MIPS Digital Media Extension

C

C 1 Introduction

The MIPS Digital Media Extension supports video, audio, and graphics pixel processing by introducing vectors of small integers.

The MIPS Digital Media Extension (MDMX) is not a part of the MIPS Instruction Set Architecture (ISA). If a MIPS processor implements the MDMX, that implementation will follow this specification with no supersetting or subsetting. There is no requirement that a MIPS processor implement the MDMX; a processor that implements the MDMX must implement the MIPS-V ISA.

The MIPS MDMX is not intended for general purpose computing. Software support for the MDMX is via shared libraries (DSOs) and assembly language only. Compiler support is neither implied nor planned.

C 2 Register files

The Digital Media extension shares a register file with the Floating Point Unit. Data is moved between the shared register file and memory with existing Floating Point load and store doubleword operations (LDC1, SDC1, LDXC1, SDXC1, LUXC1, and SUXC1). These operations were extended with MIPS-V to include unaligned (that is, ignore any misaligned) loads and stores. Alignment within a double word is performed by Align and Merge instructions. The DMTC1 and DMFC1 instructions may also be used to move data to and from the integer GPRs.

The registers are interpreted in two new formats: Quad Half (QH) and Oct Byte (OB). In Quad Half format, a 64-bit FPR is interpreted as a vector of 4 signed 16-bit integers. In Oct Byte format, a 64-bit FPR is interpreted as a vector of 8 unsigned 8-bit integers. There is no data format conversion between floating-point and the new formats.

The MDMX also shares the 8 Floating Point Condition Code bits. Unlike the FPU, the MDMX is capable of reading and writing subsets or even all 8 of these bits simultaneously during vector compare and select operations.

The MDMX has a private 192-bit Accumulator register. The format of the Accumulator is determined by the format of the elements accumulated. In QH format, the Accumulator contains 4 48-bit elements; in OB format, the Accumulator contains 8 24-bit elements. Accumulator elements are always signed.The Accumulator cannot be directly loaded from or stored to main memory, but rather must be staged through the shared FP register file.

Digital Media operations always write all 192 bits of the Accumulator or all 64 bits of an FPR, or the condition codes. Results are not stored to multiple destinations (including the condition codes).

C 3 Exceptions

With the exception of the SHFL instruction, integer vector operations that write to the FPRs clamp the values being written to the target's representable range. Integer vector operations that write to an Accumulator do not clamp their values before writing, but allow underflows and overflows to wrap around the representable range. It is the responsibility of software to ensure that unwanted overflows and underflows do not occur when writing to the Accumulator or FPRs.

C 4 Instruction Format and VT Selection

The *fmt/sel* field in many integer vector instructions specifies the data format and those elements of vector *vt* which are used with each element of the accumulator *acc*, vector *vs*, or vector *vd*. The format encoding is shown in Table C-1 below. The BW and L formats are reserved for future use.

Table C-1 Format Encoding

fmt/sel	Format
s s s s 0	OB (oct byte)
s s s 0 1	QH (quad halfword)
s s 0 1 1	BW (bi word), reserved
s s 1 1 1	L (long), reserved

The part of the field labeled “s” indicates the VT selection for the specified format. Table C-2 describes the VT select encoding:

Table C-2 Select Encoding

fmt/sel	VT select
0 x x x x	element select
1 0 x x x	select vector
1 1 x x x	select immediate

Element select will select one element in VT and replicate it for every element of VT. For select vector, VT is passed without any modification. For select immediate, the VT field of the instruction opcode is used as an immediate value that is replicated for every element of VT.

The following two tables, Table C-3 and Table C-4, show all valid OB and QH *sel* / *fmt* encodings and the vector element used. All other encodings are reserved or invalid.

Table C-3 OB Format and Selects

fmt/sel	OB Element							
	H	G	F	E	D	C	B	A
0 000 0	A	A	A	A	A	A	A	A
0 001 0	B	B	B	B	B	B	B	B
0 010 0	C	C	C	C	C	C	C	C
0 011 0	D	D	D	D	D	D	D	D
0 100 0	E	E	E	E	E	E	E	E
0 101 0	F	F	F	F	F	F	F	F
0 110 0	G	G	G	G	G	G	G	G
0 111 0	H	H	H	H	H	H	H	H
10 11 0	H	G	F	E	D	C	B	A
11 11 0	#	#	#	#	#	#	#	#

Table C-4 QH Format and Selects

fmt/sel	QH Element			
	D	C	B	A
0 00 01	A	A	A	A
0 01 01	B	B	B	B
0 10 01	C	C	C	C
0 11 01	D	D	D	D
10 1 01	D	C	B	A
11 1 01	#	#	#	#

Most commonly, elements of vector *vt* are used with the same-numbered elements of the other vector operands, in which case the *fmt/sel* field contains binary 10xxx, and the assembly notation looks like any other vector register, e.g.:

```
sub.ob      $v4, $v7, $v2
```

However, the *fmt/sel* field can also direct that the second argument to an instruction be a vector of immediates -- copies of the *vt* field interpreted as a five bit unsigned number, like this:

```
add.qh      $v10, $v9, 25
```

An element of vector *vt* can be propagated, to be used with each of the elements of the other vector operand *vs*. Following is the notation to propagate one element of vector *vt* to be used in every element of the computation:

```
addl.ob     $acc0, $v4, $v6[7]
```

C 5 Data format conversion

There is no implicit data type conversion from QH to OB or from OB to QH. Both are stored as bit-arrays in memory, however the internal floating-point register formats may differ. QH and OB vectors may be read or written without regard to datatype. Conversion from a bit-array to either a QH or OB occurs during the execution of the first MDMX opcode which includes a format field (e.g., ADD). Subsequent operations must use the same datatype; mixing QH and OB operations without explicit register content conversion results in an undefined operation.

