yy*	Add unsigned (no overflow).
1010x	10010	x		*reserved*
1010x	10011	x		*reserved*
1010v	10100	0	and.*yy*	Bitwise logical AND.
1010v	10101	0	or.*yy*	Bitwise logical OR.
1010v	10110	0	xor.*yy*	Bitwise logical XOR.
1010v	10111	0	nor.*yy*	Bitwise logical NOR.
1010x	11000	x		*reserved*
1010v	11001	0	cmvnez.*yy*	Conditional move not equal zero.
1010v	11010	0	cmvgez.*yy*	Conditional move greater than or equal zero.
1010v	11011	0	cmvlez.*yy*	Conditional move less than or equal zero.
1010x	11100	x		*reserved*
1010v	11101	0	cmveqz.*yy*	Conditional move equal zero.
1010v	11110	0	cmvltz.*yy*	Conditional move less than zero.
1010v	11111	0	cmvgtz.*yy*	Conditional move greater than zero.

Note:
When format is 11110 and opers is 0, *yy* is “vv”
When format is 11111 and opers is 0, *yy* is “vs”
When format is 11111 and opers is 1, *yy* is “sv”
All other encodings are reserved.

Table 6: Field encoding for integer register-register instructions.

AND.VV — And Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	AND 10100	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

and.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vt* are combined with first n elements of vector register *vd* in a bitwise logical AND operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] and v[vt][i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

AND.VS — And Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	AND 10100	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

and.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vd* are combined with scalar register *rt* in a bitwise logical AND operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] and r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

OR.VV Or Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	OR 10101	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

or.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vd* are combined with first n elements of vector register *vt* in a bitwise logical OR operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] or v[vt][i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

OR.VS Or Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	OR 10101	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

or.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vd* are combined with scalar register *rt* in a bitwise logical OR operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] or r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

XOR.VV — Xor Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	XOR 10110	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

xor.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vd* are combined with first n elements of vector register *vt* in a bitwise logical XOR operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] xor v[vt][i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

XOR.VS — Xor Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	XOR 10110	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

xor.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vd* are combined with scalar register *rt* in a bitwise logical XOR operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] xor r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

NOR.VV — Nor Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	NOR 10111	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

nor.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vd* are combined with first n elements of vector register *vt* in a bitwise logical NOR operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Note: the operation "nor.vv vw, vd, `$vr0`" performs a vector bitwise NOT operation.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] nor v[vt][i];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

NOR.VS — Nor Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	NOR 10111	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

nor.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to compute. The first n elements of vector register *vt* are combined with scalar register *rt* in a bitwise logical NOR operation. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] nor r[rt];
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

ADD.VV — Add Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	ADD 10000	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

add.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be added. The first n elements of vector register *vt* are added to the first n elements of vector register *vd* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as signed integers. The appropriate bit in `vovf` is set for any result that overflows.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] + v[vt][i];
   if (overflow_on_add(v[vd][i],v[vt][i]))
      vcr[VOVF] |= (1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

ADD.VS — Add Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	ADD 10000	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

add.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be added. The first n elements of vector register *vd* are added to the scalar register *rt* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as signed integers. The appropriate bit of `vovf` is set for any result that overflows.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] + r[rt];
   if (overflow_on_add(v[vd][i],r[rt]))
      vcr[VOVF] |= (1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

ADDU.VV Add Unsigned Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	ADDU 10001	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

addu.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be added. The first n elements of vector register *vt* are added to the first n elements of vector register *vd* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as unsigned integers. Overflows are ignored.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] + v[vt][i];
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

ADDU.VS Add Unsigned Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	ADDU 10001	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

addu.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be added. The first n elements of vector register *vd* are added to the scalar register *rt* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as unsigned integers. Overflows are ignored.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] + r[rt];
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SUB.VV — Subtract Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SUB 00000	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

sub.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be subtracted. The first n elements of vector register *vt* are subtracted from the first n elements of vector register *vd* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as signed integers. The appropriate bit of `vovf` is set for any result that overflows.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Note: the operation "sub.vv vw, `$vr0`, vt" performs a vector negate operation.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] - v[vt][i];
   if (overflow_on_sub(v[vd][i],v[vt][i]))
      vcr[VOVF] |= (1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SUB.VS — Subtract Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SUB 00000	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

sub.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be subtracted. The scalar register *rt* is subtracted from the first n elements of vector register *vd* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as signed integers. The appropriate bit of `vovf` is set for any result that overflows.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] - r[rt];
   if (overflow_on_sub(v[vd][i],r[rt]))
      vcr[VOVF] |= (1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SUB.SV — Subtract Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SUB 00000	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

sub.sv vw, rt, vd

Description:

The vector register **vlr** is read to give *n* the number of elements to be subtracted. The first *n* elements of vector register *vd* are subtracted from the scalar register *rt* and the results are placed in the first *n* elements of vector register *vw*.

The input elements are treated as signed integers. The appropriate bit of **vovf** is set for any result that overflows.

A vector operation exception is raised if **vlr** is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = r[rt] - v[vd][i];
   if (overflow_on_sub(r[rt],v[vd][i]))
      vcr[VOVF] |= (1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SUBU.VV — Subtract Unsigned Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SUBU 00001	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

subu.vv vw, vd, vt

Description:

The vector register **vlr** is read to give *n* the number of elements to be subtracted. The first *n* elements of vector register *vt* are subtracted from the first *n* elements of vector register *vd* and the results are placed in the first *n* elements of vector register *vw*.

The input elements are treated as unsigned integers. Overflows are ignored.

A vector operation exception is raised if **vlr** is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] - v[vt][i];
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SUBU.VS **Subtract Unsigned Vector-Scalar**

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SUBU 00001	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

subu.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be subtracted. The scalar register *rt* is subtracted from the first n elements of vector register *vd* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as unsigned integers. Overflows are ignored.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = v[vd][i] - r[rt];
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SUBU.SV **Subtract Unsigned Scalar-Vector**

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SUBU 00001	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

subu.sv vw, rt, vd

Description:

The vector register `vlr` is read to give n the number of elements to be subtracted. The first n elements of vector register *vd* are subtracted from the scalar register *rt* and the results are placed in the first n elements of vector register *vw*.

