PA-RISC 2.0 Architecture

This document is an excerpt from PA-RISC 2.0 ARCHITECTURE, by Gerry Kane, published by Prentice Hall PTR, isbn 0-13-182734-0. Print copies of the book can be ordered at www.hp.com/hpbooks.

This document contains:
    Chapter 9: Floating-point Instruction Set
This chapter provides a description of each of the instructions supported by the floating-point coprocessor. The instructions are listed in alphabetical order, according to the name of the instruction - as opposed to the instruction mnemonic.

The Description section of each non-load/store instruction contains a list of the Floating-Point Exceptions which the instruction may cause. Each instruction description has an Exceptions section which lists the processor interruptions that may occur while the instruction is pointed to by the front of the IA queues.

In the following pages, the notation, FPR, refers to floating-point coprocessor registers 0 through 31. FPSR refers to the Floating-point Status Register. Refer to “Instruction Notations” on page xviii for the explanation of the operation section. The mem_load and the mem_store descriptions are located in “Memory Reference Instructions” on page 6-6.
Floating-Point Absolute Value

Format: FABS,fmt r,t

<table>
<thead>
<tr>
<th></th>
<th>0E</th>
<th>r</th>
<th>0</th>
<th>3</th>
<th>0</th>
<th>f</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>t</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0C</th>
<th>r</th>
<th>0</th>
<th>3</th>
<th>fmt</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To perform a floating-point absolute value.

Description: The floating-point register specified by r is copied to the floating-point register specified by t with the sign bit set to 0. This instruction is non-arithmetic and does not cause an invalid operation exception when the sign of a NaN is set to 0.

Floating-point exceptions:
- Unimplemented

Operation: FPR[t]{all_bits_except_sign} ← FPR[r]{all_bits_except_sign};
FPR[t]{sign_bit} ← 0;

Exceptions: Assist emulation trap
Assist exception trap
Floating-Point Add

<table>
<thead>
<tr>
<th>Format:</th>
<th>FADD,fmt r1,r2,t</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0E</th>
<th>r1</th>
<th>r2</th>
<th>0</th>
<th>r2 f</th>
<th>3</th>
<th>0</th>
<th>r1 t</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(52)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(48)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To perform a floating-point addition.

Description: The floating-point registers specified by r1 and r2 are interpreted in the specified format and arithmetically added. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by t.

Floating-point exceptions:
- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Operation: \( \text{FPR}[t] \leftarrow \text{FPR}[r1] + \text{FPR}[r2]; \)

Exceptions: Assist emulation trap
Assist exception trap
Floating-Point Compare

Format: FCMP,fmt,cond r1,r2,cbit /*targeted compare*/
FCMP,fmt,cond r1,r2 /*queued compare*/

(51)

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>y</th>
<th>r2</th>
<th>f</th>
<th>2</th>
<th>0</th>
<th>r1</th>
<th>0</th>
<th>0</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(47)

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>y</th>
<th>fmt</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0C</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To perform a floating-point comparison.

Description: The floating-point registers specified by \( r1 \) and \( r2 \) are interpreted in the specified format and arithmetically compared. A result is determined based on the comparison and the condition, \( \text{cond} \). The condition is encoded in the \( c \) field of the instruction.

There are two types of floating-point compare instructions – targeted compares and queued compares.

- A targeted compare targets a specific bit of the compare array (CA) in the floating-point Status Register (see the second format in Figure 8-6 on page 8-9.) The CA-bit specified by \( cbit \) is set to 1 if the comparison result is true, or set to 0 otherwise. The CA-bit to set is encoded in the \( y \) field of the instruction as \( cbit + 1 \).

- A queued compare updates the compare queue (CQ) in the floating-point Status Register (see the first format in Figure 8-6 on page 8-9) and is specified by omitting the \( cbit \) operand. The CQ field in the floating-point Status Register is shifted right by one bit (discarding the rightmost bit) and the C-bit is copied into CQ\( \{0\} \). Then, if the comparison result is true, the C-bit in the floating-point Status Register is set to 1, otherwise the C-bit is set to 0. A queued compare is encoded with a \( y \) field of 0.

If at least one of the values is a signaling NaN, or if at least one of the values is a NaN and the low-order bit of the condition is 1, an invalid operation exception is signaled.

For unimplemented and trapped invalid operation exceptions, the state of the C-bit and CA field is unchanged, and the CQ field is not shifted.

For untrapped invalid operation exceptions, the state of the C-bit, or the CA-bit specified by the instruction is the AND of the unordered relation (which is true) and bit 3 of the \( c \) field.