The shuffle (SHFL) and MIN / MAX operators can be used to convert QH and OB vectors. To convert either the lower or upper bytes of an OB vector into a QH vector, the SHF.UPUL and SHF.UPUH can be used to convert the lower [0:3] and upper [4:7] unsigned bytes into signed halves. To convert two QH vectors to an OB

vector, the QH vectors should first be clamped to 0..255, then packed (via SHFL.PACL.OB). Clamping can be done with the MIN.QH and MAX.QH instructions.

C 6 Description of an Instruction

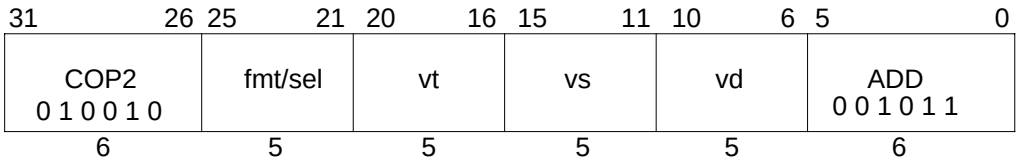
For the Digital Media instruction documentation, all variable subfields in an instruction format (such as *vs, vt, acc, sel,* and so on) are shown in lower-case. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

In some instructions, the instruction subfields *op* and *function* can have constant 6- and 5-bit values. When reference is made to these instructions, upper-case mnemonics are used. For instance, in the floating-point ADD instruction uses *op* = COP1 and *function* = ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters.

C 7 Opcode encoding

		2..0	**function (for opcode = COP2)**						
	bits	0	1	2	3	4	5	6	7
	5..3	000	001	010	011	100	101	110	111
0	000	MSGN	C.EQ	PICKF	PICKT	C.LT	C.LE	MIN	MAX
1	001	†	†	SUB	ADD	AND	XOR	OR	NOR
2	010	SLL	†	SRL	SRA	†	†	†	†
3	011	ALNI.OB	ALNV.OB	ALNI.QH	ALNV.QH	†	†	†	SHFL
4	100	RZU	RNAU	RNEU	†	RZS	RNAS	RNES	†
5	101	†	†	†	†	†	†	†	†
6	110	MUL	†	MULS{,L}	MUL{A,L}	†	†	SUB{A,L}	ADD{A,L}
7	111	†	†	†	†	†	†	WAC	RAC

Vector Add **ADD.fmt**



Format: ADD.QH vd, vs, vt MDMX
ADD.OB vd, vs, vt

Purpose: To add integer vectors.

Description: vd[i] ← vs[i]+select(i,sel,vt)

The values in vector *vt* are added to the values in vector *vs*. Saturated arithmetic is performed, such that overflows and underflows clamp to the largest or smallest representable value before writing to vector *vd*.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-3 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

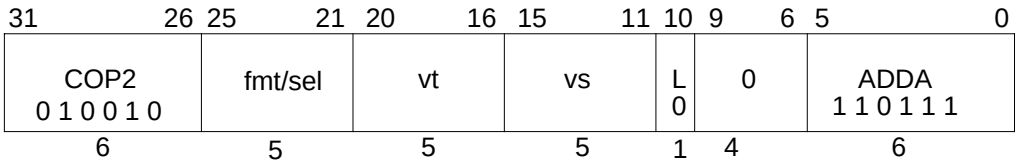
```
StoreFPR (vd, fmt, Clamp(FGR[vs] + FGR[vt]))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

ADDA.fmt

Accumulate Vector Add



Format: ADDA.QH vs, vt MDMX
ADDA.OB vs, vt

Purpose: To add integer vectors.

Description: acc[i] ← acc[i]+vs[i]+select(i,sel,vt)

The values in vector *vt* and vector *vs* are added to those in the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

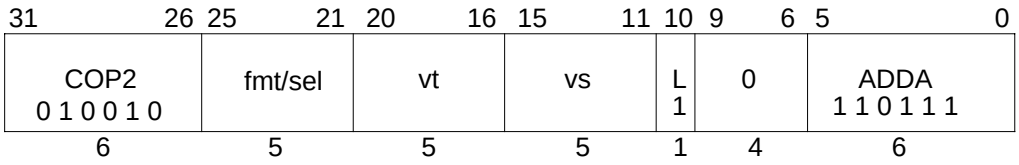
```
StoreACC (acc, fmt, Wrap(ValueACC(acc, fmt) + FGR[vs] + FGR[vt]))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Load Vector Add

ADDL.fmt



Format: ADDL.QH vs, vt MDMX
ADDL.OB vs, vt

Purpose: To add integer vectors.

Description: acc[i] ← vs[i]+select(i,sel,vt)

The values in vector *vt* and vector *vs* are added to those in the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

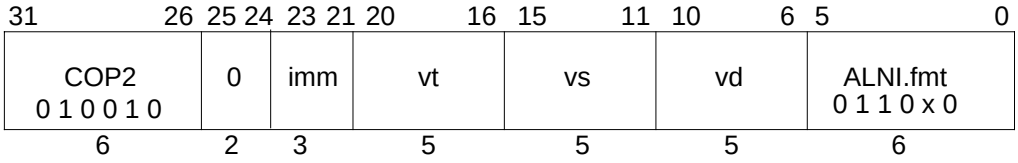
Operation:

```
StoreACC (acc, fmt, FGR[vs] + FGR[vt])
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

ALNI.fmt — Vector align, Constant Alignment



Format: ALNI.QH vd, vs, vt, imm — MDMX
ALNI.OB vd, vs, vt, imm

Purpose: To perform a byte-wise funnel shift.

Description: vd ← ByteAlign($imm_{2..0}$, vs, vt)

The align amount is computed by masking the immediate, then using that value to control a funnel shift of vector *vs* concatenated with vector *vt*.

This operation is a media unit operation, and so no data-dependent exceptions are possible.

The operands must be a value in QH or OB format. If not, the results are undefined and the values of the operand vectors become undefined.

This operation does not interpret the format of the registers specified.

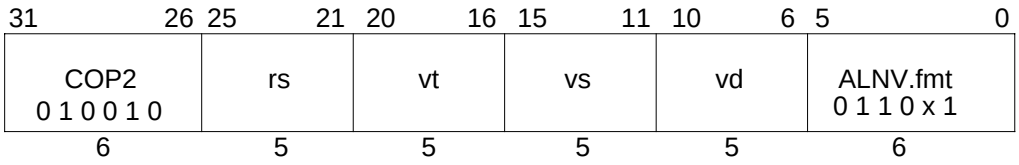
Operation:

```
s ← imm2..0*8
if BigEndianCPU then
    vd ← (vs || vt)127-s..64-s
else
    vd ← (vs || vt)63+s..s
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Vector Align, Variable Alignment — ALNV.fmt



Format: ALNV.QH vd, vs, vt, rs — MDMX
ALNV.OB vd, vs, vt, rs

Purpose: To perform a byte-wise funnel shift.