The input elements are treated as unsigned integers. Overflows are ignored.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   v[vw][i] = r[rt] - v[vd][i];
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLLV.VV Shift Left Logical Variable Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SLLV 01000	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

sllv.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to shift. The first n elements of vector register *vd* are shifted left by the number of bits given in the least significant 5 bits of the first n elements of vector register *vt*. Zeros are inserted into the low order bits. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] << (v[vt][i] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLLV.VS Shift Left Logical Variable Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SLLV 01000	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

sllv.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to shift. The first n elements of vector register *vd* are shifted left by the number of bits given in the least significant 5 bits of scalar register *rt*. Zeros are inserted into the low order bits. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] << (r[rt] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLLV.SV Shift Left Logical Variable Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SLLV 01000	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

sllv.sv vw, rt, vd

Description:

The vector register `vlr` is read to give n the number of elements to shift. The scalar register *rt* is shifted left by the number of bits given in the least significant 5 bits of the first n elements of vector register *vd*. Zeros are inserted into the low order bits. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = r[rt] << (v[vd][i] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SRAV.VV Shift Right Arithmetic Variable Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SRAV 01011	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

srav.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to shift. The first n elements of vector register *vd* are shifted right by the number of bits given in the least significant 5 bits of the first n elements of vector register *vt*. The high order bits are sign-extended. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
  v[vw][i] = v[vd][i] shra (v[vt][i] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SRAV.VS Shift Right Arithmetic Variable Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SRAV 01011	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

srav.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to shift. The first n elements of vector register *vd* are shifted right by the number of bits given in the least significant 5 bits of scalar register *rt*. The high order bits are sign-extended. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
  v[vw][i] = v[vd][i] shra (r[rt] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SRAV.SV Shift Right Arithmetic Variable Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SRAV 01011	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

srav.sv vw, rt, vd

Description:

The vector register `vlr` is read to give n the number of elements to shift. The scalar register *rt* is shifted right by the number of bits given in the least significant 5 bits of the first n elements of vector register *vd*. The high order bits are sign-extended. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
  v[vw][i] = r[rt] shra (v[vd][i] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SRLV.VV Shift Right Logical Variable Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SRLV 01001	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

srlv.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to shift. The first n elements of vector register *vd* are shifted right by the number of bits given in the least significant 5 bits of the first n elements of vector register *vt*. Zeros are inserted into the high order bits. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] >> (v[vt][i] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SRLV.VS Shift Right Logical Variable Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SRLV 01001	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

srlv.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to shift. The first n elements of vector register *vd* are shifted right by the number of bits given in the least significant 5 bits of scalar register *rt*. Zeros are inserted into the high order bits. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = v[vd][i] >> (r[rt] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SRLV.SV Shift Right Logical Variable Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SRLV 01001	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

srlv.sv vw, rt, vd

Description:

The vector register `vlr` is read to give n the number of elements to shift. The scalar register *rt* is shifted right by the number of bits given in the least significant 5 bits of the first n elements of vector register *vd*. Zeros are inserted into the high order bits. The results are placed in the first n elements of vector register *vw*.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
   v[vw][i] = r[rt] >> (v[vd][i] & 0x1f);
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLT.VV Set Less Than Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SLT 01100	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

slt.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the first n elements of vector register *vt*. If an element of *vd* is less than an element of *vt*, the corresponding element in *vw* is set to 1, else it is set to 0. The elements are considered as signed integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < v[vt][i])
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLT.VS Set Less Than Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SLT 01100	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

slt.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the scalar register *rt*. If an element of *vd* is less than *rt*, the corresponding element in *vw* is set to 1, else it is set to 0. The elements are considered as signed integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < rt)
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLT.SV Set Less Than Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SLT 01100	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

slt.sv vw, rt, vd

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the scalar register *rt*. If *rt* is less than an element of *vd*, the corresponding element in *vw* is set to 1, else it is set to 0. The elements are considered as signed 32-bit integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (r[rt] < v[vd][i])
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLTU.VV — Set Less Than Unsigned Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SLTU 01101	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

sltu.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the first n elements of vector register *vt*. If an element of *vd* is less than an element of *vt*, the corresponding element in *vw* is set to 1, else it is set to 0. The elements are considered as unsigned integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < v[vt][i])
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLTU.VS — Set Less Than Unsigned Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SLTU 01101	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

sltu.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the scalar register *rt*. If an element of *vd* is less than *rt*, the corresponding element in *vw* is set to 1, else it is set to 0. The elements are considered as unsigned integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < rt)
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SLTU.SV — Set Less Than Unsigned Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SLTU 01101	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

sltu.sv vw, rt, vd

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the scalar register *rt*. If *rt* is less than an element of *vd*, the corresponding element in *vw* is set to 1, else it is set to 0. The elements are considered as unsigned integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (r[rt] < v[vd][i])
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SEQ.VV — Set Equal Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	SEQ 01110	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

seq.vv vw, vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the first n elements of vector register *vt*. If an element of *vd* is equal to an element of *vt*, the corresponding element in *vw* is set to 1, else it is set to 0.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] == v[vt][i])
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

SEQ.VS Set Equal Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	SEQ 01110	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

seq.vs vw, vd, rt

Description:

The vector register `vlr` is read to give *n* the number of elements to be compared. The first *n* elements of vector register *vd* are compared with the scalar register *rt*. If an element of *vd* is equal to *rt*, the corresponding element in *vw* is set to 1, else it is set to 0.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] == rt)
      v[vw][i] = 1;
   else
      v[vw][i] = 0;
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FLT.VV Flag Less Than Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	FLT 00100	VS 0	00000	vd
6	5	5	5	1	5	5

Format:

flt.vv vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the first n elements of vector register *vt*. If an element of *vd* is less than an element of *vt*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0. The elements are considered as signed integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < v[vt][i])
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FLT.VS Flag Less Than Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	FLT 00100	VS 0	00000	vd
6	5	5	5	1	5	5

Format:

flt.vs vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the scalar register *rt*. If an element of *vd* is less than *rt*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0. The elements are considered as signed integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < rt)
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FLT.SV — Flag Less Than Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	FLT 00100	SV 1	00000	vd
6	5	5	5	1	5	5

