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 7: Instruction Descriptions
This chapter provides a description of each of the instructions (except floating-point instructions which are described in Chapter 9, “Floating-Point Instruction Set”) supported by the PA-RISC architecture. The instructions are listed in alphabetical order, according to the instruction’s mnemonic.

Figure 7-1 illustrates the information presented in each of the instruction descriptions. The information presented in this figure is for illustrative purposes only and does not represent a valid instruction.

![Figure 7-1. Instruction Description Example](image-url)
Add

Add,cmp,carry,cond r1,r2,t

Format: ADD,cmp,carry,cond r1,r2,t

(8)  

<table>
<thead>
<tr>
<th></th>
<th>r2</th>
<th>r1</th>
<th>c</th>
<th>f</th>
<th>e1</th>
<th>e2</th>
<th>0</th>
<th>d</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To do 64-bit integer addition and conditionally nullify the following instruction.

Description: GR r1 and GR r2 are added. If no trap occurs, the result is placed in GR t. The variable “carry_borrows” in the operation section captures the 4-bit carries resulting from the add operation. The completer, cmp, encoded in the e1 field, specifies whether the carry/borrow bits in the PSW are updated and whether a trap is taken on signed overflow. The completer, carry, encoded in the e2 field, specifies whether the addition is done with carry in.

The following instruction is nullified if the values added satisfy the specified condition, cond. The condition is encoded in the c, d, and f fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a 32-bit signed overflow (d=0) or a 64-bit signed overflow (d=1.) For addition with carry in, the d field encodes whether the word carry (PSW C/B{8}, d=0), or the doubleword carry (PSW C/B{0}, d=1) is used.

The e1 field encodes whether the carry/borrow bits in the PSW are updated and whether a trap is taken on overflow (e1=1: carries updated, no trap, e1=2: carries not updated, no trap, e1=3: carries updated, trap on overflow.) The e2 field encodes whether addition with carry in is performed (e2=0: no carry in, e2=1: addition performed with carry in.) The combination e1=2, e2=1 is not defined. The following table shows the allowed combinations:

<table>
<thead>
<tr>
<th>Completer</th>
<th>Description</th>
<th>e1</th>
<th>e2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>Add</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C or DC</td>
<td>Add with carry/doubleword carry</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>Add logical</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TSV</td>
<td>Add and trap on signed overflow</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>C,TSV or DC,TSV</td>
<td>Add with carry/doubleword carry</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>and trap on signed overflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conditions: The condition is any of the 32-bit add conditions shown in Table D-6 on page D-5 or any of the 64-bit add conditions shown in Table D-7 on page D-6. When a condition completer is not specified, the “never” condition is used. The boolean variable “cond_satisfied” in the operation section is set when the values added satisfy the specified condition.
Operation: switch (carry) {
    case C: res ← GR[r1] + GR[r2] + PSW[C/B]{8}; break;
    case DC: res ← GR[r1] + GR[r2] + PSW[C/B]{0}; break;
    default: res ← GR[r1] + GR[r2]; break;
}
if (cmplt == TSV && overflow)
    overflow_trap;
else {
    GR[t] ← res;
    if (cmplt != 'L')
        PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}

Exceptions: Overflow trap

Notes: When the ,C completer is specified, only 32-bit conditions are available. When the ,DC completer is specified, only 64-bit conditions are available.
Add and Branch

ADD

Format: 

```
ADD,cond,n  r1,r2,target
```

(17)

<table>
<thead>
<tr>
<th>28/2A</th>
<th>r2</th>
<th>r1</th>
<th>c</th>
<th>w1</th>
<th>n</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: 
To add two values and perform an IA-relative branch conditionally based on the values added.

Description: 
GR r1 and GR r2 are added and the result is placed in GR r2. If the values added satisfy the specified condition, *cond*, the word displacement is assembled from the *w* and *w1* fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, *target*, in the assembly language format is encoded in the *w* and *w1* fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ,N completer, encoded in the *n* field of the instruction, specifies nullification.

Conditions: 
The condition, *cond*, is any of the 32-bit add conditions shown in Table D-6 on page D-5 or any of the 64-bit add and branch conditions shown in Table D-8 on page D-6 and is encoded in the *c* and *opcode* fields of the instruction. When the PSW W-bit is 0, only the 32-bit conditions are available. Opcode 28 is used for the 32-bit non-negated add conditions (those with *f* = 0 in Table D-6) and opcode 2A is used for the 32-bit negated add conditions (those with *f* = 1 in Table D-6.) When the PSW W-bit is 1, a subset of the 32-bit and 64-bit conditions are available. Opcode 28 is used for the non-negated conditions (those with *f* = 0 in Table D-8) and opcode 2A is used for the negated conditions (those with *f* = 1 in Table D-8.) When a condition completer is not specified, the “never” condition is used. The boolean variable “cond_satisfied” in the operation section is set to 1 when the values added satisfy the specified condition and set to 0 otherwise.

Operation: 

```
GR[r2] ← GR[r1] + GR[r2];
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
   IAOQ.Next ← IAOQ.Front + disp + 8;
if (n)
   if (disp < 0)
      PSW[N] ← !cond_satisfied;
   else
      PSW[N] ← cond_satisfied;
```

Exceptions: 
Taken branch trap
Add to Immediate

Format: ADDI,cmplt,trapc,cond i,r,t

(9)

<table>
<thead>
<tr>
<th></th>
<th>2C/2D</th>
<th>r</th>
<th>t</th>
<th>c</th>
<th>f</th>
<th>e1</th>
<th>im11</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>11</td>
</tr>
</tbody>
</table>

Purpose: To add an immediate value to a register and conditionally nullify the following instruction.

Description: The sign-extended immediate value \( i \) is added to GR \( r \). If no trap occurs, the result is placed in GR \( t \) and the carry/borrow bits in the PSW are updated. The immediate value is encoded into the \( \text{im11} \) field. The variable “carry_borrows” in the operation section captures the 4-bit carries resulting from the add operation.

The completer, \( \text{cmplt} \), encoded in the \( e1 \) field, specifies whether a trap is taken on a 32-bit signed overflow (\( e1=0 \): no trap, \( e1=1 \): trap on 32-bit signed overflow.) The completer, \( \text{trapc} \), encoded in the opcode, specifies whether a trap is taken if the values added satisfy the condition specified (no trap for opcode 2D, trap on condition for opcode 2C.) The table below shows the \( \text{cmplt} \) and \( \text{trapc} \) combinations.

<table>
<thead>
<tr>
<th>Completer</th>
<th>Description</th>
<th>Opcode</th>
<th>e1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>Add to immediate</td>
<td>2D</td>
<td>0</td>
</tr>
<tr>
<td>TSV</td>
<td>Add to immediate and trap on signed overflow</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>TC</td>
<td>Add to immediate and trap on condition</td>
<td>2C</td>
<td>0</td>
</tr>
<tr>
<td>TSV, TC</td>
<td>Add to immediate and trap on signed overflow</td>
<td>2C</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>or condition</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For opcode 2D, the following instruction is nullified if the values added satisfy the specified condition, \( \text{cond} \). The condition is encoded in the \( c \) and \( f \) fields of the instruction. The boolean variable “overflow” in the operation section is set if the operation results in a 32-bit signed overflow.

Conditions: The condition is any of the 32-bit add conditions shown in Table D-6 on page D-5. When a condition completer is not specified, the “never” condition is used. The boolean variable “\( \text{cond\_satisfied} \)” in the operation section is set when the values added satisfy the specified condition.

Operation:

\[
\text{res} \leftarrow \text{low\_sign\_ext}(\text{im11},11) + \text{GR}[r];
\]

if (cmplt == TSV && overflow)

overflow_trap;

else if (trapc == TC && \( \text{cond\_satisfied} \))

conditional_trap;

else {

GR[t] \leftarrow \text{res};

PSW[C/B] \leftarrow \text{carry\_borrows};

if (\( \text{cond\_satisfied} \)) PSW[N] \leftarrow 1;

}

Exceptions: Overflow trap Conditional trap
Add Immediate and Branch

ADDIB

Format: ADDIB,cond,n i,r,target

<table>
<thead>
<tr>
<th>29/2B</th>
<th>r</th>
<th>im5</th>
<th>c</th>
<th>w1</th>
<th>n</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To add two values and perform an IA-relative branch conditionally based on the values added.

Description: The sign-extended immediate value \( im5 \) is added to \( GR[r] \) and the result is placed in \( GR[r] \). If the values added satisfy the specified condition, \( cond \), the word displacement is assembled from the \( w \) and \( w1 \) fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, \( target \), in the assembly language format is encoded in the \( w \) and \( w1 \) fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The \( ,N \) completer, encoded in the \( n \) field of the instruction, specifies nullification.

Conditions: The condition, \( cond \), is any of the 32-bit add conditions shown in Table D-6 on page D-5 or any of the 64-bit add and branch conditions shown in Table D-8 on page D-6 and is encoded in the \( c \) and \( opcode \) fields of the instruction. When the PSW W-bit is 0, only the 32-bit conditions are available. Opcode 29 is used for the 32-bit non-negated add conditions (those with \( f = 0 \) in Table D-6) and opcode 2B is used for the 32-bit negated add conditions (those with \( f = 1 \) in Table D-6.) When the PSW W-bit is 1, a subset of the 32-bit and 64-bit conditions are available. Opcode 29 is used for the non-negated conditions (those with \( f = 0 \) in Table D-8) and opcode 2B is used for the negated conditions (those with \( f = 1 \) in Table D-8.) When a condition completer is not specified, the “never” condition is used. The boolean variable “\( cond \_satisfied \)” in the operation section is set to 1 when the values added satisfy the specified condition and set to 0 otherwise.