Description: vd ← ByteAlign($rs_{2..0}$, vs, vt)

The align amount is computed by masking the contents of GPR *rs*, then using that value to control a funnel shift of vector *vs* concatenated with vector *vt*.

This operation is a media unit operation, and so no data-dependent exceptions are possible.

The operands must be a value in QH or OB format. If not, the results are undefined and the values of the operand vectors become undefined.

This operation does not interpret the format of the registers specified.

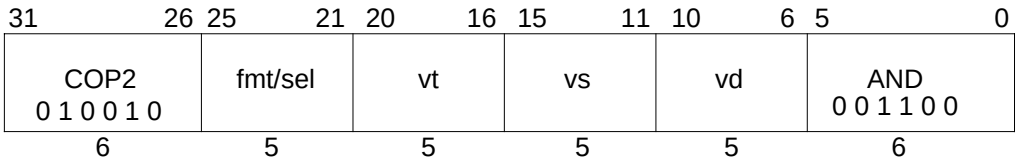
Operation:

```
s ← GPR[rs]2..0*8
if BigEndianCPU then
    vd ← (vs || vt)127-s..64-s
else
    vd ← (vs || vt)63+s..s
endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

AND.fmt — Vector And



Format: AND.QH vd, vs, vt — MDMX
AND.OB vd, vs, vt

Purpose: To do a bitwise logical AND.

Description: vd[i] ← vs[i] AND select(i,sel,vt)

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical AND operation.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

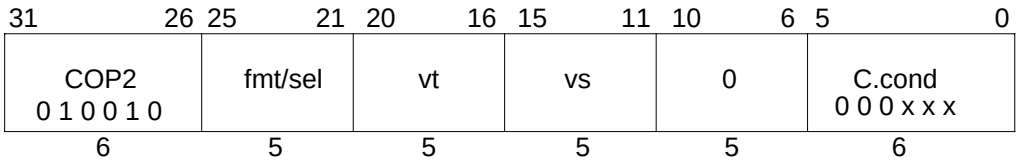
Operation:

```
StoreFPR(fd, fmt, ValueFPR(fs,fmt) and ValueFPR(ft,fmt))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Vector Compare **C.cond.fmt**



Format: C.cond.QH vs, vt MDMX
C.cond.OB vs, vt

Purpose: To perform vector comparison.

Description: cc[i] ← vs[i] cond select(i,sel,vt)

The values in vector *vt* are compared to the values in vector *vs*, and the result is written to the condition codes. In OB format, all 8 CC bits are set. In QH format, *cc* bits 0 through 3 are written, and *cc* bits 4 through 7 are unaffected.

The comparisons available are less than (LT), less than or equal (LE), and equal (EQ). The inverse comparisons (GE, GT, NE) are not necessary; the instructions that use condition codes (BC1F, BC1T, MOVF, MOVT, PICKF, PICKT) all allow both cc=0 and cc=1 tests. Both LT and LE comparisons are necessary since the operands are not symmetrical — every element of vector *vs* is used, whereas *sel* selects the values of *vt*[] used for each i.

The operands are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

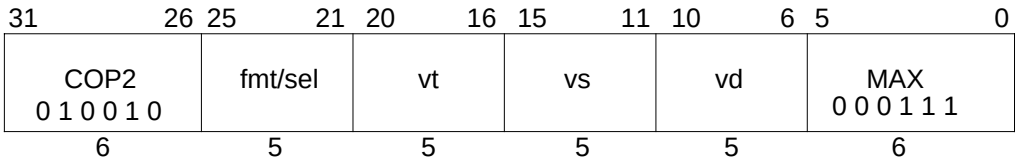
The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

MAX.fmt — Vector Maximum



Format: MAX.QH vd, vs, vt — MDMX
MAX.OB vd, vs, vt

Purpose: To perform vector maximum.

Description: vd[i] ← max(vs[i], select(i,sel,vt))

The values in vector *vt* are compared to the values in vector *vs,* and the larger is written to each element of vector *vd*.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

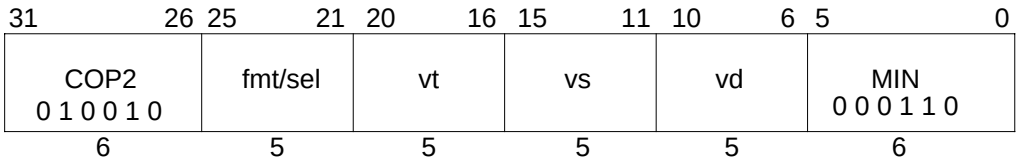
The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

Vector Minimum — MIN.fmt



Format: MIN.QH vd, vs, vt — MDMX
MIN.OB vd, vs, vt

Purpose: To perform vector minimum.

Description: vd[i] ← min(vs[i], select(i,sel,vt))

The values in vector *vt* are compared to the values in vector *vs*, and the smaller is written to each element of vector *vd*.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

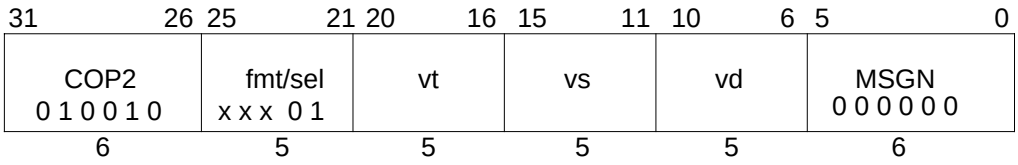
Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

MSGN.fmt

Vector Sign



Format: MSGN.QH vd, vs, vt MDMX

Purpose: To multiply sign bits from one vector by another.