Format:

flt.sv rt, vd

Description:

The vector register `vlr` is read to give *n* the number of elements to be compared. The first *n* elements of vector register *vd* are compared with the scalar register *rt*. If *rt* is less than an element of *vd*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0. The elements are considered as signed 32-bit integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (r[rt] < v[vd][i])
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FLTU.VV — Flag Less Than Unsigned Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	FLTU 00101	VS 0	00000	vd
6	5	5	5	1	5	5

Format:

fltu.vv vd, vt

Description:

The vector register `vlr` is read to give *n* the number of elements to be compared. The first *n* elements of vector register *vd* are compared with the first *n* elements of vector register *vt*. If an element of *vd* is less than an element of *vt*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0. The elements are considered as unsigned integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < v[vt][i])
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FLTU.VS — Flag Less Than Unsigned Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2	IVS	rt	FLTU	VS		vd
010010	10101		00101	0	00000	
6	5	5	5	1	5	5

Format:

fltu.vs vd, rt

Description:

The vector register `vlr` is read to give *n* the number of elements to be compared. The first *n* elements of vector register *vd* are compared with the scalar register *rt*. If an element of *vd* is less than *rt*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0. The elements are considered as unsigned integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] < rt)
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FLTU.SV — Flag Less Than Unsigned Scalar-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2	IVS	rt	FLTU	SV		vd
010010	10101		00101	1	00000	
6	5	5	5	1	5	5

Format:

fltu.sv rt, vd

Description:

The vector register `vlr` is read to give *n* the number of elements to be compared. The first *n* elements of vector register *vd* are compared with the scalar register *rt*. If *rt* is less than an element of *vd*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0. The elements are considered as unsigned integers.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (r[rt] < v[vd][i])
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FEQ.VV — Flag Equal Vector-Vector

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVV 10100	vt	FEQ 00110	VS 0	00000	vd
6	5	5	5	1	5	5

Format:

feq.vv vd, vt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the first n elements of vector register *vt*. If an element of *vd* is equal to an element of *vt*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] == v[vt][i])
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FEQ.VS — Set Equal Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	FEQ 00110	VS 0	00000	vd
6	5	5	5	1	5	5

Format:

feq.vs vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be compared. The first n elements of vector register *vd* are compared with the scalar register *rt*. If an element of *vd* is equal to *rt*, the corresponding bit in the vector unit condition code is set to 1, else it is set to 0.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] == rt)
      vcr[VCOND] |= (1<<i);
   else
      vcr[VCOND] &= ~(1<<i);
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

CMVccc.VV — Conditional Move Vector-Vector

31–26	25–21	20–16	15–11	10	9–5	4–0
COP2 010010	IVV 10100	vt	CMVNEZ 11001	VS 0	vw	vd
6	5	5	5	1	5	5

31–26	25–21	20–16	15–11	10	9–5	4–0
COP2 010010	IVV 10100	vt	CMVGEZ 11010	VS 0	vw	vd
6	5	5	5	1	5	5

31–26	25–21	20–16	15–11	10	9–5	4–0
COP2 010010	IVV 10100	vt	CMVLEZ 11011	VS 0	vw	vd
6	5	5	5	1	5	5

31–26	25–21	20–16	15–11	10	9–5	4–0
COP2 010010	IVV 10100	vt	CMVEQZ 11101	VS 0	vw	vd
6	5	5	5	1	5	5

31–26	25–21	20–16	15–11	10	9–5	4–0
COP2 010010	IVV 10100	vt	CMVLTZ 11110	VS 0	vw	vd
6	5	5	5	1	5	5

31–26	25–21	20–16	15–11	10	9–5	4–0
COP2 010010	IVV 10100	vt	CMVGTZ 11111	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

cmvnez.vv vw, vd, vt
cmvgez.vv vw, vd, vt
cmvlez.vv vw, vd, vt
cmveqz.vv vw, vd, vt
cmvltz.vv vw, vd, vt
cmvgtz.vv vw, vd, vt

Description:

The vector register `vlr` is read to give *n* the number of elements to be moved. The first *n* elements of vector register *vd* are read. For all those elements that satisfy the comparison with zero, the corresponding element of vector *vw* is updated with the corresponding element of *vt*. All other elements of *vw* are unchanged.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Note: the operation "cmveqz.vv vw, `$vr0`, vt" performs an unconditional vector move.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] condop 0)
   {
      v[vw][i] = v[vt][i];
   }
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

CMVccc.VS — Conditional Move Vector-Scalar

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	CMVNEZ 11001	VS 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	CMVGEZ 11010	VS 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	CMVLEZ 11011	VS 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	CMVEQZ 11101	VS 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	CMVLTZ 11110	VS 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	IVS 10101	rt	CMVGTZ 11111	VS 0	vw	vd
6	5	5	5	1	5	5

Format:

cmvnez.vs vw, vd, rt
cmvgez.vs vw, vd, rt
cmvlez.vs vw, vd, rt
cmveqz.vs vw, vd, rt
cmvltz.vs vw, vd, rt
cmvgtz.vs vw, vd, rt

Description:

The vector register `vlr` is read to give n the number of elements to be moved. The first n elements of vector register *vd* are read. For all those elements that satisfy the comparison with zero, the corresponding element of vector *vw* is updated with the value of scalar register *rt*. All other elements of *vw* are unchanged.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Note: the operation "cmveqz.vs vw, `$vr0`, vt" performs an unconditional scalar to vector move.

Operation:

```
for (i=0; i<vlr; i++)
{
   if (v[vd][i] condop 0)
   {
      v[vw][i] = r[rt];
   }
}
```

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

5.7 Vector Fixed-Point Arithmetic Operations

The add, subtract and multiply fixed-point instructions are primarily used to implement scaled, rounded, and clipped fixed-point arithmetic. The scaling, rounding, and clipping information is supplied by a normal scalar register specified by the instruction. This register is termed the "configuration register". The fixed point instructions can also perform some unsigned arithmetic with an appropriate value in the configuration register. The contents of the configuration register are interpreted as shown in Figure 9.

31 12	11	10	9	8 7	6 5	4 0
0	`shrst`	`shrnr`	`shlnr`	`clipam`	0	`sham`
20	1	1	1	2	2	5

Figure 9: Fixed Point Configuration Register Format.