Operation: \[
GR[r] \leftarrow \text{low\_sign\_ext}(im5,5) + GR[r]; \\
disp \leftarrow \text{lshift}(\text{sign\_ext}(\text{assemble\_12}(w1,w),12),2); \\
\text{if } (\text{cond\_satisfied}) \\
\quad \text{IAOQ\_Next} \leftarrow \text{IAOQ\_Front} + \text{disp} + 8; \\
\text{if } (n) \\
\quad \text{if } (\text{disp} < 0) \\
\quad \quad \text{PSW}[N] \leftarrow \neg \text{cond\_satisfied}; \\
\quad \text{else} \\
\quad \quad \text{PSW}[N] \leftarrow \text{cond\_satisfied};
\]

Exceptions: Taken branch trap
Add Immediate Left

ADDIL i,r,r1

Format: ADDIL i,r,r1

<table>
<thead>
<tr>
<th></th>
<th>0A</th>
<th>r</th>
<th>im21</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

Purpose: To add the upper portion of a 32-bit immediate value to a general register.

Description: The 21-bit immediate value, i, is assembled, shifted left 11 bits, sign extended, added to GR r and placed in GR1. Overflow, if it occurs, is ignored.

Operation: GR[1] ← sign_ext(lshift(assemble_21(im21),11),32) + GR[r];

Exceptions: None

Programming Note
ADD IMMEDIATE LEFT can be used to perform a load or store with a 32-bit displacement. For example, to load a word from memory into general register t with a 32-bit displacement, the following sequence of assembly language code could be used:

ADDIL 1%literal,GRb
LDW  r%literal(0,GR1),GRt
AND

Format: AND,cond r1,r2,t

<table>
<thead>
<tr>
<th></th>
<th>02</th>
<th>r2</th>
<th>r1</th>
<th>c</th>
<th>f</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>d</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To do a 64-bit, bitwise AND.

Description: GR r1 and GR r2 are ANDed and the result is placed in GR t. The following instruction is nullified if the values ANDed satisfy the specified condition, cond. The condition is encoded in the c, d, and f fields of the instruction.

Conditions: The condition is any of the 32-bit logical conditions shown in Table D-9 on page D-7 or any of the 64-bit logical conditions shown in Table D-10 on page D-7. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ANDed satisfy the specified condition.

Operation: GR[t] ← GR[r1] & GR[r2];
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None
AND Complement

Format: ANDCM,cond r1,r2,t

(8)

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Purpose: To do a 64-bit bitwise AND with complement.

Description: GR r1 is ANDed with the one’s complement of GR r2 and the result is placed in GR t. The following instruction is nullified if the values ANDed satisfy the specified condition, cond. The condition is encoded in the c, d, and f fields of the instruction.

Conditions: The condition is any of the 32-bit logical conditions shown in Table D-9 on page D-7 or any of the 64-bit logical conditions shown in Table D-10 on page D-7. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ANDed satisfy the specified condition.

Operation: GR[t] ← GR[r1] & GR[r2];
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None
Branch

Format: \texttt{B,cmplt,stack,n target,t}

<table>
<thead>
<tr>
<th>3A</th>
<th>t/w3</th>
<th>w1</th>
<th>0/1/4/5</th>
<th>w2</th>
<th>n</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To do IA-relative branches with optional privilege level change and procedure calls with a static displacement.

Description: The word displacement is assembled from the \texttt{w}, \texttt{w1}, \texttt{w2}, and (when the \texttt{L} completer is specified with GR 2 as the link register) \texttt{w3} fields in the instruction. The displacement is sign extended, and the result plus 8 is added to the offset of the current instruction to form the target offset.

The completer, \texttt{stack}, specifies whether the offset of the return point is pushed onto the branch target stack. If the \texttt{L} completer is specified and the \texttt{PUSH} completer is specified, either the offset of the return point or an “invalid” value is pushed onto the branch target stack. On machines that do not implement the branch target stack, the instruction executes the same as if the \texttt{PUSH} completer had not been specified.

The completer, \texttt{cmplt}, specifies whether a return link is saved, or whether a privilege-increasing GATEWAY function is performed. If the \texttt{L} completer is specified, the offset of the return point is placed in GR \texttt{t}. The return point is 4 bytes beyond the following instruction.

If the \texttt{GATE} completer is specified and the PSW C-bit is 1, the privilege level is changed to that given by the two rightmost bits of the type field in the TLB entry for the page (when the type field is greater than 3) from which the BRANCH instruction is fetched if that results in a higher privilege. If privilege is not increased, then the current privilege is used at the target. In all cases, the privilege level of the BRANCH instruction is deposited into bits 62..63 of GR \texttt{t}. The privilege change occurs at the target of the BRANCH. If the PSW C-bit is 0, the privilege level is changed to 0. An illegal instruction trap is taken if a BRANCH instruction is attempted with the \texttt{GATE} completer and the PSW B-bit is 1.

If the \texttt{GATE} completer is specified, sub-opcode 1 is used. If the \texttt{L} completer is specified and the target register is GR 2, sub-opcode 5 is used. If the \texttt{L} completer is specified, the target register is GR 2, and the \texttt{PUSH} completer is specified, sub-opcode 4 is used. Otherwise, sub-opcode 0 is used.

The variable “page_type” is set to the value of the access rights field, bits \texttt{0..2}, from the translation used to fetch the instruction. See “Access Rights Interpretation” on page 3-14.

The following instruction is nullified if the \texttt{N} completer is specified. The completer is encoded in the \texttt{n} field of the instruction. The branch target, \texttt{target}, in the assembly language format is encoded in the \texttt{w}, \texttt{w1}, \texttt{w2}, and (when GR 2 is the link register) \texttt{w3} fields.
Operation:
if (cmplt == ‘GATE’ && PSW[B])
    illegal_instruction_trap;
else {
    if (cmplt == ‘L’ && t == GR2) {
        disp ← lshift(sign_ext(assemble_22(w3,w1,w2,w),22),2)
        if(stack == ‘PUSH’)
            push_onto_BTS(( IAOQ_Back + 4){0..61});
    }
    else
        disp ← lshift(sign_ext(assemble_17(w1,w2,w),17),2);
    if (cmplt == ‘GATE’) {
        GR[t] ← cat(GR[t]{0..61},IAOQ_Front{62..63});
        if (PSW[C]) {
            if (page_type <= 3)
                priv ← IAOQ_Front{62..63};
            else
                priv ← min(IAOQ_Front{62..63}, page_type{1..2});
        } else
            priv ← 0;
    } else
        priv ← IAOQ_Front{62..63};
    IAOQ_Next{0..61} ← (IAOQ_Front + disp + 8){0..61};
    IAOQ_Next{62..63} ← priv;
    if (cmplt == ‘L’) {
        GR[t] ← IAOQ_Back + 4;
        BNR ← ( IAOQ_Back + 4){0..61};
    }
    if (n) PSW[N] ← 1;
}

Exceptions: Illegal instruction trap
Taken branch trap

Notes: When the .GATE completer is specified, the privilege information must be captured when the TLB is read for instruction fetch and that information kept for the determination of the new execution privilege.

To perform an unconditional branch without saving a link, the B,n target pseudo-operation generates a a B,L,n target,%R0 instruction with GR0 as the link register.

The CALL,n target pseudo-operation generates a B,L,n target,%R2 instruction to perform a procedure call with GR2 specified as the link register.

Restrictions: The .PUSH completer can be used only if the ,L completer is specified and the target register is GR2.
Programming Note

It is possible for a BRANCH to promote the privilege level so that the process cannot continue executing on that page (because it violates PL2 of the TLB access rights field.) In that case, software should ensure that the BRANCH nullifies execution of the following instruction and its target should be on a page whose range of execute levels includes the new privilege level. Otherwise, an instruction memory protection trap may result.
Branch on Bit

Format: \( BB, \text{cond}, n \quad r, \text{pos}, \text{target} \)

(18)

\[
\begin{array}{cccccc}
\text{bit} & 30/31 & p & r & c & d & w1 & n & w \\
\hline
6 & 5 & 5 & 1 & 1 & 11 & 11 & 1 & 1
\end{array}
\]

Purpose: To perform an IA-relative branch conditionally based on the value of a bit in a register.

Description: If the bit in GR \( r \) specified by \( \text{pos} \) satisfies the condition, \( \text{cond} \), the word displacement is assembled from the \( w \) and \( w1 \) fields of the instruction, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, \( \text{target} \), in the assembly language format is encoded in the \( w \) and \( w1 \) fields.

The bit position, \( \text{pos} \), can either be a constant (fixed bit position, opcode 31), or can be SAR, the Shift Amount Register (CR 11) (variable bit position, opcode 30.)

With a fixed bit position, the \( p \) field encodes the lower 5 bits of \( \text{pos} \), and the \( d \) field encodes either 0 or the complement of the upper bit. If a word condition is specified (either < or \( \geq \)), \( \text{pos} \) may take on the values 0..31, and the bit tested is one of the bits in the lower word of GR \( r \). For word conditions, the \( d \) field is 0. If a doubleword condition is specified (either \( \ast < \) or \( \ast \geq \)), \( \text{pos} \) may take on the values 0..63, and the complement of the upper bit of \( \text{pos} \) is encoded in the \( d \) field. Any bit in the doubleword in GR \( r \) may be tested.

With a variable bit position, the \( p \) field is 0. If a word condition is specified (either < or \( \geq \)), the leftmost bit of the SAR is ignored, and 32 is added to the value in the lower 5 bits. Thus, the bit tested is one of the bits in the lower word of GR \( r \). If a doubleword condition is specified (either \( \ast < \) or \( \ast \geq \)), the full value of the SAR is used. Any bit in the doubleword in GR \( r \) may be tested. For word conditions, the \( d \) field is 0; for doubleword conditions, the \( d \) field is 1.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The \( ,N \) completer, encoded in the \( n \) field of the instruction, specifies nullification.

Conditions: The condition, \( \text{cond} \), is any of the branch on bit conditions from Table D-15 on page D-9. The boolean variable “\text{cond}satisfied” in the operation section is set to 1 when the bit tested satisfies the specified condition and set to 0 otherwise.
Operation: if (variable_bit_position)
  if (cond == < || cond == >=) /* word conditions */
    shamt ← CR[11][1..5] + 32;
  else /* doubleword conditions */
    shamt ← CR[11];
else
  shamt ← cat(~cp,p);
lshift(GR[r],shamt);
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
  IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
  if (disp < 0)
    PSW[N] ← !cond_satisfied;
  else
    PSW[N] ← cond_satisfied;

Exceptions: Taken branch trap
Branch External

Format: 

- **BE,n**: \( wd(sr,b) \)
- **BE,L,n**: \( wd(sr,b),sr0,r31 \)

<table>
<thead>
<tr>
<th>38/39</th>
<th>b</th>
<th>w1</th>
<th>s</th>
<th>w2</th>
<th>n</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To do procedure calls, branches and returns to another space.