Description: vd[i] ← (vs[i] < 0) ? -select(i,sel,vt) : ((vs[i] = 0) ? 0 : select(i,sel,vt))

The values in vector *vt* are multiplied by the sign of the values in vector *vs*, and the result is written to vector *vd*. If an element of vector *vs* is zero, the corresponding element of vector *vd* is set to zero.

Should select(i,sel,*vt*) be the maximum negative value (-2^{15}), and *vs*[i] < 0, then -select(i,sel,*vt*) will overflow and be clamped to the maximum positive value ($2^{15} - 1$).

The operands are values in integer vector format QH. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

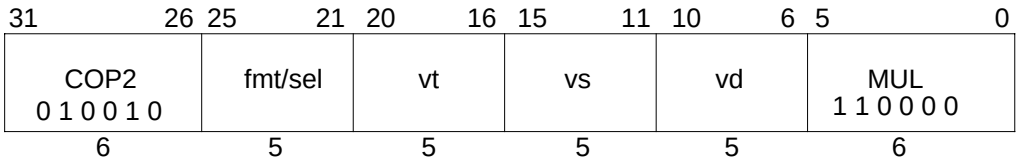
The operands must be a value in format QH. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

Vector Multiply — MUL.fmt



Format: MUL.QH vd, vs, vt — MDMX
MUL.OB vd, vs, vt

Purpose: To multiply integer vectors.

Description: vd[i] ← vs[i]*select(i,sel,vt)

The values in vector *vt* are multiplied by the values in vector *vs*, and the product is written into vector *vd*. Saturated arithmetic is performed, such that overflows and underflows clamp to the largest or smallest representable value before writing to vector *vd*.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

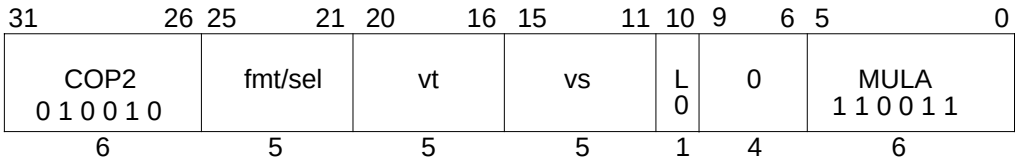
```
StoreFPR (vd, fmt, Clamp(FGR[vs] * FGR[vt]))
```

Exceptions:

Coprocessor Unusable

MULA.fmt

Accumulate Vector Multiply



Format: MULA.QH vs, vt MDMX
MULA.OB vs, vt

Purpose: To perform a combined multiply-then-add of integer vectors.

Description: acc[i] ← acc[i]+(vs[i]*select(i,sel,vt))

The values in vector *vt* are multiplied by the values in vector *vs,* and the product is added to the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

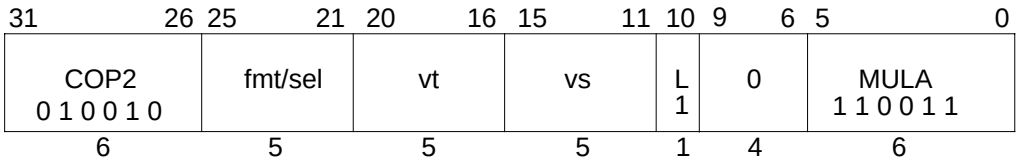
Operation:

```
StoreACC (acc, fmt, Wrap(ValueACC(acc,fmt) + (FGR[vs] * FGR[vt])))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Add Vector Multiply to Accumulator — MULL.fmt



Format: MULL.QH vs, vt — MDMX
MULL.OB vs, vt

Purpose: To perform a combined multiply-then-add of integer vectors.

Description: acc[i] ← vs[i]*select(i,sel,vt)

The values in vector *vt* are multiplied by the values in vector *vs*, and the product is added to the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

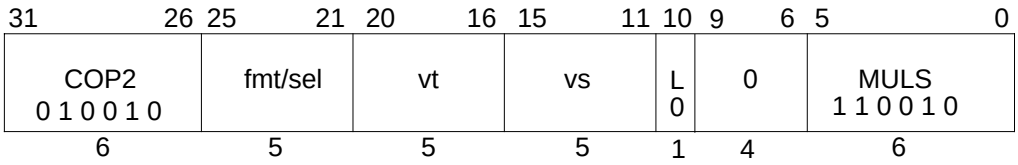
Operation:

```
StoreACC (acc, fmt, FGR[vs] * FGR[vt])
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

MULS.fmt — Subtract Vector Multiply from Accumulator



Format: MULS.QH vs, vt MDMX
MULS.OB vs, vt

Purpose: To perform a combined multiply-then-subtract of integer vectors.

Description: acc[i] ← acc[i]-(vs[i]*select(i,sel,vt))

The values in vector *vt* are multiplied by the values in vector *vs,* and the product is subtracted from the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