Comparisons are exact and neither overflow nor underflow. Four mutually exclusive relations are possible results: less than, equal, greater than, and unordered. The last case arises when at least one operand is a NaN. Every NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so \(+0 = -0\).
Floating-point exceptions:
- Unimplemented
- Invalid operation

Operation:
```c
if (NaN(FPR[r1]) \| NaN(FPR[r2]))
  if (c[4])
    invalid_operation_exception;
  else {
    greater_than \leftarrow false;
    less_than \leftarrow false;
    equal_to \leftarrow false;
    unordered \leftarrow true;
  }
else {
  greater_than \leftarrow FPR[r1] > FPR[r2];
  less_than \leftarrow FPR[r1] < FPR[r2];
  equal_to \leftarrow FPR[r1] = FPR[r2];
  unordered \leftarrow false;
}
if (y) { /*targeted compare*/
  FPSR[CA{y–1}] \leftarrow (((c[0] == 1) \&\& greater_than) \|
    ((c[1] == 1) \&\& less_than) \|
    ((c[2] == 1) \&\& equal_to) \|
    ((c[3] == 1) \&\& unordered));
} else { /*queued compare*/
  FPSR[CQ] \leftarrow rshift(FPSR[CQ],1);
  FPSR[CQ(0)] \leftarrow FPSR[C];
  FPSR[C] \leftarrow (((c[0] == 1) \&\& greater_than) \|
    ((c[1] == 1) \&\& less_than) \|
    ((c[2] == 1) \&\& equal_to) \|
    ((c[3] == 1) \&\& unordered));
}
```

Exceptions:
- Assist emulation trap
- Assist exception trap
### Floating-Point Convert

**Purpose:** To change the value in a floating-point register from one format to a different format.

**Description:** The floating-point register specified by \( r \) is interpreted in the specified source format, \( sf \), and arithmetically converted to the specified destination format, \( df \). The result is placed in the floating-point register specified by \( t \).

The \( sf \) and \( df \) completers specify both the type of conversion and the size of the source and destination formats, and are encoded in the \( sub, sf, \) and \( df \) fields of the instruction (see Table 8-8 on page 8-14 and Table 8-11 on page 8-16.)

If the “\( t \)” (truncate) completer is specified, the current rounding mode is ignored and the result is rounded toward zero. Otherwise, rounding occurs according to the currently specified rounding mode.

**Floating-point exceptions:**
- Unimplemented
- Invalid operation
- Overflow*
- Underflow*
- Inexact

* Not reported unless both source and destination formats are floating-point formats.

**Operation:**

\[
\text{if (truncate)} \\
\quad \text{FPR}[t] \leftarrow \text{convert(FPR}[r],sf,df,ROUND\_TOWARD\_ZERO); \\
\text{else} \\
\quad \text{FPR}[t] \leftarrow \text{convert(FPR}[r],sf,df,FPSR[RM]); \\
\]

**Exceptions:**
- Assist emulation trap
- Assist exception trap

**Restrictions:** The “\( t \)” completer may only be specified with a fixed-point destination format.

Specifying the same source and destination format is an undefined operation.

Specifying any quadword format in the 0E opcode is an undefined operation.
Floating-Point Copy

**Purpose:** To copy a floating-point value to another floating-point register.

**Description:** The floating-point register specified by \( r \) is copied into the floating-point register specified by \( t \). This operation is non-arithmetic and does not cause an invalid operation exception when a NaN is copied.

Floating-point exceptions:
- Unimplemented

**Operation:** \( \text{FPR}[t] \leftarrow \text{FPR}[r]; \)

**Exceptions:**
- Assist emulation trap
- Assist exception trap
Floating-Point Divide

FDIV

Format:  \( \text{FDIV,fmt \ r1,r2,t} \)

(52)

<table>
<thead>
<tr>
<th>0E</th>
<th>r1</th>
<th>r2</th>
<th>3</th>
<th>r2</th>
<th>f</th>
<th>3</th>
<th>0</th>
<th>r1</th>
<th>t</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

(48)

<table>
<thead>
<tr>
<th>0C</th>
<th>r1</th>
<th>r2</th>
<th>3</th>
<th>fmt</th>
<th>3</th>
<th>0</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To perform a floating-point division.

Description: The floating-point registers specified by \( r1 \) and \( r2 \) are interpreted in the specified format and arithmetically divided. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by \( t \).

Floating-point exceptions:
- Unimplemented
- Invalid operation
- Division-by-zero
- Overflow
- Underflow
- Inexact

Operation: \( \text{FPR}[t] \leftarrow \text{FPR}[r1] / \text{FPR}[r2]; \)

Exceptions: Assist emulation trap
Assist exception trap
Floating-Point Identify (FID)

Format: FID

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>OC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Purpose: To validate fields in the Status Register which identify the floating-point coprocessor.