Description: The word displacement, \( wd \), is assembled from the \( w, w1, \) and \( w2 \) fields in the instruction. The displacement is sign extended and added to GR \( b \) to form the target offset. SR \( sr \) (which is assembled from the \( s \) field of the instruction) becomes the target space.

If the \( ,L \) completer is specified, the offset of the return point is placed in GR 31 and the space of the return point is placed in SR 0. The return point is 4 bytes beyond the following instruction. If the \( ,L \) completer is specified, opcode 39 is used; otherwise opcode 38 is used.

If the two rightmost bits of GR \( b \) designate a lower privileged level than the current instruction, the privilege level of the target is set to that specified by the rightmost bits of GR \( b \). The decrease in privilege level takes effect at the branch target.

When a BRANCH EXTERNAL is executed with the PSW C-bit 0 (code address translation is disabled) the effect on IASQ (and SR 0 if the \( ,L \) completer is specified) is not defined.

The following instruction is nullified if the \( ,N \) completer is specified. The completer is encoded in the \( n \) field of the instruction.

Operation:

\[
\begin{align*}
\text{disp} & \leftarrow \text{lshift}(\text{sign_ext}(\text{assemble}_17(w1,w2,w),17),2); \\
\text{IAOQ\_Next}{0..61} & \leftarrow (\text{GR}[b] + \text{disp}){0..61}; \\
\text{if} \ (\text{IAOQ\_Front}{62..63} < \text{GR}[b]{62..63}) \\
& \quad \text{IAOQ\_Next}{62..63} \leftarrow \text{GR}[b]{62..63}; \\
\text{else} \\
& \quad \text{IAOQ\_Next}{62..63} \leftarrow \text{IAOQ\_Front}{62..63}; \\
\text{IASQ\_Next} & \leftarrow \text{SR}[\text{assemble}_3(s)]; \\
\text{if} \ (\text{cmplt} == \text{L}) \ {\{ \\
& \quad \text{GR}[31] \leftarrow \text{IAOQ\_Back} + 4; \\
& \quad \text{SR}[0] \leftarrow \text{IASQ\_Back}; \\
\text{\}} \text{if} \ (n) \ \text{PSW}[N] \leftarrow 1;
\end{align*}
\]

Exceptions: Taken branch trap
Programming Note
If a taken local branch is executed following a BRANCH EXTERNAL instruction, the target’s address is computed based on the value of the IASQ set by the BRANCH EXTERNAL instruction. This results in a transfer of control to possibly a meaningless location in the new space.
Branch and Link Register

**Format:** \( \text{BLR},n \ x,t \)

<table>
<thead>
<tr>
<th>n</th>
<th>2</th>
<th>0</th>
<th>t</th>
<th>x</th>
<th>3A</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>3A</td>
</tr>
</tbody>
</table>

**Purpose:** To do IA-relative branches with a dynamic displacement and store a return link.

**Description:** The index from GR \( x \) is shifted left 3 bits and the result plus 8 is added to the offset of the current instruction to form the target offset. The offset of the return point is placed in GR \( t \). The return point is 4 bytes beyond the following instruction.

The following instruction is nullified if the \( .N \) completer is specified. The completer is encoded in the \( n \) field of the instruction.

**Operation:**

\[
\text{IAOQ\_Next} \leftarrow \text{IAOQ\_Front} + \text{lshift} (\text{GR}[x], 3) + 8;
\]

\[
\text{GR}[t] \leftarrow \text{IAOQ\_Back} + 4;
\]

\[
\text{if (n)} \quad \text{PSW}[N] \leftarrow 1;
\]

**Exceptions:** Taken branch trap

---

**Programming Note**

BRANCH AND LINK REGISTER with GR 0 as the link register does a IA-relative branch without saving a link. Jump tables based on the index value can be constructed using this instruction. When the jump table begins at the instruction which is located at the BLR plus 8 bytes, an index value of 0 can be used to branch to the first entry of the table.
**Break**

<table>
<thead>
<tr>
<th>Format:</th>
<th>BREAK</th>
<th>im5,im13</th>
</tr>
</thead>
<tbody>
<tr>
<td>(27)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>im13</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To cause a break instruction trap for debugging purposes.

**Description:** A break instruction trap occurs when this instruction is executed.

**Operation:** break_instruction_trap;

**Exceptions:** None

**Notes:** *im5* and *im13* can be used as parameters to the "BREAK" processing code.
Branch Vectored

Format: \texttt{BV,n x(b)}

(21) \begin{tabular}{ccccccc}
3A & b & x & 6 & 0 & n & 0 \\
\hline
6 & 5 & 5 & 3 & 11 & 1 & 1 \\
\end{tabular}

Purpose: To do base-relative branches with a dynamic displacement in the same space.

Description: The index from GR $x$ is shifted left 3 bits and the result is added to GR $b$ to form the target offset.

The following instruction is nullified if the \texttt{.N} completer is specified. The completer is encoded in the $n$ field of the instruction.

If the two rightmost bits of GR $b$ designate a lower privilege level than the current privilege level, the privilege level of the target is set to that specified by the rightmost bits of GR $b$. The decrease in privilege level takes effect at the branch target.

Operation: \begin{verbatim}
IAOQ\_Next\{0..61\} \leftarrow (GR[b] + \text{lshift(GR}[x],3))\{0..61\};
if (IAOQ\_Front\{62..63\} < GR[b]\{62..63\})
    IAOQ\_Next\{62..63\} \leftarrow GR[b]\{62..63\};
else
    IAOQ\_Next\{62..63\} \leftarrow IAOQ\_Front\{62..63\};
if (n) PSW[N] \leftarrow 1;
\end{verbatim}

Exceptions: Taken branch trap
Branch Vectored External

<table>
<thead>
<tr>
<th>Format</th>
<th>BVE,stack,n (b)</th>
<th>BVE,L,stack,n (b),r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(22)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3A</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To do base-relative branches and procedure calls to another space.

Description: Either GR b or the branch target stack provides the offset of the target instruction.

The completer, stack, specifies whether a branch target stack operation is performed. If no completer is specified, GR b provides the target offset and the branch target stack is not changed.

If the ,POP completer is specified and if the branch target stack is non-empty and the top entry is valid, the target offset can be provided by either GR b or the top entry of the branch target stack. If the ,POP completer is specified, the top entry of the branch target stack is popped. If the ,POP completer is specified and the top entry of the branch target stack is valid and does not equal the value in GR b, the results are undefined. On machines that do not implement the branch target stack, GR b provides the target offset.

If the ,L completer is specified and the ,PUSH completer is specified, either the offset of the return point or an “invalid” value is pushed onto the branch target stack. On machines that do not implement the branch target stack, the instruction executes the same as if the ,PUSH completer had not been specified.

If a stack completer is specified, the p field is 1. Otherwise the p field is 0.

If the ,L completer is specified, the offset of the return point is placed in GR 2. The return point is 4 bytes beyond the following instruction. The completer is encoded in the subopcode field of the instruction (6: no link, 7: link.) The space of the target instruction is specified implicitly by the base register. The upper two bits of GR b are added to 4 to select a space register which gives the target space.

The following instruction is nullified if the ,N completer is specified. The completer is encoded in the n field of the instruction.

If the two rightmost bits of GR b designate a lower privilege level than the current privilege level, then the privilege level of the target is set to that specified by the rightmost bits of GR b. The decrease in privilege level takes effect at the branch target.

When a BRANCH VECTORED EXTERNAL is executed with the PSW C-bit 0 (code address translation is disabled) the effects on IASQ are not defined.
Operation: if (stack == ‘POP’) {
    tmp ← pop_from_BTS();
    valid ← tmp{62};
    if (valid)
        IAOQ_Next{0..61} ← tmp{0..61};
    else
        IAOQ_Next{0..61} ← GR[b]{0..61};
}
if (IAOQ_Front{62..63} < GR[b]{62..63})
    IAOQ_Next{62..63} ← GR[b]{62..63};
else
    IAOQ_Next{62..63} ← IAOQ_Front{62..63};
if (cmplt == L) {
    GR[2] ← IAOQ_Back + 4;
    BNR ← (IAOQ_Back + 4){0..61};
    if (stack == ‘PUSH’)
        push_onto_BTS((IAOQ_Back + 4){0..61});
}
IASQ_Next ← space_select(0, GR[b], LONG_DISP);
if (n) PSW[N] ← 1;

Exceptions: Taken branch trap.

Notes: The CALL,n (b) pseudo-operation generates a BVE,L,n (b),%R2 instruction to perform an indirect procedure call with GR2 specified as the link register.

The RET,n pseudo-operation generates a BVE,n (%R2) instruction to perform a procedure return.
Coprocessor Load Doubleword  

**Format:**  
CLDD,uid,cmplt,cc d(s,b),t  

<table>
<thead>
<tr>
<th>(41)</th>
<th>0B</th>
<th>b</th>
<th>im5</th>
<th>s</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>0</th>
<th>uid</th>
<th>m</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(39)</th>
<th>0B</th>
<th>b</th>
<th>x</th>
<th>s</th>
<th>u</th>
<th>cc</th>
<th>0</th>
<th>uid</th>
<th>m</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To load a doubleword into a coprocessor register.

**Description:** The aligned doubleword at the effective address is loaded into register \( t \) of the coprocessor identified by \( uid \). The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 39), or a displacement \( d \) (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-3 on page H-8, and Table H-1 on page H-4 for the assembly language completer mnemonics.) The completer, \( cc \), specifies the cache control hint (see Table 6-7 on page 6-10.)