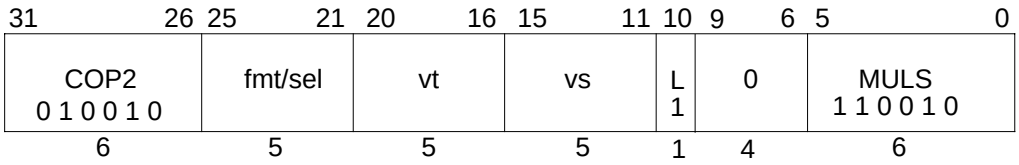
```
StoreACC (acc, fmt, Wrap(ValueACC(acc, fmt) - (FGR[vs] * FGR[vt])))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Load Negative Vector Multiply

MULSL.fmt



Format: MULSL.QH vs, vt MDMX
MULSL.OB vs, vt

Purpose: To perform a combined multiply-then-subtract of integer vectors.

Description: acc[i] ← -(vs[i]*select(i,sel,vt))

The values in vector *vt* are multiplied by the values in vector *vs*, and the product is subtracted from the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

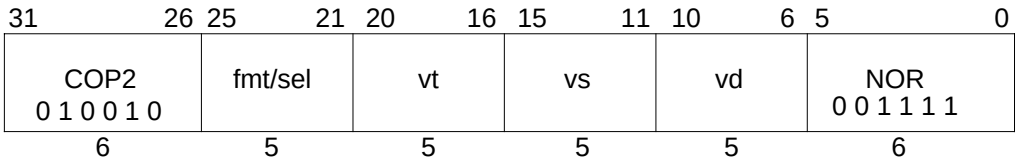
```
StoreACC (acc, fmt, - (FGR[vs] * FGR[vt]))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

NOR.fmt

Vector Nor



Format: NOR.QH vd, vs, vt MDMX
NOR.OB vd, vs, vt

Purpose: To do a bitwise logical NOR.

Description: vd[i] ← vs[i] NOR select(i,sel,vt)

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical NOR operation.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

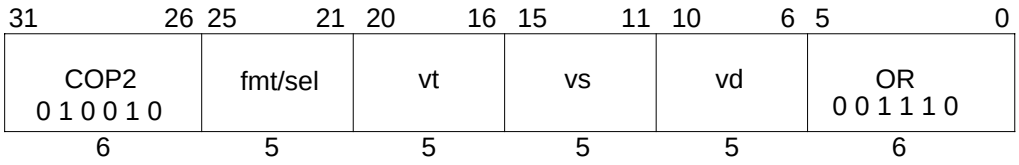
Operation:

```
StoreFPR(fd, fmt, ValueFPR(fs,fmt) nor ValueFPR(ft,fmt))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Vector Or — OR.fmt



Format: OR.QH vd, vs, vt — MDMX
OR.OB vd, vs, vt

Purpose: To do a bitwise logical OR.

Description: vd[i] ← vs[i] OR select(i,sel,vt)

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical OR operation.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

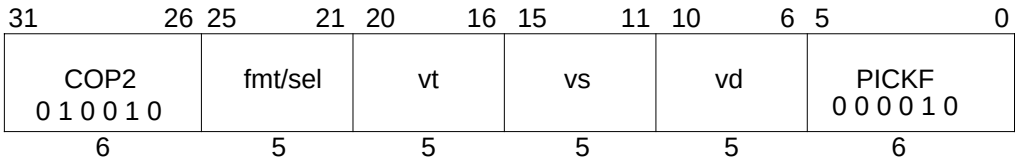
```
StoreFPR(fd, fmt, ValueFPR(fs,fmt) or ValueFPR(ft,fmt))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

PICKF.fmt

Select Vector Elements



Format: PICKF.QH vd, vs, vt MDMX
PICKF.OB vd, vs, vt

Purpose: To select elements of a vector.

Description: vd[i] ← cc[i] = 0 ? vs[i] : select(i,sel,vt)

Depending on the *cc* bits, the vector *vd* is written with either the corresponding element of vector *vs* or the corresponding element of vector *vt*. When operating on OB format data, all 8 *cc* bits are used. When operating on QH format data, *cc* bits 0 through 3 are used.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

Both PICKF and PICKT are necessary since the operands are not symmetrical — every element of vector *vs* is used, whereas *sel* selects the values of *vt*[] used for each i.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

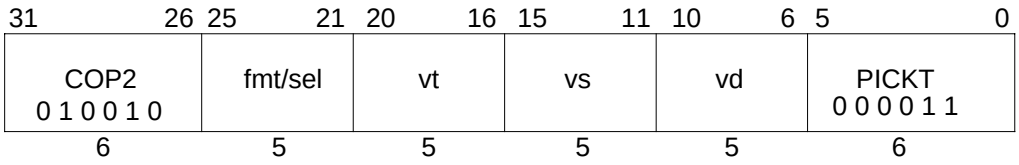
Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

Select Vector Elements

PICKT.fmt



Format: PICKT.QH vd, vs, vt MDMX
PICKT.OB vd, vs, vt

Purpose: To select elements of a vector.

Description: vd[i] ← cc[i] = 1 ? vs[i] : select(i,sel,vt)

Depending on the *cc* bit, the vector *vd* is written with either the corresponding element of vector *vs* or the corresponding element of vector *vt*. When operating on OB format data, all 8 *cc* bits are used. When operating on QH format data, *cc* bits 0 through 3 are used.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

Both PICKF and PICKT are necessary since the operands are not symmetrical — every element of vector *vs* is used, whereas *sel* selects the values of *vt*[] used for each i.

This operation is a signal processing operation, no data-dependent exceptions are possible.

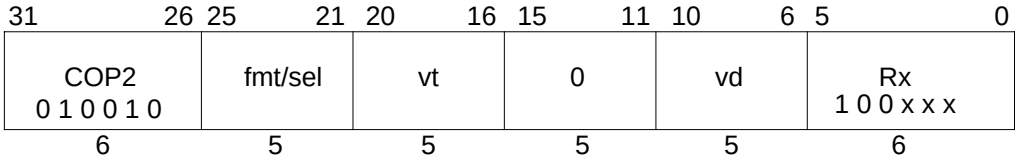
The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

Rx.fmt — Scale, Round and Clamp Accumulator



Format: Rx.QH vd, vt MDMX
Rx.OB vd, vt

Purpose: To scale, round and then clamp an accumulator's values into a vector register.

Description: vd[i] ← Clamp(Round(acc[i] >> select(i,sel,vt)))

The values in the Accumulator are shifted right by the values in vector *vt*, rounded by the indicated mode, and clamped to either a signed or unsigned subset of the range of *vd*[]. This is the only instruction type that can do an unsigned quad-half clamp.