A short description of each configuration register field is given in Table 7. The use of these fields is explained in more detail in the description of the individual instructions that use them (fxadd.*yy*, fxsub.*yy*, and fxmul.*yy*). Note that unused fields may provide additional operations in specific Torrent implementations. For execution of the operations described here, all unused bits must be set to 0.

Name	Description
`clipam`	Clip amount.
`shrst`	Jam sticky bit or round to even.
`shrnr`	Don't alter right shift output.
`shlnr`	Don't add in round bit.
`sham`	Shift amount.

Table 7: Fixed point register fields.

The clip amount field is interpreted as shown in Table 8.

The three bits `shrst`, `shrnr` and `shlnr` effectively control the rounding mode used for fixed point operations. Table 9 describes some useful combinations of these bits.

`clipam`	Clip output
00	Clip to 32b.
01	Clip to 8b.
10	Clip to 16b.
11	Unpredictable.

Table 8: Clip amount values.

`shrst/shrnr/shlnr`	Rounding mode
000	Round to even.
X11	Truncation.
X10	Round up.
101	Zero bias jamming.

Table 9: Fixed point rounding modes.

The encoding has been designed such that the most common operation of performing a scaled, round-to-even operation with a clip to 32b has zeros in all bits other than the shift field.

A saturation status register, `vsat`, is accessible in coprocessor 2. It is updated by fixed point arithmetic operations, with one "sticky" saturation bit provided for each vector element. Vector elements which have modified results due to clipping will set the appropriated bit in the saturation register. Bits in the register can only be cleared by explicit writes.

FXADD.yy **Fixed Point Add**

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXAVV 11000	vt	rd	SV 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXAVS 11001	rt	rd	SV 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXAVS 11001	rt	rd	VS 1	vw	vd
6	5	5	5	1	5	5

Format:

fxadd.vv vw, vd, vt, rd
fxadd.vs vw, vd, rt, rd
fxadd.sv vw, rt, vd, rd

Description:

This command is available in vector-vector, vector-scalar, and scalar-vector forms. The vector register `vlr` is read to give *n* the number of elements to be operated upon. The following operations are performed under the control of the scalar configuration register *rd*. The configuration register is formatted as shown in Figure 9.

First stage — source operand mux. For a vector-vector operation, the A input to the pipeline comes from the *vd* vector register, and the B input to the pipeline comes from the *vt* vector register. For a vector-scalar operation, the A input to the pipeline comes from the *vd* vector register and the B input to the pipeline comes from the *rt* scalar register. For a scalar-vector operation, the A input to the pipeline comes from the *rt* scalar register and the B input to the pipeline comes from the *vd* vector register.

Second stage — left shifter. The A input is shifted left by the amount given in the `sham` field of register *rd*. If the `shlnr` bit of *rd* is set, zeros are shifted in from the right. If the `shlnr` bit is clear, a 1 is shifted in from the 1/2 LSB position with zeros following. Only the low 32b of the result are kept, and no overflow checking is performed.

Third stage — adder. The B input is added to the shifted A input in a 33b adder. The extra bit on the adder ensures there can be no overflow at this stage.

Fourth stage — right shifter. The 33b adder result is shifted right by the number of bits given in the `sham` field of *rd*. Sign bits are shifted in to the high order bits. The bits which are shifted off to the right are OR-ed together to form a sticky bit. If the `shrnr` bit in *rd* is clear, then the right shifted output is altered depending on the sticky bit. If both `shrnr` and `shrst` are 0 and `sham` is not 0, the LSB of the right shifted output is cleared if the sticky bit is 0. If `shrnr` is clear and `shrst` is set, the LSB of the right shifted output is OR-ed together with the sticky bit — effectively forming a new sticky bit over `sham`+1 bits. If `shrnr` is set, then the right shifted result is not altered.

Fifth stage — clipper. The right shifted result is then clipped according to the value in the `clipam` field of *rd*. If the result is changed by clipping, the corresponding bit in the `vsat` register is set.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FXSUB.yy **Fixed Point Subtract**

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXSVV 11010	vt	rd	SV 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXSVS 11011	rt	rd	SV 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXSVS 11011	rt	rd	VS 1	vw	vd
6	5	5	5	1	5	5

Format:

fxsub.vv vw, vd, vt, rd
fxsub.vs vw, vd, rt, rd
fxsub.sv vw, rt, vd, rd

Description:

This command is available in vector-vector, vector-scalar, and scalar-vector forms. The vector register `vlr` is read to give n the number of elements to be operated upon. The following operations are performed under the control of the scalar configuration register *rd*. The configuration register is formatted as shown in Figure 9.

First stage — source operand mux. For a vector-vector operation, the A input to the pipeline comes from the *vd* vector register, and the B input to the pipeline comes from the *vt* vector register. For a vector-scalar operation, the A input to the pipeline comes from the *vd* vector register and the B input to the pipeline comes from the *rt* scalar register. For a scalar-vector operation, the A input to the pipeline comes from the *rt* scalar register and the B input to the pipeline comes from the *vd* vector register.

Second stage — left shifter. The A input is shifted left by the amount given in the `sham` field of *rd*. If the `shlnr` bit of *rd* is set, zeros are shifted in from the right. If the `shlnr` bit is clear, a 1 is shifted in from the 1/2 LSB position with zeros following. Only the low 32b of the result are kept, and no overflow checking is performed.

Third stage — subtractor. The B input is subtracted from the shifted A input in a 33b subtractor. The extra bit on the result ensures there can be no overflow at this stage.

Fourth stage — right shifter. The 33b subtractor result is shifted right by the number of bits given in the `sham` field of *rd*. Sign bits are shifted in to the high order bits. The bits which are shifted off to the right are OR-ed together to form a sticky bit. If the `shrnr` bit of *rd* is clear, then the right shifted output is altered depending on the sticky bit. If both `shrnr` and `shrst` are 0 and `sham` is not 0, the LSB of the right shifted output is cleared if the sticky bit is 0. If `shrnr` is clear and `shrst` is set, the LSB of the right shifted output is OR-ed together with the sticky bit — effectively forming a new sticky bit over `sham`+1 bits. If `shrnr` is set, then the right shifted result is not altered.