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.
Operation: if (indexed_load)
    switch (cmplt) {
        case S:
        case SM:
            dx ← lshift(GR[x],3);
            break;
        case M:
        default:
            dx ← GR[x];
            break;
    }
} 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;
    }
    send_to_copr(uid,t);
    CPR[uid][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  Data memory protection ID trap
             Assist emulation trap

Restrictions: If the completer $O$ is specified, the displacement must be 0.
**Coprocessor Load Word**

**Format:** CLDW,uid,cmplt,cc \( x|d(s,b),t \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>s</th>
<th>a</th>
<th></th>
<th></th>
<th>uid</th>
<th>m</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>09</td>
<td>09</td>
<td>b</td>
<td>im5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(41)</td>
<td>(39)</td>
<td>cc</td>
<td>0</td>
<td>uid</td>
<td>m</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:** To load a word into a coprocessor register.

**Description:** The aligned word at the effective address is loaded into register \( t \) of the coprocessor identified by \( uid \). The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 39), or a displacement \( d \) (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-3 on page H-8, and Table H-1 on page H-4 for the assembly language completer mnemonics.) The completer, \( cc \), specifies the cache control hint (see Table 6-7 on page 6-10.)

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.
Operation:  
if (indexed_load)  
    switch (cmplt) {  
        case S:  
        case SM:  
            dx ← lshift(GR[x],2);  
            break;  
        case M:  
        default:  
            dx ← GR[x];  
            break;  
    }  
} else  
    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;  
}  
send_to_copr(uid,t);  
CPR[uid][t] ← mem_load(space,offset,0,31,cc);  
if (cmplt == O)  
    enforce_ordered_load;

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

Restrictions:  
If the completer O is specified, the displacement must be 0.
Clear Branch Target Stack

Format: CLRBTS

(23)

<table>
<thead>
<tr>
<th></th>
<th>3A</th>
<th>0</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To clear the branch target stack.

Description: The branch target stack is cleared, either by making it empty or by invalidating all entries. If this instruction is nullified, the results are undefined. This instruction is executed as a NOP on machines that do not implement the branch target stack.

Operation: clear_BTS();

Exceptions: none
Compare and Branch

**CMPB**

<table>
<thead>
<tr>
<th>Format:</th>
<th>CMPB,cond,n r1,r2,target</th>
</tr>
</thead>
<tbody>
<tr>
<td>(17)</td>
<td>20/22/27/2F r2 r1 c w1 n w</td>
</tr>
<tr>
<td></td>
<td>6 5 5 3 11 1 1</td>
</tr>
</tbody>
</table>

**Purpose:** To compare two values and perform an IA-relative branch conditionally based on the values compared.

**Description:** GR $r1$ is compared with GR $r2$. If the values compared satisfy the specified condition, $cond$, the word displacement is assembled from the $w$ and $w1$ fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, $target$, in the assembly language format is encoded in the $w$ and $w1$ fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The $N$ completer, encoded in the $n$ field of the instruction, specifies nullification.

**Conditions:** The condition, $cond$, can be any of the 32-bit compare or subtract conditions shown in Table D-3 on page D-4 or any of the 64-bit compare or subtract conditions shown in Table D-4 on page D-4 and is encoded in the $c$ and $ opcode$ fields of the instruction. Opcode 20 is used for the 32-bit non-negated conditions (those with $f = 0$ in Table D-3), opcode 22 for the 32-bit negated conditions (those with $f = 1$ in Table D-3), opcode 27 for the 64-bit non-negated conditions (those with $f = 0$ in Table D-4), and opcode 2F for the 64-bit negated conditions (those with $f = 1$ in Table D-4.) When a condition completer is not specified, the “never” condition is used. The boolean variable “cond_satisfied” in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

**Operation:**

1. $GR[r1] + \sim GR[r2] + 1$;
2. $disp \leftarrow \text{lshift}(	ext{sign_ext(assemble}_{12}(w1, w), 12), 2)$;
3. if (cond_satisfied)
   - $IAOQ\_Next \leftarrow IAOQ\_Front + disp + 8$;
4. if (n)
   - if (disp < 0)
     - $PSW[N] \leftarrow \neg \text{cond_satisfied}$;
   - else
     - $PSW[N] \leftarrow \text{cond_satisfied}$;

**Exceptions:** Taken branch trap
## Compare and Clear  

**CMPCLR**

<table>
<thead>
<tr>
<th>Format:</th>
<th>CMPCLR,cond r1,r2,t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>02  r2  r1  c  f  2  0  0  2  d  t</td>
</tr>
<tr>
<td></td>
<td>6   5   5   3  1  2  1  1  2  1  5</td>
</tr>
</tbody>
</table>

**Purpose:** To compare two registers, set a register to 0, and conditionally nullify the following instruction, based on the result of the comparison.

**Description:** GR \(r1\) and GR \(r2\) are compared and GR \(t\) is set to zero. The following instruction is nullified if the values compared satisfy the specified condition, \(\text{cond}\). The condition is encoded in the \(c\), \(d\), and \(f\) fields of the instruction.

**Conditions:** The condition is any of the 32-bit compare or subtract conditions shown in Table D-3 on page D-4 or any of the 64-bit compare or subtract conditions shown in Table D-4 on page D-4. When a condition completer is not specified, the “never” condition is used. The boolean variable "\(\text{cond} \_ \text{satisfied}\)" in the operation section is set when the values compared satisfy the specified condition.

**Operation:**

\[
\begin{align*}
\text{GR}[r1] + &- \text{GR}[r2] + 1; \\
\text{GR}[t] &\leftarrow 0; \\
\text{if (cond\_satisfied) PSW}[N] &\leftarrow 1;
\end{align*}
\]

**Exceptions:** None

### Programming Note

COMPARE AND CLEAR can be used to produce the logical value of the result of a comparison (assuming false is represented by 0 and true by 1) in a register. The following example will set \(ra\) to 1 if \(rb\) and \(rc\) are equal, and to 0 if they are not equal:

```
CMPCLR,<> rb,rc,ra
LDO 1(0),ra
```
Compare Immediate and Branch

Format: CMPIB,cond,n i,r,target

<table>
<thead>
<tr>
<th>Format</th>
<th>CMPIB, cond, n</th>
<th>i, r, target</th>
</tr>
</thead>
<tbody>
<tr>
<td>(17)</td>
<td>21/23/3B</td>
<td>r, im5, c, w1, n, w</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To compare two values and perform an IA-relative branch conditionally based on the values compared.

Description: The sign-extended immediate value im5 is compared with GR r. If the values compared satisfy the specified condition, cond, the word displacement is assembled from the w and w1 fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The branch target, target, in the assembly language format is encoded in the w and w1 fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ,N completer, encoded in the n field of the instruction, specifies nullification.

Conditions: The condition, cond, can be any of the 32-bit compare or subtract conditions shown in Table D-3 on page D-4 or any of the 64-bit compare immediate and branch conditions shown in Table D-5 on page D-5 and is encoded in the c and opcode fields of the instruction. Opcode 21 is used for the 32-bit non-negated conditions (those with f = 0 in Table D-3), opcode 23 is used for the 32-bit negated conditions (those with f = 1 in Table D-3), and opcode 3B is used for the 64-bit conditions (those in Table D-5.) When a condition completer is not specified, the “never” condition is used. The boolean variable “cond_satisfied” in the operation section is set to 1 when the values compared satisfy the specified condition and set to 0 otherwise.

Operation:

\[
\text{low\_sign\_ext(im5,5) + } \overline{\text{GR}[r]} + 1;
\]
\[
\text{disp } \leftarrow \text{lsift(sign\_ext(assemble\_12(w1,w),12),2)};
\]
\[
\text{if (cond\_satisfied)}
\]
\[
\text{IAOQ\_Next } \leftarrow \text{IAOQ\_Front + disp + 8};
\]
\[
\text{if (n)}
\]
\[
\text{if (disp < 0)}
\]
\[
\text{PSW}[N] \leftarrow \neg \text{cond\_satisfied};
\]
\[
\text{else}
\]
\[
\text{PSW}[N] \leftarrow \text{cond\_satisfied};
\]

Exceptions: Taken branch trap
**Compare Immediate and Clear**

**CMPICLR**

<table>
<thead>
<tr>
<th>Format:</th>
<th>CMPICLR,cond i,r,t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(9)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>im11</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

**Purpose:** To compare an immediate value with the contents of a register, set a register to 0, and conditionally nullify the following instruction.

**Description:** The sign-extended immediate and GR \( r \) are compared and GR \( t \) is set to zero. The immediate value is encoded into the \( \text{im11} \) field. The following instruction is nullified if the values compared satisfy the specified condition, \( \text{cond} \). The condition is encoded in the \( c \), \( d \), and \( f \) fields of the instruction.

**Conditions:** The condition is any of the 32-bit compare or subtract conditions shown in Table D-3 on page D-4 or any of the 64-bit compare or subtract conditions shown in Table D-4 on page D-4. When a condition completer is not specified, the "never" condition is used. The boolean variable "\( \text{cond\_satisfied} \)" in the operation section is set when the values compared satisfy the specified condition.

**Operation:**

\[
\text{low\_sign\_ext(} \text{im11,11}) + \neg \text{GR}[r] + 1; \\
\text{GR}[t] \leftarrow 0; \\
\text{if (cond\_satisfied) PSW}[N] \leftarrow 1;
\]

**Exceptions:** None
Coprocessor Operation

Format: COPR,uid,sop,n

<table>
<thead>
<tr>
<th></th>
<th>COPR</th>
<th>uid</th>
<th>sop</th>
<th>n</th>
<th>sop2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0C</td>
<td>17</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

(38)

Purpose: To invoke a coprocessor unit operation.

Description: The coprocessor operation code sop (assembled from the sop1 and sop2 fields) is sent to the coprocessor identified by uid and the indicated operation is performed. If nullification is specified and the coprocessor condition is satisfied, the following instruction is nullified.

Operation: sop ← cat(sop1,sop2);
coprocessor_op(uid,sop,n,IAOQ_Front{30..31});
if (n && coprocessor_condition(uid,sop,n))
    PSW[N] ← 1;

Exceptions: Assist emulation trap Assist exception trap

Notes: The COPROCESSOR OPERATION instruction is used to implement the IDENTIFY COPROCESSOR pseudo-operation. This operation places an identification number from the coprocessor uid into coprocessor register 0. This value is implementation dependent and is useful for configuration, diagnostic, and error recovery.