The *vt* operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

In the QH format, if an element of *vt*[] is negative, the corresponding element of *vd*[] is undefined. If an element of *vt*[] is greater than 48, all significant bits will be shifted away and the result will be zero. In the OB format, if an element of *vt*[] is greater than 24, then the result will be zero.

The rounding modes available depend on the format selected, and in the QH format are available in signed and unsigned versions, as shown below:

Rounding Modes Used in Rx.fmt

Rounding direction	Quad Half format		Oct Byte format
	Signed	Unsigned	Unsigned
all fractional values round toward zero	RZS.QH	RZU.QH	RZU.OB
to nearest, exactly halfway rounds away from zero	RNAS.QH	RNAU.QH	RNAU.OB
to nearest, exactly halfway rounds to even	RNES.QH	RNEU.QH	RNEU.OB

This operation is a signal processing operation, no data-dependent exceptions are possible.

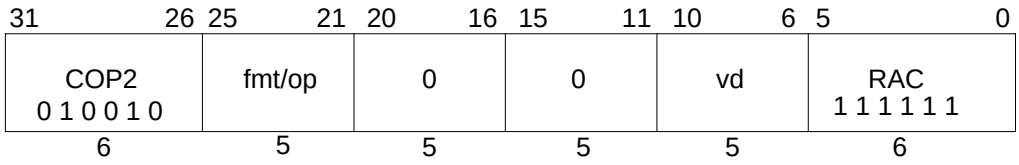
The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

RAC.fmt — Read Accumulator



Format: MDMX

```
RACL.QH   vd
RACM.QH   vd
RACH.QH   vd
RACL.OB   vd
RACM.OB   vd
RACH.OB   vd
```

Purpose: To read sections of the accumulator into a vector register.

Description: vd[i] ← acc[i].{low, med, high}

Read either the least significant, middle significant, or most significant third of the bits of the Accumulator elements. No clamping of the values extracted is performed; the bits are simply copied into elements of *vd*[].

The field *fmt/op* specifies which of the 8 or 16 bits of the Accumulator to read the following:

RAC fmt/op Encodings

operation	fmt / op	
	OB Fmt	QH Fmt
RACL	0000 0	000 01
RACM	0100 0	010 01
RACH	1000 0	100 01

This operation is a signal processing operation, no data-dependent exceptions are possible.

A RACL / RACM / RACH followed by WACL / WACH are used to save and restore the Accumulator. This save:restore function is format independent, either format can be used to save or restore Accumulator values generated by either QH or OB operations. There is no implied data conversion; the mapping between element bits of the OB format Accumulator and bits of the same Accumulator interpreted in QH format is implementation specific, but consistent for each implementation.

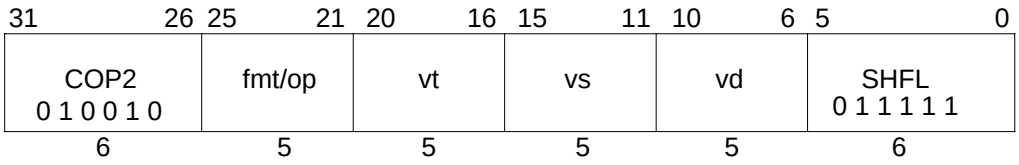
Operation:

Exceptions:

Coprocessor Unusable

Vector Element Shuffle

SHFL.op.fmt



Format: SHFL.op.QH vd, vs, vt MDMX
SHFL.op.OB vd, vs, vt

Purpose: To make a new vector of the elements of two other vectors.

Description: vd[i] ← one of vs[j] or vt[j]

Elements of vectors vs and vt and merged into a new vector. All possible value rearrangings are not available -- the operations of the variants of this instruction are tailored to the data movement patterns of specific calculations. The shuffles available in OB and QH formats are given in the tables below.

Note that UPSL.OB and UPSH.OB are the only MU instructions that interpret an element of an OB format vector as a signed quantity.

The operands are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt* encoding. The remaining bits in the field are not used for a *vt*[] select but rather are used to encode the shuffle operation.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Oct Byte Shuffles

fmt/op	Operation	vd[7]	vd[6]	vd[5]	vd[4]	vd[3]	vd[2]	vd[1]	vd[0]
0000 0	UPUH	0	vs[7]	0	vs[6]	0	vs[5]	0	vs[4]
0001 0	UPUL	0	vs[3]	0	vs[2]	0	vs[1]	0	vs[0]
0010 0	UPSH	sign vs[7]	vs[7]	sign vs[6]	vs[6]	sign vs[5]	vs[5]	sign vs[4]	vs[4]
0011 0	UPSL	sign vs[3]	vs[3]	sign vs[2]	vs[2]	sign vs[1]	vs[1]	sign vs[0]	vs[0]
0100 0	PACH	vs[7]	vs[5]	vs[3]	vs[1]	vt[7]	vt[5]	vt[3]	vt[1]
0101 0	PACL	vs[6]	vs[4]	vs[2]	vs[0]	vt[6]	vt[4]	vt[2]	vt[0]
0110 0	MIXH	vs[7]	vt[7]	vs[6]	vt[6]	vs[5]	vt[5]	vs[4]	vt[4]
0111 0	MIXL	vs[3]	vt[3]	vs[2]	vt[2]	vs[1]	vt[1]	vs[0]	vt[0]

Quad Half shuffles

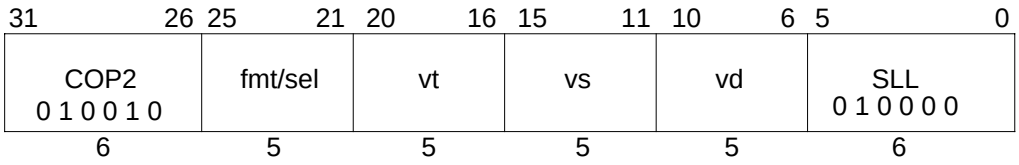
fmt/op	Operation	vd[3]	vd[2]	vd[1]	vd[0]
000 01	MIXH	vs[3]	vt[3]	vs[2]	vt[2]
001 01	MIXL	vs[1]	vt[1]	vs[0]	vt[0]
010 01	PACH	vs[3]	vs[1]	vt[3]	vt[1]
011 01	PACL	vs[2]	vs[0]	vt[2]	vt[0]
100 01	BFLA	vs[2]	vt[3]	vs[0]	vt[1]
101 01	BFLB	vs[0]	vt[1]	vs[2]	vt[3]
110 01	REPA	vs[3]	vs[2]	vt[3]	vt[2]
111 01	REPB	vs[1]	vs[0]	vt[1]	vt[0]

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

Vector Shift Left Logical — SLL.fmt



Format: SLL.QH vd, vs, vt — MDMX
SLL.OB vd, vs, vt

Purpose: To shift a vector's elements by a variable number of bits.