Fifth stage — clipper. The right shifted result is then clipped according to the value in the `clipam` field of *rd*. If the result is changed by clipping, the corresponding bit in the `vsat` register is set.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

FXMUL.yy **Fixed Point Multiply**

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXMVV 11100	vt	rd	VS 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXMVS 11101	rt	rd	VS 0	vw	vd
6	5	5	5	1	5	5

31 26	25 21	20 16	15 11	10	9 5	4 0
COP2 010010	FXMVS 11101	rt	rd	SV 1	vw	vd
6	5	5	5	1	5	5

Format:

fxmul.vv vw, vd, vt, rd
fxmul.vs vw, vd, rt, rd
fxmul.sv vw, rt, vd, rd

Description:

This command is available in vector-vector, vector-scalar and scalar-vector forms. For normal use of the instruction as described below, the vector-scalar and scalar-vector forms are identical. The vector register `vlr` is read to give n the number of elements to be operated upon. The following operations are performed under the control of the scalar configuration register *rd*. The configuration register is formatted as shown in Figure 9.

First stage — source operand mux. For a vector-vector operation, the A input to the pipeline comes from the *vd* vector register, and the B input to the pipeline comes from the *vt* vector register. For a vector-scalar operation, the A input to the pipeline comes from the *vd* vector register and the B input to the pipeline comes from the *rt* scalar register.

Second stage — sign extension. The low 16 bits of both the A and B operands are sign extended to 32 bits.

Third stage — multiplier. The sign extended A operand is multiplied by the sign extended B operand. The multiplier produces an exact 32b signed result.

Fourth stage — round. If the `shlnr` bit in *rd* is clear, a single 1 is added into the multiplier result at a position given by the `sham` field in *rd*. The `sham` field contains a bit index one greater than the position where the round bit is added. E.g. when `sham` is zero no bit is added, when `sham` is one, a 1 is added into the least significant bit of the multiplier result. The rounding bit is added in a 33b adder so no overflows can occur. If the `shlnr` bit is set, no rounding bit is added in.

Fifth stage — right shifter. The 33b multiplier result is shifted right by the number of bits given in the `sham` of *rd*. Sign bits are shifted in to the high order bits. The bits which are shifted off to the right are OR-ed together to form a sticky bit. If the `shrnr` bit of *rd* is clear, then the right shifted output is altered depending on the sticky bit to help implement different rounding schemes. If both `shrnr` and `shrst` are 0 and `sham` is not 0, the LSB of the right shifted output is cleared if the sticky bit is 0. If `shrnr` is clear and `shrst` is set, the LSB of the right shifted output is OR-ed together with the sticky bit — effectively forming a sticky bit over `sham`+1 bits. If `shrnr` is set, then the right shifted result is not altered.

Sixth stage — clipper. The right shifted result is then clipped according to the value in the `clipam` field of *rd*. If the result is changed by clipping, the corresponding bit in the `vsat` register is set.

A vector operation exception is raised if `vlr` is larger than the implementation's maximum vector length.

Exceptions:

Reserved instruction exception.
Coprocessor unusable exception.
Vector operation exception.

6 Future Extensions

There are several areas where Torrent could be extended.

- The CPU may adopt the MIPS-III (64b) ISA extensions.
- The vector coprocessor may add vector floating-point, or further assist for floating-point operations.
- The vector coprocessor may add vector 64b integer and fixed-point operations.
- The vector coprocessor may add further vector move instructions to better support certain common operations, such as sorting, FFTs, and convolutions.
- The vector coprocessor may add segmented operations that provide higher throughput for low precision arithmetic.

A T0 Fixed Point Pipe Operations

A.1 Overview

The Torrent fixed point add, fixed point subtract and fixed point multiply instructions (*fxadd*, *fxsub* and *fxmul*) use a general purpose scalar register as a "configuration register" to control their operation. The architecture manual section for these instructions describes the "fully supported" bits within these configuration words — i.e. bits that are guaranteed to function identically in all processors that implement the Torrent architecture. However, T0 (the first Torrent processor) also assigns functions to many other bits in the configuration register (although these functions are disabled when the corresponding bits are zero). This appendix describes the operation of the fixed point pipe and the effect of the bits in the configuration register on its operation.

Figure 10 details the useful bits within the configuration register — all unused bits should be set to 0.

A.2 Fixed Point Add/Subtract Pipeline

The T0 fixed point add/subtract pipeline contains 9 stages controlled by a scalar register specified in the *rd* field of the instruction. Figure 11 shows the logical structure of the fixed point add/subtract pipeline.

First stage — Operand Mux.
For a vector-vector operation, the A input to the pipeline comes from the *vd* vector register, and the B input to the pipeline comes from the *vt* vector register. For a vector-scalar operation, the A input to the pipeline comes from the *vd* vector register and the B input to the pipeline comes from the *rt* scalar register. For a scalar-vector operation, the A input to the pipeline comes from the *rt* scalar register and the B input to the pipeline comes from the *vd* vector register.

Second stage — Logic Unit
The logic unit can perform any of the 16 possible bitwise logical operations on A and B under control of the *lufunc* field, and produces a 32b result $LUOUT$. The default when *lufunc* is zero is to pass the B input unchanged. See Table 10 for bit encodings.

Third stage — Left shifter
If *shlza* is clear, the left shifter takes A as the input, otherwise it takes zero as input.

If *shlv* is clear, the shift amount is taken from the *shlam* field, otherwise the shift amount is taken from the low 5 bits of $LUOUT$.

If *shlnr* is clear, a single 1 bit (followed by zeros) is shifted in from the right in the LSB $-$ 1 position. This bit effectively adds in 1/2 LSB for the rounding modes. If *shlnr* is set, all zeros are shifted in from the right.

The left shifter output, $SHLOUT$, is 32b wide.

Fourth stage — Sign Extenders
The fourth stage extends $SHLOUT$ and $LUOUT$ to 33b to form the $ADDA$ and $ADDB$ adder inputs respectively.