Description: The *model* and *revision* fields in the Status Register become defined. The contents of the other fields in the Status Register are undefined after the execution of this instruction. The *model* and *revision* fields remain defined until a floating-point instruction is executed which is not a double-word store of register 0.

Floating-point exceptions:
- None

Operation: FPSR[model] ← implementation-dependent model number;
FPSR[revision] ← implementation-dependent revision number;

Exceptions: Assist emulation trap

Notes: This instruction must be implemented. Software may use the following sequence to obtain the *model* and *revision* fields in the Status Register:

```
.CODE
LDIL L%freg0,r2 ; load address of
LDO R%freg0(r2),r2 ; fp reg0 save area
FSTD fr0,0(r2) ; save fp reg0, cancel exception traps
FID ; identify coprocessor
LDIL L%version,r2 ; load address of
LDO R%version(r2),r2 ; model/rev save area
FSTD fr0,0(r2) ; store coprocessor id, cancel
; exception traps

.DATA
freg0 .DOUBLE 0
version .DOUBLE 0
```

For the FID instruction to work correctly, the floating-point instructions immediately preceding and following it must be double-word stores of Floating-Point Register 0. If not, the instruction is an undefined operation.

The sequence described will work in user mode. For example, if a context switch occurs just prior to FID but after the first FSTD 0,0(2) instruction, the floating-point state save and state restore sequence will restore the state of the Status Register ("T" bit off, cancel trap) just prior to the execution of FID.
Floating-Point Load Doubleword

**Format:** FLDD, cmplt, cc \( \times |d(s, b), t \)

<table>
<thead>
<tr>
<th></th>
<th>14</th>
<th>b</th>
<th>t</th>
<th>s</th>
<th>im10a</th>
<th>m</th>
<th>a</th>
<th>1</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(41)</td>
<td>0B</td>
<td>b</td>
<td>im5</td>
<td>s</td>
<td>a</td>
<td>1</td>
<td>cc</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(39)</td>
<td>0B</td>
<td>b</td>
<td>x</td>
<td>s</td>
<td>u</td>
<td>0</td>
<td>cc</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Purpose:** To load a doubleword into a floating-point coprocessor register.

**Description:** The aligned doubleword at the effective address is loaded into floating-point register \( t \). The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 39), or a displacement \( d \). The displacement can be either long (Format 3) or short (Format 41). The displacement is encoded into the immediate field. Optional base modification can also be performed.

The completer, \( cmplt \), determines whether the offset is the base register, or the base register plus the index register or displacement. The completer also specifies base register modification, optional index prescaling, and ordering constraints (see Table H-1 on page H-4, and Table H-3 on page H-8 for the assembly language completer mnemonics.) The completer, \( cc \), specifies the cache control hint (see Table 6-7 on page 6-10.)

For long and short displacements, a one in the \( m \) field specifies base modification, and the \( a \) field encodes whether pre-modification (\( a=1 \)), or post-modification (\( a=0 \)) is performed. For indexed loads, a one in the \( m \) field specifies base modification, and a one in the \( u \) field specifies index prescaling.

Specifying Floating-Point Register 0 forces the coprocessor to complete all previous floating-point instructions.
Operation:  
if (indexed_load)  
  switch (cmplt) {  
    case S:  
      case SM:  
        dx ← lshift(GR[x],3);  
        break;  
    case M:  
      default: dx ← GR[x];  
        break;  
  }  
else if (d > 15 || d < -16) {  
  /* long displacement */  
  dx ← sign_ext(assemble_16a(s,cat(im10a,0),i),16); /* (Format 3) */  
  cc ← NO_HINT;  
} else /* short displacement */  
  dx ← low_sign_ext(im5,5);  
  space ← space_select(s,GR[b],format);  
  switch (cmplt) {  
  case MB:  
    offset ← GR[b] + dx;  
    GR[b] ← GR[b] + dx;  
    break;  
  case MA:  
  case M:  
  case SM:  
    offset ← GR[b];  
    GR[b] ← GR[b] + dx;  
    break;  
  default:  
    offset ← GR[b] + dx;  
    break;  
}  
  FPR[t] ← mem_load(space,offset,0,63,cc);  
if (cmplt == O)  
enforce_ordered_load;

Exceptions:  
Assist exception trap  
Unaligned data reference trap  
Data TLB miss fault/data page fault  
Page reference trap  
Data memory access rights trap  
Assist emulation trap  
Data memory protection ID trap

Restrictions:  
For loads with long displacements (Format 3), only displacements which are multiples of eight may be used.  
If the completer O is specified, the displacement must be 0.
Floating-Point Load Word