Description: vd[i] ← vs[i] << select(i,sel,vt)

Each element of vector *vs* is shifted left by an amount specified by the corresponding element of vector *vt*, and zeros are shifted into the low-order bits. The results are written into vector *vd*. In QH format, all but the lower 4 bits of the shift amount are masked to zero; the largest shift possible is 15 places. In OB format, all but the lower 3 bits of the shift amount are masked to zero; the largest possible shift is 7 places.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

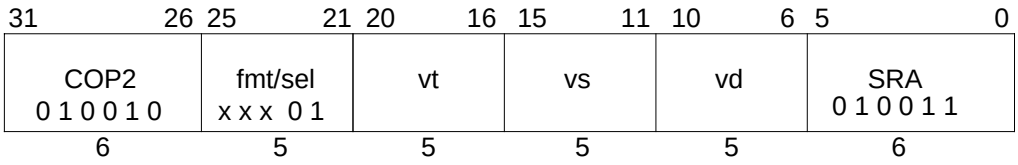
The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

SRA.fmt — Vector Shift Right Arithmetic



Format: SRA.QH vd, vs, vt MDMX

Purpose: To arithmetic right shift a vector.

Description: vd[i] ← vs[i] >> select(i,sel,vt)

Each element of vector *vs* is shifted right by an amount specified by the corresponding element of vector *vt*. The high-order bits are filled with copies of the original sign bit. The results are written into vector *vd*. All but the lower 4 bits of the shift amount are masked to zero; the largest shift possible is 15 places. This operation is undefined for the OB format, since values in that format are unsigned.

The operands and results are values in integer vector format QH. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

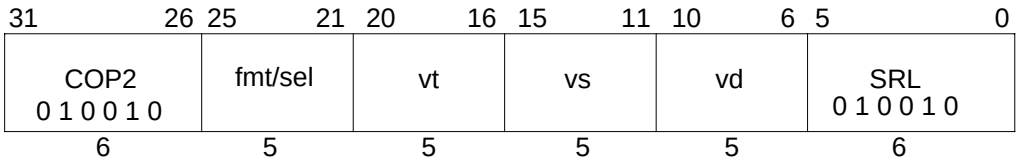
The operands must be a value in the QH format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

Vector Shift Right Logical — SRL.fmt



Format: SRL.QH vd, vs, vt — MDMX
SRL.OB vd, vs, vt

Purpose: To shift a vector's elements by a variable number of bits.

Description: vd[i] ← vs[i] >> select(i,sel,vt)

Each element of vector *vs* is shifted right by an amount specified by the corresponding element of vector *vt*, and zeros are shifted into the high-order bits. The results are written into vector *vd*. In QH format, all but the lower 4 bits of the shift amount are masked to zero; the largest shift possible is 15 places. In OB format, all but the lower 3 bits of the shift amount are masked to zero; the largest possible shift is 7 places.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

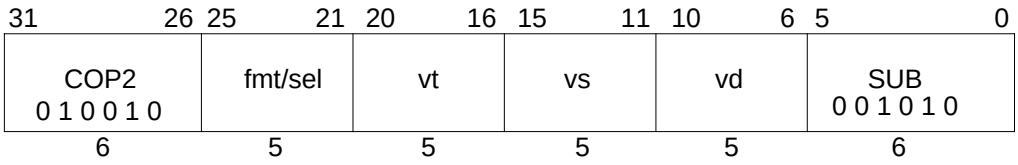
The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

SUB.fmt — Vector Subtract



Format: SUB.QH vd, vs, vt MDMX
SUB.OB vd, vs, vt

Purpose: To subtract integer vectors.

Description: vd[i] ← vs[i]-select(i,sel,vt)

The difference of the values in vector *vt* and vector *vs* are written into vector *vd*. Saturated arithmetic is performed, such that overflows and underflows clamp to the largest or smallest representable value before writing to vector *vd*.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

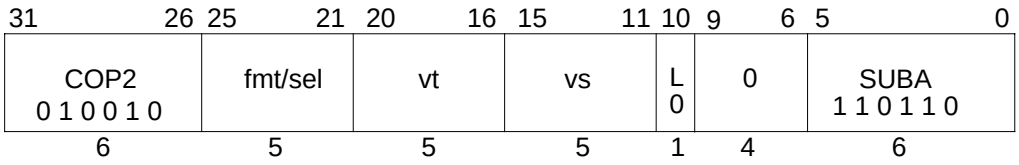
```
StoreFPR (vd, fmt, Clamp(FGR[vs] - FGR[vt]))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Accumulate Vector Difference

SUBA.fmt



Format: SUBA.QH vs, vt MDMX
SUBA.OB vs, vt

Purpose: To subtract integer vectors and accumulate the difference.

Description: acc[i] ← acc[i]+vs[i]-select(i,sel,vt)

The differences of vector *vt* and vector *vs* are added to those in the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

If *L* is 1 then the Accumulator is cleared to zero before the operation.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

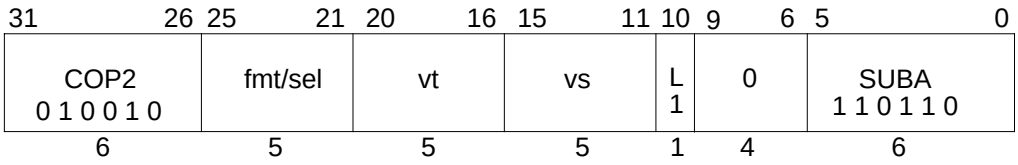
```
StoreACC (acc, fmt, Wrap(ValueACC(acc, fmt) + FGR[vs] - FGR[vt]))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

SUBL.fmt

Load Vector Difference



Format: SUBL.QH vs, vt MDMX
SUBL.OB vs, vt

Purpose: To subtract integer vectors.