Each implementation must choose an identification number that identifies the version of the coprocessor. The values all zeros and all ones are reserved. An assist exception trap is not allowed and this instruction must be implemented by all coprocessors with unit identifiers 4 through 7. Unit identifiers 0 and 2 have a uid-specific sequence to obtain the identification number.

The format of the identification number for the floating-point coprocessor is described in “Floating-Point Status Register” on page 8-8.

The IDENTIFY COPROCESSOR pseudo-operation is coded as follows: COPR,uid,0
### Coprocessor Store Doubleword

**Format:**

\[
\text{CSTD}, \text{uid}, \text{cmplt}, \text{cc} \quad r \times d(s, b)
\]

<table>
<thead>
<tr>
<th>0B</th>
<th>b</th>
<th>im5</th>
<th>s</th>
<th>a</th>
<th>cc</th>
<th>1</th>
<th>uid</th>
<th>m</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0B</th>
<th>b</th>
<th>x</th>
<th>s</th>
<th>u</th>
<th>cc</th>
<th>1</th>
<th>uid</th>
<th>m</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:**

To store a doubleword from a coprocessor register.

**Description:**

Register \( r \) of the coprocessor identified by \( uid \) 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 \( d \) (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. The completer also specifies base register modification, optional index prescaling, and ordering constraints (see Table H-3 on page H-8, and Table H-1 on page H-4 for the assembly language completer mnemonics.) The completer, \( \text{cc} \), specifies the cache control hint (see Table 6-8 on page 6-10.)

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.
Operation: if (indexed_store) /* indexed (Format 40)*/
   switch (cmplt) {
     case S:
     case SM:
       dx ← lshift(GRx, 3);
       break;
     case M:
     default:
       dx ← GRx;
       break;
   }
} else /* short displacement */ /* (Format 42) */
   dx ← low_sign_ext(im5, 5);
   space ← space_select(s, GRb, format);
   if (cmplt == O)
     enforce_ordered_store;
   switch (cmplt) {
     case MB:
       offset ← GRb + dx;
       GRb ← GRb + dx;
       break;
     case MA:
     case M:
     case SM:
       offset ← GRb;
       GRb ← GRb + dx;
       break;
     default:
       offset ← GRb + dx;
       break;
   }
   send_to_copr(uid, r);
   mem_store(space, offset, 0, 63, cc, CPR[uid][r]);

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

Restrictions: If the completer O is specified, the displacement must be 0.
**Coprocessor Store Word**

**Format:** \[\text{CSTW}, \text{uid}, \text{cmplt}, \text{cc}, \text{r}, \text{x}|\text{d}(s, b)\]

<table>
<thead>
<tr>
<th>Format</th>
<th>09</th>
<th>b</th>
<th>im5</th>
<th>s</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>1</th>
<th>uid</th>
<th>m</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>(42)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>(40)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To store a word from a coprocessor register.

**Description:** Register \(r\) of the coprocessor identified by \(uid\) 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 \(d\) (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. The completer also specifies base register modification, optional index prescaling, and ordering constraints (see Table H-3 on page H-8, and Table H-1 on page H-4 for the assembly language completer mnemonics.) The completer, \(\text{cc}\), specifies the cache control hint (see Table 6-8 on page 6-10.)

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.
Operation:  
if (indexed_store)  
switch (cmplt) {  
case S:  
case SM:  
dx ← lshift(GR[x],2);  
break;  
case M:  
default:  
dx ← GR[x];  
break;  
}  
} 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:  
offset ← GR[b];  
GR[b] ← GR[b] + dx;  
break;  
default:  
offset ← GR[b] + dx;  
break;  
}  
send_to_copr(uid,r);  
mem_store(space,offset,0,31,cc,CPR[uid][r]);  
/* indexed (Format 40)*/  
/* short displacement */  
/* (Format 42) */  

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

Restrictions:  
If the completer O is specified, the displacement must be 0.
**Decimal Correct (DCOR)**

**Format:**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DCOR</td>
<td>cmplt</td>
<td>cond</td>
<td>r</td>
<td>t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(8)

<table>
<thead>
<tr>
<th></th>
<th>02</th>
<th></th>
<th></th>
<th>0</th>
<th></th>
<th>1</th>
<th>1</th>
<th></th>
<th>1</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To separately correct the 16 BCD digits of the result of an addition or subtraction.

**Description:** A decimal correction value, computed from the 4-bit carries in the PSW C/B bits, is combined with GR\_r, and the result is placed in GR\_t. The correction can be either an intermediate correction (cmplt == 1), which leaves the result pre-biased, or a final correction (no cmplt), which removes the pre-bias. This is encoded in the e1 field (e1=3 for intermediate correction, e1=2 for final correction.)

For intermediate correction, every digit of GR\_r corresponding to a bit which is 1 in the PSW C/B-bits has 6 added to it. For final correction, every digit of GR\_r corresponding to a bit which is 0 in the PSW C/B-bits has 6 subtracted from it.

The following instruction is nullified if the result of the operation satisfies the specified condition cond. The condition is encoded in the c, d, and f fields of the instruction.

**Conditions:** The condition cond is any of the 32-bit unit conditions shown in Table D-11 on page D-8 or any of the 64-bit unit conditions shown in Table D-12 on page D-8. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond\_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.
Operation: if (cmplt == I)

GR[t] ← GR[r] + cat(
  0x6*PSW[C/B][0], 0x6*PSW[C/B][1],
  0x6*PSW[C/B][2], 0x6*PSW[C/B][3],
  0x6*PSW[C/B][4], 0x6*PSW[C/B][5],
  0x6*PSW[C/B][6], 0x6*PSW[C/B][7],
  0x6*PSW[C/B][8], 0x6*PSW[C/B][9],
  0x6*PSW[C/B][10], 0x6*PSW[C/B][11],
  0x6*PSW[C/B][12], 0x6*PSW[C/B][13],
  0x6*PSW[C/B][14], 0x6*PSW[C/B][15]);

else

GR[t] ← GR[r] - cat(
  0x6*(1 - PSW[C/B][0]), 0x6*(1 - PSW[C/B][1]),
  0x6*(1 - PSW[C/B][2]), 0x6*(1 - PSW[C/B][3]),
  0x6*(1 - PSW[C/B][4]), 0x6*(1 - PSW[C/B][5]),
  0x6*(1 - PSW[C/B][6]), 0x6*(1 - PSW[C/B][7]),
  0x6*(1 - PSW[C/B][8]), 0x6*(1 - PSW[C/B][9]),
  0x6*(1 - PSW[C/B][10]), 0x6*(1 - PSW[C/B][11]),
  0x6*(1 - PSW[C/B][12]), 0x6*(1 - PSW[C/B][13]),
  0x6*(1 - PSW[C/B][14]), 0x6*(1 - PSW[C/B][15]));

if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

---

**Programming Note**

DECIMAL CORRECT can be used to take the sum of 64-bit BCD values. *ra*, *rb*, *rc*, and *rd* each contain a 64-bit BCD value and *rt* will hold the result at the end of the sequence. The UADDCM operation is used to pre-bias the value in *ra* in order to perform BCD arithmetic. The DCOR,I operations between the ADD operations are used to re-adjust the BCD bias of the result. The final DCOR operation is used to remove the bias and leave the value in *rt* in BCD format. For the following example, the register *nines* contains the value 0x99999999 99999999.

```
UADDCM ra,nines,rt ; pre-bias first operand
ADD rt,rb,rt ; add in the next value
DCOR,I rt,rt ; correct result, retaining bias
ADD rt,rc,rt ; add in the next value
DCOR,I rt,rt ; correct result, retaining bias
ADD rt,rd,rt ; add in the next value
DCOR rt,rt ; final correction
```
Deposit Doubleword

Format: DEPD, cmpl, cond → r, pos, len, t

<table>
<thead>
<tr>
<th>Format</th>
<th>DEPD, cmpl, cond</th>
<th>r, pos, len, t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13)</td>
<td>35</td>
<td>6</td>
</tr>
<tr>
<td>(16)</td>
<td>3C</td>
<td>6</td>
</tr>
</tbody>
</table>

Purpose: To deposit a value into a register at a fixed or variable position, and conditionally nullify the following instruction.

Description: A right-justified field from GR r is deposited (merged) into GR t. The field begins at the bit position given by pos and extends len bits to the left. The remainder of GR t is optionally zeroed or left unchanged.

The bit position, pos, can either be a constant (specifying a fixed deposit), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable deposit.) Format 13 is used for variable deposits; Format 16 is used for fixed deposits. For variable deposits, if the deposited field extends beyond the leftmost bit, it is truncated and the higher bits are ignored. For fixed deposits, it is an undefined operation for the field to extend beyond the leftmost bit.

The completer, cmplt, determines whether or not the target register is zeroed before the field is deposited into it. (Table 7-1 defines the assembly language completer mnemonics.) This is encoded in the nz field of the instruction, with 0 indicating that the register is zeroed and 1 indicating that it is not.

Table 7-1. Deposit Instruction Completers

<table>
<thead>
<tr>
<th>cmplt</th>
<th>Description</th>
<th>nz</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>value is deposited into the old value of the target register</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>value is deposited into a field of zeros</td>
<td>0</td>
</tr>
</tbody>
</table>

The following diagram illustrates a fixed deposit of a 50-bit field at bit position 56. The instruction is: DEPD r, 56, 50, t.

The length len in the assembly language format is encoded into the cl and clen fields. For fixed deposits, the bit position pos in the assembly language format is represented by
cat(cp,cpos) in the machine instruction, whose value is 63–pos.

The following instruction is nullified if the result of the operation satisfies the specified condition, cond. The condition is encoded in the c field of the instruction.

Conditions: The condition is any of the 64-bit extract/deposit conditions shown in Table D-14 on page D-9. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

len ← assemble_6(cl,clen)
if (fixed_deposit) {
    if (pos >= len–1)
        tpos ← pos;
    else
        undefined;
} else /* (Format 13) */
    tpos ← CR[11];
if (cmplt == Z) /* nz=0 */
    GR[t] ← 0;
if (tpos–len+1< 0) /* field extends beyond leftmost bit */
    GR[t]{0..tpos} ← GR[r]{63–tpos..63};
else
    GR[t]{tpos–len+1..tpos} ← GR[r]{64–len..63};
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

Restrictions: Since for fixed deposits, the deposited field is fully specified by len and pos, it is an undefined operation if the field extends beyond the leftmost bit.