Format: \( \text{FLDW},\text{cmplt},cc \ x|d(s,b),t \)

<table>
<thead>
<tr>
<th>Format</th>
<th>17 \ b \ t \ s \ im11a \ 0</th>
<th>t</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>6 5 5 2 2 1 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format</th>
<th>16 \ b \ t \ s \ im11a \ a</th>
<th>t</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>6 5 5 2 2 1 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Format | 09 \ b \ im5 \ s \ a \ 1 \ cc \ 0 | 0 | t | m | t |
|--------|-----------------|---|---|---|---|---|
| 41     | 6 5 5 2 1 1 1 1 1 5 |

<table>
<thead>
<tr>
<th>Format</th>
<th>09 \ b \ x \ s \ u \ 0</th>
<th>cc \ 0</th>
<th>0</th>
<th>t</th>
<th>m</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>6 5 5 2 1 1 1 1 1 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Purpose: To load a word into a floating-point coprocessor register.

Description: The aligned word at the effective address is loaded into floating-point register \( t \). The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 39), or a displacement \( d \). The displacement can be either long (Formats 43 and 44) or short (Format 41). The displacement is encoded into the immediate field. Optional base modification can also be performed.

The completer, \( \text{cmplt} \), determines whether the offset is the base register, or the base register plus the index register or displacement. This completer also specifies base register modification, optional index prescaling, and ordering constraints (see Table H-1 on page H-4, and Table H-3 on page H-8 for the assembly language completer mnemonics.)

The completer, \( cc \), specifies the cache control hint (see Table 6-7 on page 6-10.)

For long displacements with base modification, Format 44 is used, and the \( a \) field encodes whether pre-modification \( (a=1) \), or post-modification \( (a=0) \) is performed. For long displacements with no base modification, Format 43 is used.

For short displacements, a one in the \( m \) field specifies base modification, and the \( a \) field encodes whether pre-modification \( (a=1) \), or post-modification \( (a=0) \) is performed. For indexed loads, a one in the \( m \) field specifies base modification, and a one in the \( u \) field specifies index prescaling.

Specifying floating-point registers 0R, 1L, 1R, 2L, 2R, 3L, or 3R is an undefined operation. Specifying Floating-Point Register 0L forces the coprocessor to complete all previous floating-point instructions. However, loading Floating-Point Register 0L with a value that sets the Status Register T-bit to 1 is an undefined operation.
Operation:  
if (indexed_load)  
switch (cmplt) {  
    case S:  
        case SM:  
            dx ← lshift(GR[x],2);  
            break;  
        case M:  
            default:  
                dx ← GR[x];  
                break;  
    }  
else if (d > 15 || d < -16) {  
    /* long displacement */  
    dx ← sign_ext(assemble_16a(s,im11a,i),16);  
    cc ← NO_HINT;  
} else  
    /* short displacement */  
space ← space_select(s,GR[b],format);  
switch (cmplt) {  
    case MB:  
        offset ← GR[b] + dx;  
        GR[b] ← GR[b] + dx;  
        break;  
    case MA:  
    case M:  
    case SM:  
        offset ← GR[b];  
        GR[b] ← GR[b] + dx;  
        break;  
    default:  
        offset ← GR[b] + dx;  
        break;  
}  
FPR[t] ← mem_load(space,offset,0,31,cc);  
if (cmplt == O)  
enforce_ordered_load;

Exceptions:  
Assist exception trap  
Unaligned data reference trap  
Data TLB miss fault/data page fault  
Page reference trap  
Data memory access rights trap  
Assist emulation trap  
Data memory protection ID trap

Restrictions:  
For loads with long displacements (Formats 43 and 44), only displacements which are multiples of four may be used.  
If the completer O is specified, the displacement must be 0.
Floating-Point Multiply

Format: \texttt{FMPY,fmt r1,r2,t}

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E</td>
<td>r1</td>
<td>FPR[t] ← FPR[r1] * FPR[r2];</td>
</tr>
<tr>
<td>0C</td>
<td>r1</td>
<td>Exceptions:</td>
</tr>
</tbody>
</table>

Purpose: To perform a floating-point multiply.

Description: The floating-point registers specified by \( r1 \) and \( r2 \) are interpreted in the specified format and arithmetically multiplied. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by \( t \).

Floating-point exceptions:
- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

Exceptions: Assist emulation trap
Assist exception trap
Floating-Point Multiply/Add

Description:
The floating-point registers specified by rm1 and rm2 are interpreted in the specified format and arithmetically multiplied. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by tm.

The floating-point registers specified by ta and ra are interpreted in the specified format and arithmetically added. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by ta.