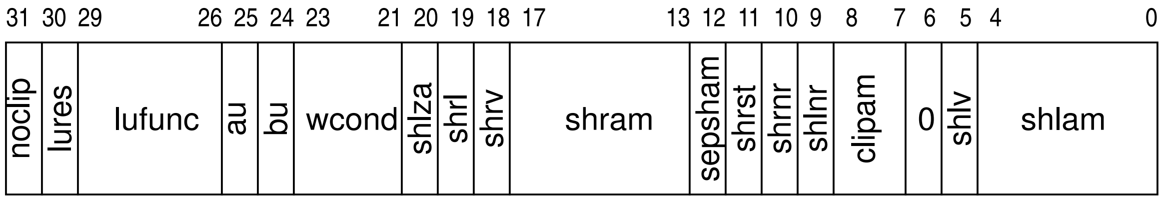


Figure 10: T0 configuration register bits

lufunc	*LUOUT*
0000	`B`
0001	`B | ~A`
0010	`A & B`
0011	`~(A ^ B)`
0100	`A | B`
0101	`~0`
0110	`A`
0111	`A | ~B`
1000	`B & ~A`
1001	`~A`
1010	`0`
1011	`~(A | B)`
1100	`A & B`
1101	`~(A & B)`
1110	`A & ~B`
1111	`~B`

Table 10: lufunc operations.

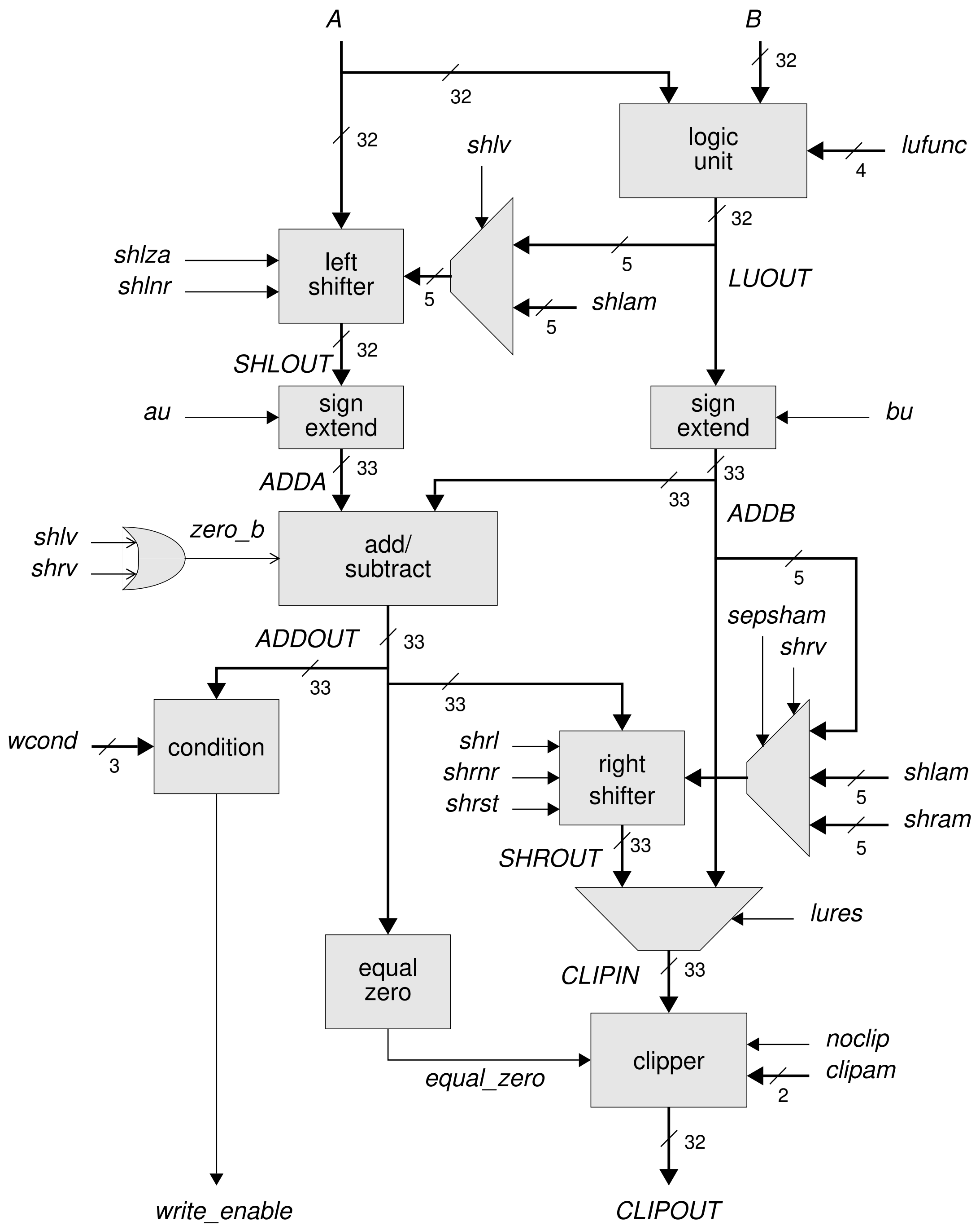


Figure 11: T0 fixed point add/subtract pipeline

If *au* is clear, *SHLOUT* is sign-extended to form *ADDA*, else *SHLOUT* is zero-extended to form *ADDA*.

If *bu* is clear, *LUOUT* is sign-extended to form *ADDB*, else *LUOUT* is zero-extended to form *ADDB*.

Fifth stage — Adder
The fifth stage is a 33b adder. If the operation is a fixed point add, *ADDA* is added to *ADDB* to give a full 33b result, *ADDOUT*. If the operation is a fixed point subtract, *ADDB* is subtracted from *ADDA* to give a 33b result, *ADDOUT*. If either *shlv* or *shrv* is set, then the *ADDB* input is ignored and the adder passes *ADDA* through unchanged.

Sixth stage — Right Shifter
The right shifter takes the 33b adder output, *ADDOUT*, and shifts it right by up to 31 places giving a 33b output *SHROUT*. It also includes sticky bit logic for round-to-nearest-even rounding.