Notes: The SHLD,cond r,sa,t pseudo-operation generates a DEPD,Z,cond r,63-sa,64-sa,t instruction to perform a shift left by sa bits on the doubleword in general register r.
Deposit Doubleword Immediate

Format: DEPDI,cmplt,cond i,pos,len,t

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>im5</th>
<th>c</th>
<th>nz</th>
<th>cl</th>
<th>0</th>
<th>clen</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3D</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To deposit an immediate value into a register at a fixed or variable position, and conditionally nullify the following instruction.

Description: A right-justified field from the sign-extended immediate $i$ is deposited (merged) into GR $t$. The field begins at the bit position given by $pos$ and extends $len$ bits to the left. The remainder of GR $t$ is optionally zeroed or left unchanged.

The bit position, $pos$, can either be a constant (specifying a fixed deposit), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable deposit.) Format 13 is used for variable deposits; Format 16 is used for fixed deposits. For variable deposits, if the deposited field extends beyond the leftmost bit, it is truncated and the higher bits are ignored. For fixed deposits, it is an undefined operation for the field to extend beyond the leftmost bit.

The completer, $cmplt$, determines whether or not the target register is zeroed before the field is deposited into it (see Table 7-1 on page 7-38 for the assembly language completer mnemonics.) This is encoded in the $nz$ field of the instruction, with 0 indicating that the register is zeroed and 1 indicating that it is not.

The following diagram illustrates a fixed deposit of the value 0x9 into a 50-bit field at bit position 56. The instruction is: DEPDI 0x9,56,50,t.

```
0: 14 i: 0x9
   |                |               |
   |                | 0x9           |
   |                | 63            |
   |                |                |
GR t: 0: 56       |
       | 0x9           |
       | 63            |
       |                |
```

The length $len$ in the assembly language format is encoded into the $cl$ and $clen$ fields. For fixed deposits, the bit position $pos$ in the assembly language format is represented by $\text{cat} (cp, cp\text{pos})$ in the machine instruction, whose value is 63–$pos$. The immediate is encoded in the $im5$ field of the instruction.

The following instruction is nullified if the result of the operation satisfies the specified condition, $cond$. The condition is encoded in the $c$ field of the instruction.

Conditions: The condition is any of the 64-bit extract/deposit conditions shown in Table D-14 on
When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

**Operation:**

\[
\begin{align*}
\text{len} & \leftarrow \text{assemble}_6(cl,clen) \\
\text{ival} & \leftarrow \text{low_sign_ext(im5,5)}; \\
\text{if} \ (\text{fixed_deposit}) \{ & \quad /* \text{Format 16} */ \\
& \quad \text{if} \ (\text{pos} \geq \text{len}-1) \\
& \quad \quad tpos \leftarrow \text{pos}; \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{undefined;} \\
\} \ \text{else} & \quad /* \text{Format 13} */ \\
& \quad \quad tpos \leftarrow \text{CR}[11]; \\
& \quad \quad \text{if} \ (\text{cmplt} \equiv Z) & \quad /* \text{nz}=0 */ \\
& \quad \quad \quad \text{GR}[t] \leftarrow 0; \\
& \quad \quad \text{if} \ (\text{tpos}-\text{len}+1<0) & \quad /* \text{field extends beyond leftmost bit} */ \\
& \quad \quad \quad \text{GR}[t][0..\text{tpos}] \leftarrow \text{ival}[63-\text{tpos}..63]; \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{GR}[t][\text{tpos}-\text{len}+1..\text{tpos}] \leftarrow \text{ival}[64-\text{len}..63]; \\
& \quad \quad \text{if} \ (\text{cond_satisfied}) \ \text{PSW}[N] \leftarrow 1; \\
\end{align*}
\]

**Exceptions:** None

**Restrictions:** Since for fixed deposits, the deposited field is fully specified by \(\text{len}\) and \(\text{pos}\), it is an undefined operation if the field extends beyond the leftmost bit.
**Deposit Word**

**DEPW, cmplt, cond r, pos, len, t**

<table>
<thead>
<tr>
<th>Format</th>
<th>DEPW, cmplt, cond</th>
<th>r, pos, len, t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13)</td>
<td>35</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>(16)</td>
<td>35</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To deposit a value into the rightmost 32 bits of a register at a fixed or variable position, and conditionally nullify the following instruction.

**Description:** A right-justified field from GR \( r \) is deposited (merged) into the rightmost 32 bits of GR \( t \). The field begins at the bit position given by \( pos + 32 \) and extends \( len \) bits to the left. The remainder of GR \( t \) is optionally zeroed or left unchanged. The leftmost 32 bits of GR \( t \) are undefined.

The bit position, \( pos \), can either be a constant (specifying a fixed deposit), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable deposit.) Format 13 is used for variable deposits; Format 16 is used for fixed deposits. For variable deposits, the leftmost bit of the SAR is ignored, and 32 is added to the value in the lower 5 bits. For fixed deposits, 32 is added to the \( pos \) value in the instruction. For variable deposits, if the deposited field extends beyond the leftmost bit (of the rightmost 32), it is truncated and the higher bits are ignored. For fixed deposits, it is an undefined operation for the field to extend beyond the leftmost bit (of the rightmost 32.)

The completer, \( \text{cmplt} \), determines whether or not the target register is zeroed before the field is deposited into it (see Table 7-1 on page 7-38 for the assembly language completer mnemonics.) This is encoded in the \( nz \) field of the instruction, with 0 indicating that the register is zeroed and 1 indicating that it is not.

The following diagram illustrates a deposit of a 10-bit field when the Shift Amount Register contains the value 24. The instruction is: DEPW r, sar, 10, t.

![Deposit Diagram](image)

The length \( len \) in the assembly language format is encoded into the \( clen \) field. For fixed deposits, the bit position \( pos \) in the assembly language format is represented by \( cpos \) in the machine instruction, whose value is \( 31 - pos \).

The following instruction is nullified if the result of the operation satisfies the specified
condition, \textit{cond}. The condition is encoded in the \textit{c} field of the instruction.

Conditions: The condition is any of the 32-bit extract/deposit conditions shown in Table D-13 on page D-9. When a condition completer is not specified, the "never" condition is used. The boolean variable "\textit{cond\_satisfied}" in the operation section is set when the result of the operation satisfies the specified condition.

Operation:

\begin{verbatim}
len ← assemble_6(0, clen);
if (fixed_deposit) {
    if (pos >= len–1)
        tpos ← pos + 32;
    else
        undefined;
} else /* (Format 13) */
    tpos ← CR[11]{1..5} + 32;
if (cmplt == Z) /* nz=0 */
    GR[t]{32..63} ← 0;
if (tpos–len+1< 32) /* field extends beyond leftmost bit */
    GR[t]{32..tpos} ← GR[r]{95–tpos..63};
else
    GR[t]{tpos–len+1..tpos} ← GR[r]{64–len..63};
    GR[t]{0..31} ← undefined;
if (cond\_satisfied) PSW[N] ← 1;
\end{verbatim}

Exceptions: None

Restrictions: Since for fixed deposits, the deposited field is fully specified by \textit{len} and \textit{pos}, it is an undefined operation if the field extends beyond the leftmost bit (of the rightmost 32.)

Notes: The \texttt{SHLW,cond r,sa,t} pseudo-operation generates a \texttt{DEPW,cond r,31-sa,32-sa,t} instruction to perform a shift left by \textit{sa} bits on the word in general register \textit{r}. 


Deposit Word Immediate

**Format:**

```
DEPWI,cmplt,cond  i,pos,len,t
```

<table>
<thead>
<tr>
<th></th>
<th>35</th>
<th>t</th>
<th>im5</th>
<th>c</th>
<th>2</th>
<th>nz</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>clen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>35</th>
<th>t</th>
<th>im5</th>
<th>c</th>
<th>1</th>
<th>nz</th>
<th>cpos</th>
<th>clen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:**
To deposit an immediate value into the rightmost 32 bits of a register at a fixed or variable position, and conditionally nullify the following instruction.

**Description:**
A right-justified field from the sign-extended immediate $i$ is deposited (merged) into the rightmost 32 bits of GR $t$. The field begins at the bit position given by $pos+32$ and extends $len$ bits to the left. The remainder of GR $t$ is optionally zeroed or left unchanged. The leftmost 32 bits of GR $t$ are undefined.

The bit position, $pos$, can either be a constant (specifying a fixed deposit), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable deposit.) Format 13 is used for variable deposits; Format 16 is used for fixed deposits. For variable deposits, the leftmost bit of the SAR is ignored, and 32 is added to the value in the lower 5 bits. For fixed deposits, 32 is added to the $pos$ value in the instruction. For variable deposits, if the deposited field extends beyond the leftmost bit (of the rightmost 32), it is truncated and the higher bits are ignored. For fixed deposits, it is an undefined operation for the field to extend beyond the leftmost bit (of the rightmost 32.)

The completer, $cmplt$, determines whether or not the target register is zeroed before the field is deposited into it (Table 7-1 on page 7-38 for the assembly language completer mnemonics.) This is encoded in the $nz$ field of the instruction, with 0 indicating that the register is zeroed and 1 indicating that it is not.

The following diagram illustrates a deposit of the value $0x9$ into a 10-bit field when the Shift Amount Register contains the value 24. The instruction is: DEPWI  $0x9,sar,10,t$.

```
0  32  54  63
GR r: ____________________________
0  32  47  56  63
GR t: ____________________________
  undefined
```

The length $len$ in the assembly language format is encoded into the $clen$ field. For fixed deposits, the bit position $pos$ in the assembly language format is represented by $cpos$ in the machine instruction, whose value is $31-pos$. The immediate is encoded in the $im5$ field of the instruction.
The following instruction is nullified if the result of the operation satisfies the specified condition, \( \text{cond} \). The condition is encoded in the \( c \) field of the instruction.