The behavior of this instruction is undefined if ra specifies the same register as tm, or if ta specifies the same register as any of rm1, rm2, or tm. The behavior of this instruction is also undefined if ra specifies double-precision register 0 or single-precision register 16L.

Floating-point exceptions:
- Unimplemented
- Invalid operation (see Notes)
- Overflow (see Notes)
- Underflow (see Notes)
- Inexact (see Notes)

Operation:
FPR[tm] ← FPR[rm1] * FPR[rm2];  
FPR[ta] ← FPR[ta] + FPR[ra];

Exceptions:
- Assist emulation trap
- Assist exception trap

Notes:
When operating on single-precision operands, each register field specifies one of registers 16L through 31L, or one of 16R through 31R. See Table 8-17 on page 8-21 for the register specifier encodings.

This instruction can be decomposed into FMPY and FADD and then the full set of floating-point exceptions can be reported (see "Exception Registers" on page 10-1).
Floating-Point Multiply Fused Add

Format: \[ \text{FMPYFADD}, \text{fmt} \quad \text{rm1}, \text{rm2}, \text{ra}, \text{t} \]

(54)

<table>
<thead>
<tr>
<th></th>
<th>2E</th>
<th>rm1</th>
<th>rm2</th>
<th>ra</th>
<th>k2</th>
<th>f</th>
<th>ra</th>
<th>k1</th>
<th>t</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To perform a floating-point multiply and fused add.

Description: The floating-point registers specified by \( \text{rm1} \) and \( \text{rm2} \) are interpreted in the specified format and arithmetically multiplied. The intermediate result is calculated to infinite precision with an unbounded exponent (and is not rounded.) The floating-point register specified by \( \text{ra} \) is interpreted in the specified format, arithmetically added to the result obtained by the multiply operation and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by \( t \).

Floating-point exceptions:
- Unimplemented

Operation: \( \text{FPR}[t] \leftarrow (\text{FPR}[\text{rm1}] \ast \text{FPR}[\text{rm2}]) + \text{FPR}[\text{ra}]; \)

Exceptions: Assist emulation trap
Assist exception trap
Floating-Point Multiply Negate Fused Add

**Format:**

```plaintext
FMPYNFADD,fmt rm1,rm2,ra,t
```

<table>
<thead>
<tr>
<th>rm1</th>
<th>rm2</th>
<th>ra</th>
<th>r2</th>
<th>f</th>
<th>ra</th>
<th>r1</th>
<th>t</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Purpose:** To perform a floating-point multiply, negate, and fused add.

**Description:** The floating-point registers specified by \( rm1 \) and \( rm2 \) are interpreted in the specified format and arithmetically multiplied. The intermediate result is calculated to infinite precision with an unbounded exponent (and is not rounded.) The floating-point register specified by \( ra \) is interpreted in the specified format, arithmetically added to the negated result obtained by the multiply operation and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by \( t \).

Floating-point exceptions:
- Unimplemented

**Operation:**

\[
FPR[t] \leftarrow -(FPR[rm1] \times FPR[rm2]) + FPR[ra];
\]

**Exceptions:**
- Assist emulation trap
- Assist exception trap
Floating-Point Multiply/Subtract

FMPYSUB

Format: FMPYSUB,fmt rm1,rm2,tm,ra,ta

<table>
<thead>
<tr>
<th>Format</th>
<th>rm1</th>
<th>rm2</th>
<th>tm</th>
<th>ra</th>
<th>f</th>
<th>tm</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To perform a floating-point multiply and a floating-point subtract.

Description: The floating-point registers specified by \( rm1 \) and \( rm2 \) are interpreted in the specified format and arithmetically multiplied. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by \( tm \).

The floating-point registers specified by \( ta \) and \( ra \) are interpreted in the specified format and arithmetically subtracted. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by \( ta \).

The behavior of this instruction is undefined if \( ra \) specifies the same register as \( tm \), or if \( ta \) specifies the same register as any of \( rm1 \), \( rm2 \), or \( tm \). The behavior of this instruction is also undefined if \( ra \) specifies double-precision register 0 or single-precision register 16L.

Floating-point exceptions:
- Unimplemented
- Invalid operation (see Notes)
- Overflow (see Notes)
- Underflow (see Notes)
- Inexact (see Notes)

Operation: \( \text{FPR}[tm] \leftarrow \text{FPR}[rm1] \times \text{FPR}[rm2]; \)
\( \text{FPR}[ta] \leftarrow \text{FPR}[ta] - \text{FPR}[ra]; \)

Exceptions: Assist emulation trap
Assist exception trap

Notes: When operating on single-precision operands, each register field specifies one of registers 16L through 31L, or one of 16R through 31R. See Table 8-17 on page 8-21 for the register specifier encodings.