If *shrv* is clear then the right shift amount is a constant given in the configuration register. If *sepsham* is clear, *shlam* gives the constant shift amount, otherwise the separate *shram* shift amount is used. The default is to have *shrv* and *sepsham* clear, so that both left and right shift amounts are specified by the *shlam* field. If *shrv* is set, then the low 5 bits of *ADDB* (same as low 5 bits of *LUOUT*) are used to give the shift amount.

If *shrl* is clear, the right shift is an arithmetic right shift with sign bits shifted in from the left, otherwise it is a logical right shift with zero bits shifted in form the left.

If *shrnr* is set, no rounding is applied to the right shift output. If *shrnr* is clear, *shrst* controls the type of rounding adjustment. A sticky bit value is calculated by OR-ing together all the bits that are shifted off to the right. If *shrnr* is clear and *shrst* is true, this sticky bit value is OR-ed into the least significant bit of the output.

If both *shrnr* and *shrst* are clear, then the least significant bit of the shifter output is AND-ed with the sticky bit value. This last, default, case implements the adjustment required for round-to-even rounding if the left shifter added in a round bit in the 1/2 LSB position and both left and right shift amounts are the same. When the shift amount is zero, the sticky bit must be zero but no modification should be made to the right shifter output. The hardware includes a check for constant right shift amounts ($shrv = 0$) and turns off rounding in this case, however variable right shifts ($shrv = 1$) of zero places ($ADDB[4:0] = 0$) with *shrnr* and *shrst* clear will always reset the low bit of the right shifter output.

Seventh stage — Result Mux
If the *lures* bit is clear, the 33b right shifter output *SHROUT* is passed to the clipper input *CLIPIN*, otherwise the *ADDB* value (sign-extended logic unit value) is passed to *CLIPIN*.

Eighth stage — Clipper
The clipper converts the 33b input *CLIPIN* to a 32b result *CLIPOUT*. It also generates a single bit *FLAG* which is OR-ed into the appropriate bit of the *vsat* register.

If *noclip* is clear, the clipper clips the 33b value to an 8b, 16b, or 32b value according to the *clipam* field. *CLIPIN* values larger than can be represented in the required number of bits are saturated at the most positive or most negative values possible. The *FLAG* bit is set if a saturation occurs. This is the normal usage where *vsat* indicates saturations. See Table 11 for details.

If *noclip* is set, the clipper performs alternate functions – see Table 12 for details. Note that these generate *FLAG* values which may alter *vsat*.

The "pass" function passes the low 32b of *CLIPIN* unchanged and always generates a zero *FLAG* so that *vsat* is unchanged.

The "overflow" function passes the low 32b of *CLIPIN* unchanged and generates *FLAG* if there is a signed overflow when truncating *CLIPIN* from 33b to the 32b *CLIPOUT*.

The "set if less than" function returns 1 if *CLIPIN* is negative (MSB = 1) or 0 if *CLIPIN* is positive (MSB = 0). *FLAG* is set in the same manner.

The "set if equal" function returns 1 if *ADDOUT* equals zero, 0 otherwise (note this function does not depend on *SHROUT*). *FLAG* is set in the same manner.

Ninth stage — Conditional Write
The last stage decides whether to write *CLIPOUT* to the vector register dependent on the value of *ADDOUT* and the setting in the *wcond* field. See Table 13 for details.

noclip	*clipam*	clip amount
0	00	32b
0	01	8b
0	10	16b
0	11	reserved (16b on T0)

Table 11: Clip amounts with *noclip* clear

noclip	*clipam*	Mnemonic	*CLIPOUT*	*FLAG*
1	00	pass	$CLIPIN[31:0]$	0
1	01	overflow	$CLIPIN[31:0]$	$CLIPIN[32] \neq CLIPIN[31]$
1	10	set if less than	$CLIPIN[32]$	$CLIPIN[32]$
1	11	set if equal	$ADDOUT = 0$	$ADDOUT = 0$

Table 12: Set operations with *noclip* set

wcond	mnemonic	write enable
000	always	1
001	nez	$ADDOUT \neq 0$
010	gez	$ADDOUT \geq 0$
011	lez	$ADDOUT \leq 0$
100	never	0
101	eqz	$ADDOUT = 0$
110	ltz	$ADDOUT < 0$
111	gtz	$ADDOUT > 0$

Table 13: Conditional write operations

A.3 Fixed Point Multiply Pipeline

The T0 fixed point multiply pipeline contains 8 stages controlled by a 32b CPU register specified in the rd field of the instruction. Figure 12 shows the logical structure of the fixed point multiply pipeline.

First stage — Operand Mux
For a vector-vector operation, the A input to the pipeline comes from the *vd* vector register, and the B input to the pipeline comes from the *vt* vector register. For a vector-scalar operation, the A input to the pipeline comes from the *vd* vector register and the B input to the pipeline comes from the *rt* scalar register.

Second stage — Sign Extenders
The second stage treats the least significant 16b of the A and B inputs as integers and then sign- or zero-extends them to form signed 17b inputs to the multiplier.

If *au* is clear, $MULA[16{:}0]$ is $A[15{:}0]$ sign-extended, else $MULA$ is $A[15{:}0]$ zero-extended.

If *bu* is clear, $MULB[16{:}0]$ is $B[15{:}0]$ sign-extended, else $MULB$ is $B[15{:}0]$ zero-extended.

Third stage — Multiplier
The multiplier performs a signed 17b by 17b multiply of $MULA$ and $MULB$ giving an exact signed 33b result $MULOUT$ (note only 32b required to represent product).

Fourth stage — Round bit
The fourth stage uses the left shifter to generate a rounding bit, $ROUND$, to be added into the product.

If *shlnr* is clear, *shlam* selects the bit position where the round bit will be placed. If *shlam* contains 1, the round bit will be in bit 0. If *shlam* contains 31, the round bit will be in bit 30.

If *shlnr* is set or if *shlam* is zero, no round bit is generated.

The *shlza* and *shlv* bits must be zero.

Fifth stage — Adder
The fifth stage is a 33b adder. The $ROUND$ bit is added to the multipler output $MULOUT$ to give a 33b signed result $ADDOUT$.