**Conditions:** The condition is any of the 32-bit extract/deposit conditions shown in Table D-13 on page D-9. When a condition completer is not specified, the "never" condition is used. The boolean variable "\( \text{cond\_satisfied} \)" in the operation section is set when the result of the operation satisfies the specified condition.

**Operation:**

\[
\begin{align*}
\text{len} & \leftarrow \text{assemble\_6}(0, \text{clen}); \\
\text{ival} & \leftarrow \text{low\_sign\_ext}(\text{im5}, 5); \\
\text{if} \ (\text{fixed\_deposit}) \ {\text{[} \} & \{ /\ast \text{ (Format 16) } /\ast \\
\text{if} \ (\text{pos} \geq \text{len} – 1) \\
& \quad \text{tpos} \leftarrow \text{pos} + 32; \\
\text{else} & \quad \text{undefined}; \\
\} \text{ else} & \quad /\ast \text{ (Format 13) } /\ast \\
& \quad \text{tpos} \leftarrow \text{CR}[11\ldots5] + 32; \\
& \quad \text{if} \ (\text{compl} = Z) \ {/\ast \text{ nz=0 } /\ast \\
& \quad \text{GR}[t][32..63] \leftarrow 0; \\
& \quad \text{if} \ (\text{tpos} – \text{len} + 1 < 32) \ {/\ast \text{ field extends beyond leftmost bit } /\ast \\
& \quad \text{GR}[t][32..\text{tpos}] \leftarrow \text{ival}[95–\text{tpos}..63]; \\
\text{else} & \quad \text{GR}[t][\text{tpos}–\text{len}+1..\text{tpos}] \leftarrow \text{ival}[64–\text{len}..63]; \\
& \quad \text{GR}[t][0..31] \leftarrow \text{undefined}; \\
& \quad \text{if} \ (\text{cond\_satisfied}) \ \text{PSW}[N] \leftarrow 1;
\end{align*}
\]

**Exceptions:** None

**Restrictions:** Since for fixed deposits, the deposited field is fully specified by \( \text{len} \) and \( \text{pos} \), it is an undefined operation if the field extends beyond the leftmost bit (of the rightmost 32.)
### Diagnose (DIAG)

**Format:**

```
DIAG i
```

<table>
<thead>
<tr>
<th>(28)</th>
<th>05</th>
<th>im26</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:**
To provide implementation-dependent operations for system initialization, reconfiguration, and diagnostic purposes.

**Description:**
The immediate value in the assembly language is encoded in the `im26` field of the instruction. Refer to the hardware reference manual for the definition on a particular machine implementation.

**Operation:**
```
if (priv != 0)
    privileged_operation_trap;
else
    implementation_dependent;
```

**Exceptions:**
- Privileged operation trap
- Implementation-dependent.

**Restrictions:**
This instruction may be executed only at the most privileged level.

**Notes:**
Since the DIAG instruction is privileged, a privileged operation trap will result from unprivileged diagnostic software executing DIAG. The trap could invoke an emulator which would allow the unprivileged software access to the required unprivileged implementation-dependent resources.
### Divide Step

**Format:**

\[
\text{DS,cond } r1, r2, t
\]

(8)

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

**Purpose:**

To provide the primitive operation for integer division.

**Description:**

This instruction performs a single-bit non-restoring divide step and produces a set of result conditions. It calculates one bit of the quotient when a 32-bit value in GR \( r1 \) is divided by a 32-bit value in GR \( r2 \) and leaves the partial remainder in GR \( t \). The quotient bit is PSW C/B{8}. The carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the single-bit divide operation.

The following instruction is nullified if the result of the operation satisfies the specified condition, \( \text{cond} \). The condition is encoded in the \( c \) and \( f \) fields of the instruction.

For this instruction, signed overflow condition means that the bit shifted out of the lower 32 bits differs from the leftmost bit of the lower 32 bits following the shift or an ordinary 32-bit signed overflow occurred during the addition or subtraction. Unsigned overflow means that the bit shifted out of the lower 32 bits is 1 or that an ordinary 32-bit unsigned overflow occurred during the addition or subtraction. The conditions take on special interpretations since the shift operation participates in overflow determination.

**Conditions:**

The condition is any of the 32-bit compare or subtract conditions shown in Table D-3 on page D-4. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

**Operation:**

\[
\text{if (PSW[V])}
\]

\[
\begin{align*}
\text{GR}[t] & \leftarrow \text{cat}(\text{lshift}([r1],1), \text{PSW}[C/B][8]) + \sim\text{GR}[r2] + 1; \\
\text{else} & \quad \text{GR}[t] \leftarrow \text{cat}(\text{lshift}([r1],1), \text{PSW}[C/B][8]) + \text{GR}[r2]; \\
\text{PSW}[C/B] & \leftarrow \text{carry_borrows}; \\
\text{PSW}[V] & \leftarrow \text{xor}([\text{carry_borrows}[8], \text{GR}[r2][32])); \\
\text{if (cond_satisfied) PSW}[N] & \leftarrow 1;
\end{align*}
\]

**Exceptions:**

None
Extract Doubleword EXTRD

Format: \texttt{EXTRD},\texttt{cmplt},\texttt{cond} \quad \texttt{r,}\texttt{pos,}\texttt{len,}\texttt{t}

\begin{center}
\begin{tabular}{cccccccc}
\hline
34 &   &   & 3 & 2 & 2 & 1 & 1 & 1 & 3 & 5 \\
6 &   &   & 5 & 5 & 3 & 1 & 1 & 1 & 5 & 5 \\
\hline
\end{tabular}
\end{center}

Purpose: To extract any 64-bit or shorter field from a fixed or variable position, and conditionally nullify the following instruction.

Description: A field is extracted from GR \texttt{r}, zero or sign extended and placed right-justified in GR \texttt{t}. The field begins at the bit position given by \texttt{pos} and extends \texttt{len} bits to the left.

The bit position, \texttt{pos}, can either be a constant (specifying a fixed extract), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable extract.) Format 12 is used for variable extracts; Format 15 is used for fixed extracts. For variable extracts, if the extracted field extends beyond the leftmost bit, it is zero or sign extended. For fixed extracts, it is an undefined operation for the field to extend beyond the leftmost bit.

The completer, \texttt{cmplt}, determines whether the extracted field is zero extended or sign extended. (Table 7-2 defines the assembly language completer mnemonics.) This is encoded in the \texttt{se} field of the instruction, with 1 indicating sign extension and 0 indicating zero extension.

The following diagram illustrates a fixed extract of a 50-bit field at bit position 56. The instruction is: EXTRD,\texttt{U} \quad \texttt{r,56,50,}\texttt{t}.

The length \texttt{len} in the assembly language format is encoded into the \texttt{cl} and \texttt{clen} fields. For fixed extracts, the bit position \texttt{pos} in the assembly language format is represented in the machine instruction by cat(\texttt{p,}\texttt{pos}).
The following instruction is nullified if the result of the operation satisfies the specified condition, \( \text{cond} \). The condition is encoded in the \( c \) field of the instruction.

**Conditions:**
The condition is any of the 64-bit extract/deposit conditions shown in Table D-14 on page D-9. When a condition completer is not specified, the "never" condition is used. The boolean variable "\( \text{cond} \text{satisfied} \)" in the operation section is set when the result of the operation satisfies the specified condition.

**Operation:**

\[
\text{len} \leftarrow \text{assemble}_6(\text{cl},\text{clen})
\]

if (\( \text{variable\_extract} \)) { /* (Format 12) */
    \text{pos} \leftarrow \text{CR}[11];
    \text{shamt} \leftarrow 63 - \text{pos};
    \text{if} (\text{pos} \geq \text{len}-1)
        \text{tlen} \leftarrow \text{len};
    \text{else}
        \text{if} (\text{variable\_extract}) /* (Format 12) */
            \text{tlen} \leftarrow \text{pos} + 1;
        \text{else} /* (Format 15) */
            \text{undefined};
    \text{if} (\text{cmplt} == \text{U}) /* \text{se}=0 */
        \text{GR}[\text{t}] \leftarrow \text{zero\_ext}(\text{rshift}(\text{GR}[\text{r}],\text{shamt}),\text{tlen});
    \text{else} /* \text{se}=1 */
        \text{GR}[\text{t}] \leftarrow \text{sign\_ext}(\text{rshift}(\text{GR}[\text{r}],\text{shamt}),\text{tlen});
\]

\text{if} (\text{cond} \text{satisfied}) \text{PSW}[\text{N}] \leftarrow 1;

**Exceptions:** None

**Restrictions:** Since for fixed extracts, the extracted field is fully specified by \( \text{len} \) and \( \text{pos} \), it is an undefined operation if the field extends beyond the leftmost bit.

---

**Programming Note**

An arithmetic right shift of a 64-bit value in \( \text{GR} \text{ r} \) by a variable amount contained in \( \text{GR} \text{ p} \) leaving the result in \( \text{GR} \text{ t} \) may be done by the following sequence:

\[
\text{MTSARCM} \ p
\text{EXTRD.S} \ r,\text{sar},64,t
\]

**Notes:**
The \text{SHRD.S},\text{cond} \text{ r,sa,t} pseudo-operation generates a \text{EXTRD.S},\text{cond} \text{ r,63-sa,64-sa,t} instruction to perform a signed shift right by \( \text{sa} \) bits on the doubleword in general register \text{r}.

The \text{SHRD.U},\text{cond} \text{ r,sa,t} pseudo-operation generates a \text{EXTRD.U},\text{cond} \text{ r,63-sa,64-sa,t} instruction to perform an unsigned shift right by \( \text{sa} \) bits on the doubleword in general register \text{r}.
Extract Word

**EXTRW**

**Format:** EXTRW,cmplt,cond r,pos,len,t

<table>
<thead>
<tr>
<th>(12)</th>
<th>34</th>
<th>r</th>
<th>t</th>
<th>c</th>
<th>2</th>
<th>se</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>clen</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(15)</th>
<th>34</th>
<th>r</th>
<th>t</th>
<th>c</th>
<th>1</th>
<th>1</th>
<th>se</th>
<th>pos</th>
<th>clen</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>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To extract any 32-bit or shorter field from a fixed or variable position, and conditionally nullify the following instruction.