This instruction can be decomposed into FMPY and FSUB and then the full set of floating-point exceptions can be reported (see “Exception Registers” on page 10-1).
### Floating-Point Negate

**FNEG**

**Format:**

FNEG,fmt \( r, t \)

**Purpose:** To negate a floating-point value.

**Description:** The floating-point register specified by \( r \) is copied into the floating-point register specified by \( t \) and negated. This operation is non-arithmetic and does not cause an invalid operation exception when a NaN is negated.

Floating-point exceptions:
- Unimplemented

**Operation:**

\[
\begin{align*}
FPR[t]\{\text{all_bits\_except\_sign}\} & \leftarrow FPR[r]\{\text{all_bits\_except\_sign}\}; \\
FPR[t]\{\text{sign\_bit}\} & \leftarrow \neg FPR[r]\{\text{sign\_bit}\};
\end{align*}
\]

**Exceptions:**
- Assist emulation trap
- Assist exception trap
Floating-Point Negate Absolute Value

Purpose: To negate a floating-point absolute value.

Description: The floating-point register specified by \( r \) is copied into the floating-point register specified by \( t \) with the sign bit set to 1. This operation is non-arithmetic and does not cause an invalid operation exception when the sign of a NaN is set to 1.

Floating-point exceptions:
- Unimplemented

Operation: \[
\text{FPR}[t]\{\text{all\_bits\_except\_sign}\} \leftarrow \text{FPR}[r]\{\text{all\_bits\_except\_sign}\};
\text{FPR}[t]\{\text{sign\_bit}\} \leftarrow 1;
\]

Exceptions: Assist emulation trap
Assist exception trap
## Floating-Point Round to Integer

### Format:

\[
\text{FRND,fmt } r, t
\]

<table>
<thead>
<tr>
<th></th>
<th>0E</th>
<th>r</th>
<th>0</th>
<th>5</th>
<th>0</th>
<th>f</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>t</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0C</th>
<th>r</th>
<th>0</th>
<th>5</th>
<th>fmt</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

### Purpose:

To round a floating-point value to an integral value.

### Description:

The floating-point register specified by \( r \) is interpreted in the specified format and arithmetically rounded to an integral value. This result remains a floating-point number. Results are rounded according to the current rounding mode with the proviso that when rounding to nearest, if the difference between the unrounded operand and the rounded result is exactly one half, the rounded result is even. The result is placed in the floating-point register specified by \( t \). An inexact exception is signaled when the result and source are not the same.

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Inexact

### Operation:

\[
\text{FPR}[t] \leftarrow \text{floating-point_round(FPR}[r]);
\]

### Exceptions:

- Assist emulation trap
- Assist exception trap
Floating-Point Square Root

FSQRT

Format: FSQRT.fmt r,t

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(49)</td>
<td>To perform a floating-point square root.</td>
</tr>
<tr>
<td>(45)</td>
<td>The floating-point register specified by ( r ) is interpreted in the specified format and the positive arithmetic square root is taken. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. If the source register contains (-0), the result will be (-0). The result is placed in the floating-point register specified by ( t ).</td>
</tr>
</tbody>
</table>

Floating-point exceptions:

- Unimplemented
- Invalid operation
- Inexact

Operation: \( \text{FPR}[t] \leftarrow \text{square_root}(\text{FPR}[r]) \); 

Exceptions: Assist emulation trap
Assist exception trap
Floating-Point Store Doubleword

Format: \( \text{FSTD,cmplt,cc \ r|x|d(s,b)} \)

<table>
<thead>
<tr>
<th>(3)</th>
<th>1C</th>
<th>b</th>
<th>t</th>
<th>s</th>
<th>\text{im10a}</th>
<th>m</th>
<th>a</th>
<th>1</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| (42) | 0B | b | \text{im5} | s | a | 1 | cc | 1 | 0 | 0 | m | r |
|      | 6  | 5  | 5  | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 5 |

| (40) | 0B | b | x | s | \text{u} | 0 | cc | 1 | 0 | 0 | m | r |
|      | 6  | 5  | 5  | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 5 |

Purpose: To store a doubleword from a floating-point coprocessor register.

Description: Floating-point register \( r \) is stored in the aligned doubleword at the effective address. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 40), or a displacement \( d \). The displacement can be either long (Format 3) or short (Format 42). The displacement is encoded into the immediate field. Optional base modification can also be performed.

The completer, \( \text{cmplt} \), determines whether the offset is the base register, or the base register plus the index register or displacement. This completer also specifies base register modification, optional index prescaling, and ordering constraints (see Table H-1 on page H-4, and Table H-3 on page H-8 for the assembly language completer mnemonics.) The completer, \( \text{cc} \), specifies the cache control hint (see Table 6-8 on page 6-10.)