Sixth stage — Right Shifter
The right shifter takes the 33b adder output, $ADDOUT$ and shifts it right by up to 32 places giving a 33b output $SHROUT$. It also includes sticky bit logic for round-to-even rounding.

The field *shrv* must be clear in the multiply pipeline. If *sepsham* is clear, *shlam* gives the constant shift amount, otherwise the separate *shram* shift amount is used. The default is to have *sepsham* clear, so that both left and right shift amounts are the specified by the *shlam* field.

If *shrl* is clear, the right shift is an arithmetic right shift with sign bits shifted in from the left, otherwise it is a logical right shift with zero bits shifted in form the left.

If *shrnr* is set, no rounding is applied to the right shift output. If *shrnr* is clear, *shrst* controls the type of rounding adjustment. A sticky bit value is calculated by OR-ing together all the bits that are shifted off to the right. If *shrnr* is clear and *shrst* is set, this sticky bit value is OR-ed into the least significant bit of the output.

If both *shrnr* and *shrst* are clear, then the least significant bit of the shifter output is AND-ed with the sticky bit value. This last, default, case implements the adjustment required for round-to-even rounding if the left shifter added in a round bit in the 1/2 LSB position and both left and right shift amounts are the same. When the shift amount is zero, the sticky bit must be zero but no modification should be made to the right shifter output. The hardware includes a check for zero right shift amounts and turns off rounding in this case.

Seventh stage — Clipper
The clipper converts the 33b $SHROUT$ value to a 32b result $CLIPOUT$. It also generates a single bit $FLAG$ which is OR-ed into the appropriate

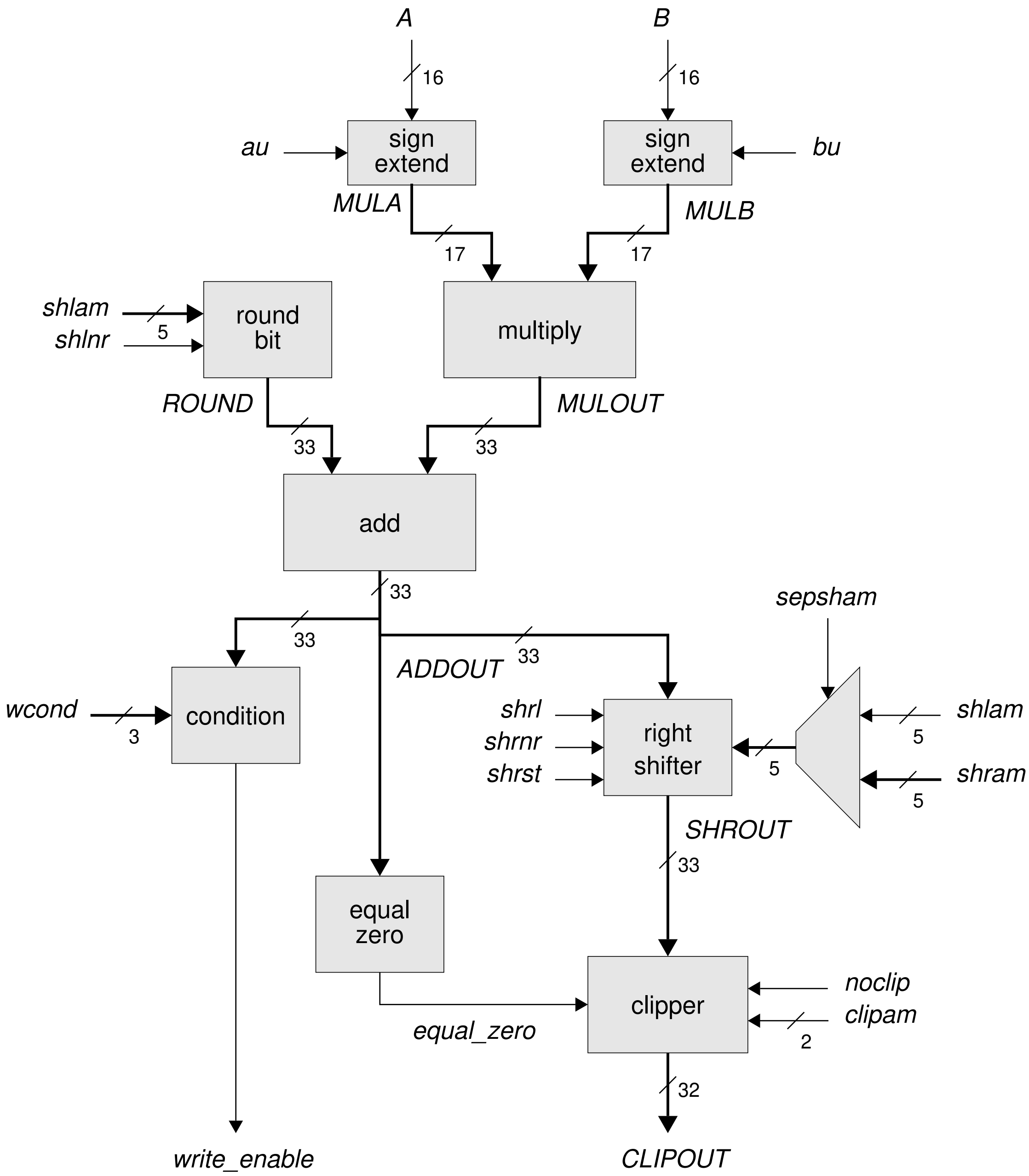


Figure 12: T0 fixed point multiply pipeline

bit of the *vsat* register.

If *noclip* is clear, the clipper clips the 33b value to an 8b, 16b, or 32b value according to the *clipam* field. *SHROUT* values larger than can be represented in the required number of bits are saturated at the most positive or most negative values possible. The *FLAG* bit is set if a saturation occurs. This is the normal usage where *vsat* indicates saturations. Table 11 for details of clip amounts.

If *noclip* is set, the clipper performs alternate functions — see Table 12 for detals. Note that these generate *FLAG* values which may alter *vsat*.

The *lures* field must be zero.

Eighth stage — Conditional Write
The last stage decides whether to write *CLIPOUT* to the vector register dependent on the value of *ADDOUT* and the setting in the *wcond* field. See Table 13 for details.