Description: acc[i] ← vs[i]-select(i,sel,vt)

The differences of vector *vt* and vector *vs* are added to those in the Accumulator. Wrapped arithmetic is performed, such that overflows and underflows wrap around the Accumulator's representable range before being written into the Accumulator.

The operands are values in integer vector format *fmt*. The Accumulator is in the corresponding Accumulator vector format. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

If *L* is 1 then the Accumulator is cleared to zero before the operation.

This operation is a signal processing operation, no data-dependent exceptions are possible.

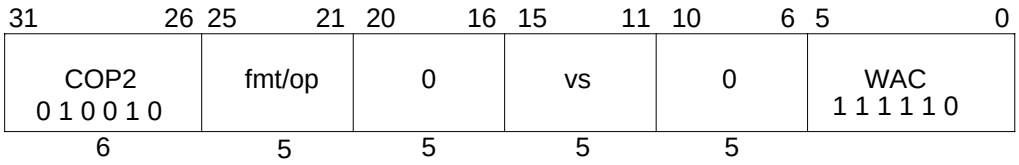
The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

```
StoreACC (acc, fmt, Wrap(FGR[vs] - FGR[vt]))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction



Format: WACH.QH vs MDMX
WACH.OB vs

Purpose: To write sections of the Accumulator from a vector register.

Description: acc[i].high ← vs[i]

Write the most significant third of the bits of the Accumulator elements. The least significant two thirds of the bits of the Accumulator elements are unaffected.

The field *fmt/op* specifies which of the 8- or 16-bits of the Accumulator to read, as shown below.

WACH.fmt Instruction fmt/op Field

operation	fmt/op	
	OB Fmt	QH Fmt
WACH	1000 0	100 01

This operation is a signal processing operation, no data-dependent exceptions are possible.

A RACL/RACM/RACH followed by WACL/WACH are used to save and restore the Accumulator. This save:restore function is format independent, either format can be used to save or restore Accumulator values generated by either QH or OB operations. There is no implied data conversion; the mapping between element bits of the OB format Accumulator and bits of the same Accumulator interpreted in QH format is implementation specific, but consistent for each implementation.

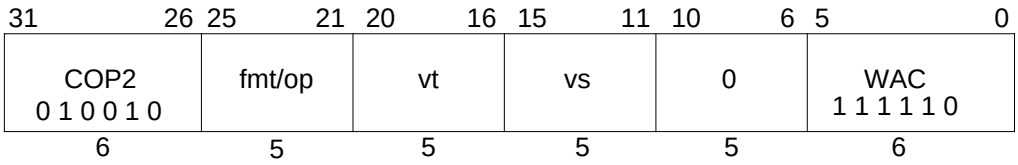
This instruction is the only instruction that writes a portion of the Accumulator.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction

WACL.fmt — Write Accumulator Low



Format: WACL.QH vs, vt — MDMX
WACL.OB vs, vt

Purpose: To load the Accumulator from a vector register.

Description: acc[i] ← {sign(vs[i]) x 16, vs[i], vt[i]} for QH
acc[i] ← {sign(vs[i]) x 8, vs[i], vt[i]} for OB

Write the least significant two thirds of the bits of the Accumulator elements. The upper one third of the bits of the Accumulator elements are written by the sign bits of the corresponding elements of vector *vs*[], replicated by 16 or 8, depending on the format.

The field *fmt/op* specifies which of the 8 or 16 bits of the Accumulator to read.

WACL.fmt Instruction fmt/op Field

operation	fmt / op	
	OB Fmt	QH Fmt
WACL	0000 0	000 01

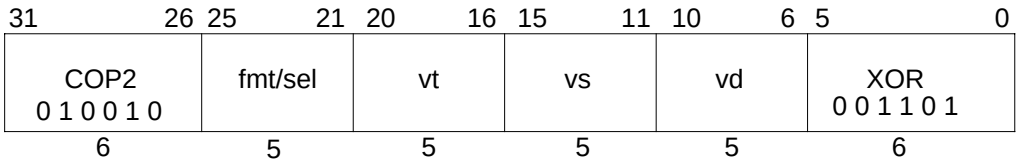
This operation is a signal processing operation, no data-dependent exceptions are possible.

A RACL / RACM / RACH followed by WACL / WACH are used to save and restore the Accumulator. This save:restore function is format independent, either format can be used to save or restore Accumulator values generated by either QH or OB operations. There is no implied data conversion; the mapping between element bits of the OB format Accumulator and bits of the same Accumulator interpreted in QH format is implementation specific, but consistent for each implementation.

Operation:

Exceptions:

Coprocessor Unusable
Reserved Instruction



Format: XOR.QH vd, vs, vt MDMX
XOR.OB vd, vs, vt

Purpose: To do a bitwise logical XOR.

Description: vd[i] ← vs[i] XOR select(i,sel,vt)

Each element of vector *vs* is combined with the corresponding element of vector *vt* in a bitwise logical XOR operation. The result is placed in vector *vd*.

The operands and results are values in integer vector format *fmt*. *sel* selects the values of *vt*[] used for each i. See section C 4 on page C-2 for a description of *fmt/sel* encoding.

This operation is a signal processing operation, no data-dependent exceptions are possible.

The operands must be a value in the specified format. If not, the results are undefined and the values of the operand vectors become undefined.

Operation:

```
StoreFPR(fd, fmt, ValueFPR(fs,fmt) xor ValueFPR(ft,fmt))
```

Exceptions:

Coprocessor Unusable
Reserved Instruction