For long and short displacements, a one in the \( m \) field specifies base modification, and the \( a \) field encodes whether pre-modification (\( a=1 \)), or post-modification (\( a=0 \)) is performed. For indexed stores, a one in the \( m \) field specifies base modification, and a one in the \( u \) field specifies index prescaling.

Specifying Floating-Point Register 0 forces the coprocessor to complete all previous floating-point instructions and sets the Status Register T-bit to 0 following completion of the store.
Operation: if (indexed_store)
    switch (cmplt) {
        case S: 
        case SM: dx ← shift(GR[x],3); break;
        case M: default: dx ← GR[x]; break;
    }
else if (d > 15 || d < -16) {
    dx ← sign_ext(assemble_16a(s,cat(im10a,0),i),16); /* (Format 3) */
    cc ← NO_HINT;
} else
    dx ← low_sign_ext(im5,5);
space ← space_select(s,GR[b],format);
if (cmplt == O)
    enforce_ordered_store;
switch (cmplt) {
    case MB: offset ← GR[b] + dx; 
    GR[b] ← GR[b] + dx; break;
    case MA: 
    case M: 
    case SM: offset ← GR[b]; 
    GR[b] ← GR[b] + dx; break;
    default: offset ← GR[b] + dx; break;
}
mem_store(space,offset,0,63,cc,FPR[r]);

Exceptions: Assist exception trap                   Data memory break trap
Data TLB miss fault/data page fault              TLB dirty bit trap
Data memory access rights trap                  Page reference trap
Data memory protection ID trap                  Assist emulation trap
Unaligned data reference trap

Restrictions: For stores with long displacements (Format 3), only displacements which are multiples of eight may be used.
If the completer O is specified, the displacement must be 0.
Floating-Point Store Word

Format: \( FSTW, cmplt, cc \times d(s,b) \)

<table>
<thead>
<tr>
<th>Format</th>
<th>IF</th>
<th>b</th>
<th>r</th>
<th>s</th>
<th>im11a</th>
<th>0</th>
<th>r</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>a</td>
<td>r</td>
<td>i</td>
</tr>
<tr>
<td>42</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To store a word from a floating-point coprocessor register.

Description: Floating-point register \( r \) is stored in the aligned word at the effective address. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 40), or a displacement \( d \). The displacement can be either long (Formats 43 and 44) or short (Format 42). The displacement is encoded into the immediate field. Optional base modification can also be performed.

The completer, \( cmplt \), determines whether the offset is the base register, or the base register plus the index register or displacement. This completer also specifies base register modification, optional index prescaling, and ordering constraints (see Table H-1 on page H-4, and Table H-3 on page H-8 for the assembly language completer mnemonics.) The completer, \( cc \), specifies the cache control hint (see Table 6-8 on page 6-10.)

For long displacements with base modification, Format 44 is used, and the \( a \) field encodes whether pre-modification \((a=1)\), or post-modification \((a=0)\) is performed. For long displacements with no base modification, Format 43 is used.

For short displacements, a one in the \( m \) field specifies base modification, and the \( a \) field encodes whether pre-modification \((a=1)\), or post-modification \((a=0)\) is performed. For indexed stores, a one in the \( m \) field specifies base modification, and a one in the \( u \) field specifies index prescaling.

Specifying floating-point registers 0R, 1L, 1R, 2L, 2R, 3L, or 3R is an undefined operation. Specifying Floating-Point Register 0L forces the coprocessor to complete all previous floating-point instructions.
Operation: if (indexed_store) /* indexed (Format 40)*/
    switch (cmplt) {
        case S:
        case SM: dx ← lshift(GR[x],2);
                  break;
        case M:
        default: dx ← GR[x];
                 break;
    }
else if (d > 15 || d < -16) { /* long displacement */
    dx ← sign_ext(assemble_16a(s,im11a,i),16);
    cc ← NO_HINT;
} else /* short displacement */
    dx ← low_sign_ext(im5,5);
space ← space_select(s,GR[b],format);
if (cmplt == O)
    enforce_ordered_store;
switch (cmplt) {
    case MB: offset ← GR[b] + dx;
             GR[b] ← GR[b] + dx;
             break;
    case MA:
    case M:
    case SM:
    default: offset ← GR[b];
             GR[b] ← GR[b] + dx;
             break;
    }
mem_store(space,offset,0,31,cc,FPR[r]);

Exceptions: Assist exception trap Data memory break trap
             Data TLB miss fault/data page fault TLB dirty bit trap
             Data memory access rights trap Page reference trap
             Data memory protection ID trap Assist emulation trap
             Unaligned data reference trap

Restrictions: For stores with long displacements (Formats 43 and 44), only displacements which are
multiples of four may be used.

If the completer O is specified, the displacement must be 0.
**Floating-Point Subtract**

**Format:** 
FSUB,fmt r1,r2,t

<table>
<thead>
<tr>
<th></th>
<th>0E</th>
<th>r1</th>
<th>r2</th>
<th>1</th>
<th>r2</th>
<th>f</th>
<th>3</th>
<th>0</th>
<th>r1</th>
<th>t</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0C</th>
<th>r1</th>
<th>r2</th>
<th>1</th>
<th>fmt</th>
<th>3</th>
<th>0</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To perform a floating-point subtraction.

**Description:** The floating-point registers specified by r1 and r2 are interpreted in the specified format and arithmetically subtracted. The result is calculated to infinite precision and then rounded to the specified format according to the current rounding mode. The result is placed in the floating-point register specified by t.

Floating-point exceptions:
- Unimplemented
- Invalid operation
- Overflow
- Underflow
- Inexact

**Operation:** 
FPR[t] ← FPR[r1] − FPR[r2];

**Exceptions:**
- Assist emulation trap
- Assist exception trap
Floating-Point Test

Format:

<table>
<thead>
<tr>
<th>FTEST</th>
<th>cbit</th>
<th>/<em>targeted test</em>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTEST,cond</td>
<td>/<em>queue test</em>/</td>
<td></td>
</tr>
</tbody>
</table>

(47)

<table>
<thead>
<tr>
<th>0C</th>
<th>0</th>
<th>0</th>
<th>y</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To test the results of one or more earlier comparisons.

Description: The specified condition in the floating-point Status Register is tested. The condition, *cond*, is encoded in the *c* field of the instruction. If the condition is satisfied, then the following instruction is nullified.

There are two types of floating-point test instructions – targeted tests and queue tests.

- A targeted test tests a specific bit of the compare array (CA) in the floating-point Status Register (see the second format in Figure 8-6 on page 8-9.) If the CA-bit specified by *cbit* is 1, the PSW[N] bit is set to 1. The CA-bit to test is encoded in the *y* field of the instruction as xor(*cbit*+1,1). No condition may be specified for a targeted test and the *c* field must be 0.

- A queue test tests for a specific condition in the C-bit and compare queue (CQ) in the floating-point Status Register (see the first format in Figure 8-6 on page 8-9) and is specified by omitting the *cbit* operand. A queue test is encoded with a *y* field of 1.

Floating-point exceptions:

- None

Conditions: For targeted tests, no condition may be specified. For queue tests, the condition is any of the conditions shown in Table 8-13 on page 8-18. When a condition completer is not specified, the “Simple Test” (C == 1) condition is used. The boolean variable “cond_satisfied” in the operation section is set when the specified condition is satisfied.

Operation:

```c
if (y == 1) { /*queue test*/
    if (cond_satisfied)
        PSW[N] ← 1;
} else { /*targeted test*/
    if (FPSR[CA{xor(y,1)}]) { /*test CA{cbit}* /
        PSW[N] ← 1
    }
}
```

Exceptions: Assist emulation trap

Restrictions: It is an undefined operation to mix targeted FCMP instructions with queue FTEST instructions or to mix queue FCMPs with targeted FTESTs. For a targeted FTEST to be defined, an FCMP to the same CA bit must precede it without any intervening queued FCMP. For a queue FTEST to be defined, enough queued FCMPs must be executed to define...
the CQ bits being tested without any intervening targeted FCMPs. Any FTEST may follow a load of the FPSR, because the load defines all of the C-, CA-, and CQ-bits.

Notes: This instruction must be implemented, may not be queued and may not cause any assist exception traps. However, any assist exception traps caused by previous instructions may be taken while this instruction is in the IA queue.
### Fixed-point Multiply Unsigned

**Format:** XMPYU  \( r1, r2, t \)

<table>
<thead>
<tr>
<th></th>
<th>0E</th>
<th>r1</th>
<th>r2</th>
<th>2</th>
<th>r2 0</th>
<th>3</th>
<th>1</th>
<th>r1 0</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(52)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Purpose:** To perform unsigned fixed-point multiplication.

**Description:** The floating-point registers specified by \( r1 \) and \( r2 \) are interpreted as unsigned 32-bit integers and arithmetically multiplied. The unsigned 64-bit result is placed in the floating-point register specified by \( t \).

- Floating-point exceptions:
  - Unimplemented

**Operation:** \( \text{FPR}[t] \leftarrow \text{FPR}[r1] \times \text{FPR}[r2] ; \)

**Exceptions:**
- Assist emulation trap
- Assist exception trap