**Description:** A field is extracted from the rightmost 32 bits of GR r, zero or sign extended and placed right-justified in GR t. The field begins at the bit position given by pos+32 and extends len bits to the left. The leftmost 32 bits of GR t are undefined.

The bit position, pos, can either be a constant (specifying a fixed extract), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable extract.) Format 12 is used for variable extracts; Format 15 is used for fixed extracts. For variable extracts, the leftmost bit of the SAR is ignored, and 32 is added to the value in the lower 5 bits. For fixed extracts, 32 is added to the pos value in the instruction. For variable extracts, if the extracted field extends beyond the leftmost bit (of the rightmost 32), it is zero or sign extended. For fixed extracts, it is an undefined operation for the field to extend beyond the leftmost bit (of the rightmost 32.)

The completer, cmplt, determines whether the extracted field is zero extended or sign extended (see Table 7-2 on page 7-48 for the assembly language completer mnemonics.) This is encoded in the se field of the instruction, with 1 indicating sign extension and 0 indicating zero extension.

The following diagram illustrates a variable extract of a 10-bit field when the Shift Amount Register contains the value 24. The instruction is: EXTRW,U r,sar,10,t.

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>32</th>
<th>47</th>
<th>56</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR r:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GR t:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>undefined</td>
</tr>
</tbody>
</table>
```

The length len in the assembly language format is encoded into the clen field.

The following instruction is nullified if the result of the operation satisfies the specified condition, cond. The condition is encoded in the c field of the instruction.
Conditions: The condition is any of the 32-bit extract/deposit conditions shown in Table D-13 on page D-9. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

Operation: len ← assemble_6(0, clen);
if (variable_extract) { /* (Format 12) */
  pos ← CR[11][1..5];
  shamt ← 31 – pos;
  if (pos >= len–1)
    tlen ← len;
  else
    if (variable_extract) /* (Format 12) */
      tlen ← pos + 1;
    else /* (Format 15) */
      undefined;
  if (cmplt == U) /* se=0 */
    GR[t]{32..63} ← zero_ext(rshift(GR[r], shamt), tlen){32..63};
  else /* se=1 */
    GR[t]{32..63} ← sign_ext(rshift(GR[r], shamt), tlen){32..63};
  GR[t]{0..31} ← undefined;
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

Restrictions: Since for fixed extracts, the extracted field is fully specified by len and pos, it is an undefined operation if the field extends beyond the leftmost bit (of the rightmost 32.)

Programming Note
An arithmetic right shift of a 32-bit value in GR r by a variable amount contained in GR p leaving the result in GR t may be done by the following sequence:

MTSARCM p
EXTRW,S r,sar,32,t

Notes: The SHRWS,cond r,sa,t pseudo-operation generates a EXTRW,S,cond r,31-sa,32-sa,t instruction to perform a signed shift right by sa bits on the word in general register r.

The SHRW,U,cond r,sa,t pseudo-operation generates a EXTRW,U,cond r,31-sa,32-sa,t instruction to perform an unsigned shift right by sa bits on the word in general register r.
Flush Data Cache  

**Format:**  

FDC,cmplt x(s,b)  
FDC d(s,b)  

(24)  

<table>
<thead>
<tr>
<th>01</th>
<th>b</th>
<th>x</th>
<th>s</th>
<th>4A</th>
<th>m</th>
<th>rv</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

(25)  

<table>
<thead>
<tr>
<th>01</th>
<th>b</th>
<th>im5</th>
<th>s</th>
<th>CA</th>
<th>0</th>
<th>rv</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:**  
To invalidate a data cache line and write it back to memory if it is dirty.

**Description:**  
The data cache line (if present) specified by the effective address generated by the instruction is written back to memory, if and only if it is dirty, and then invalidated from the data cache. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 24), or a displacement \( d \) (Format 25). The displacement is encoded into the \( \text{im5} \) field. Optional base modification can also be performed with the indexed form.

The completer, \( \text{cmplt} \), determines whether the offset is the base register, or the base register plus the index register or displacement. The completer, encoded in the \( m \)-field of the instruction, also specifies base register modification. (Table 7-3 defines the assembly language completer mnemonics.)

<table>
<thead>
<tr>
<th>cmplt</th>
<th>Description</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>don’t modify base register</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>Modify base register</td>
<td>1</td>
</tr>
</tbody>
</table>

The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

A cache line is called *dirty* if any byte has been written to since it was read from memory or if a STBY,E to the leftmost byte of a word has been performed.

In a multiprocessor system, a flush request is broadcast to all data and combined caches.

**Operation:**  

```c
space ← space_select(s,GR[b],format);  
if (indexed_load) {  
    switch (cmplt) {  
        case M:     offset ← GR[b];  
                    GR[b] ← GR[b] + GR[x];  
                    break;  
        default:    offset ← GR[b] + GR[x];  
                    break;  
    }  
}```

/* indexed (Format 24)*/
else /* short displacement (Format 25) */
    offset ← GR[b] + low_sign_ext(im5,5); /* (new) */
Dcache_flush(space,offset);

Exceptions: Non-access data TLB miss fault

Notes: For systems that do not have a cache, this instruction executes as a null instruction.

In systems with a combined cache, this instruction may be used to flush both data and
instruction lines from the cache.

This instruction may be executed out of sequence but must satisfy the instruction ordering
constraints. The SYNC instruction enforces program order with respect to the instructions
following the SYNC.

It is an undefined operation to execute an FDC with a nonzero s-field at a nonzero privilege
level when the PSW W-bit is 1.
Flush Data Cache Entry

Format: 

<table>
<thead>
<tr>
<th></th>
<th>FDCE.cmp</th>
<th>x(s,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>b</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

(24)

Purpose: To provide for flushing the entire data or combined cache by causing zero or more cache lines to be invalidated.

Description: Zero or more cache lines specified by an implementation-dependent function of the effective address are written back to main memory, if and only if they are dirty, and are invalidated in the data or combined cache. The completer, *cmp*, determines if the offset is the base register, *b*, or the base register plus the index register *x*. The completer, encoded in the *m*-field of the instruction, specifies base register modification. No address translation is performed (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.)

When this instruction is used in an architecturally defined cache flush loop, the entire data or combined cache will be flushed upon completion of the loop.

Operation: 

```
space ← space_select(s,GR[b],INDEXED);
switch (cmp) {
    case M:  offset ← GR[b];  /*m=1*/
             GR[b] ← GR[b] + GR[x];
             break;
    default: offset ← GR[b] + GR[x];  /*m=0*/
             break;
}
Dcache_flush_entries(space,offset);
```

Exceptions: None

Notes: In a multiprocessor system, this instruction is not broadcast to other processors. This instruction does not necessarily flush the entry specified by “space” and “offset”.

For systems that do not have a cache, this instruction executes as a null instruction.

It is an undefined operation to execute an FDCE with a nonzero *s*-field at a nonzero privilege level when the PSW W-bit is 1.
Flush Instruction Cache

**Format:**

FIC.cmplt $x(s|r,b)$

<table>
<thead>
<tr>
<th>Format</th>
<th>01</th>
<th>b</th>
<th>x</th>
<th>s</th>
<th>0A</th>
<th>m</th>
<th>rv</th>
</tr>
</thead>
<tbody>
<tr>
<td>(26)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>(24)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To invalidate an instruction cache line.

**Description:**

The instruction cache line (if any) specified by the effective address generated by the instruction is invalidated in the instruction cache. The completer, `cmplt`, determines if the offset is the base register, $b$, or the base register plus the index register $x$. The completer, encoded in the $m$-field of the instruction, also specifies base register modification (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.)

The space register, $sr$, is explicitly encoded in the 3-bit $s$ field of the instruction (Format 26) or is implicitly specified by the 2-bit $s$ field of the instruction (Format 24.)

The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

Either the instruction TLB or the data TLB can be used to perform the address translation for the address to be flushed. If the data TLB is used, a TLB miss fault is reported using a non-access data TLB miss fault.

In a multiprocessor system, a flush request is broadcast to all instruction and combined caches.

**Operation:**

```c
if (explicit_pointer) /*(Format 26)*/
    space ← SR[assemble_3(s)];
else /*(Format 24)*/
    space ← space_select(s,GR[b],INDEXED);
switch (cmplt) {
    case M: offset ← GR[b]; /*m=1*/
            GR[b] ← GR[b] + GR[x];
            break;
    default: offset ← GR[b] + GR[x]; /*m=0*/
            break;
}
Icache_flush(space,offset);
```

**Exceptions:**

- Non-access instruction TLB miss fault
- Non-access data TLB miss fault

**Notes:**

For systems that do not have a cache, this instruction executes as a null instruction.

In systems with a combined cache, this instruction may be used to flush both instruction
and data lines from the cache, including writing them back to main memory, if they are dirty.

This instruction may be executed out of sequence but must satisfy the instruction ordering constraints. The SYNC instruction enforces program order with respect to the instructions following the SYNC.

It is an undefined operation to execute an implicit-pointer FIC with a nonzero s-field at a nonzero privilege level when the PSW W-bit is 1.
Flush Instruction Cache Entry

<table>
<thead>
<tr>
<th>Format: FICE, cmplt x(sr,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(26)</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Purpose: To provide for flushing the entire instruction or combined cache by causing zero or more cache lines to be invalidated.

Description: Zero or more cache lines specified by an implementation-dependent function of the effective address are invalidated in the instruction or combined cache. For implementations with a combined cache, the cache lines are written back to main memory, if and only if they are dirty, and are invalidated. The completer, cmplt, determines if the offset is the base register, b, or the base register plus the index register x. The completer, encoded in the m-field of the instruction, specifies base register modification. No address translation is performed (see Table 7.3 on page 7.52 for the assembly language completer mnemonics.) The space register, sr, is encoded in the s field of the instruction.

When this instruction is used in an architecturally defined cache flush loop, the entire instruction or combined cache will be flushed upon completion of the loop (all the contents of the instruction cache, except the loop itself, prior to the beginning of the flush loop must be flushed.)

```
Operation: switch (cmplt) {
    case M: offset ← GR[b]; /*m=1*/
        GR[b] ← GR[b] + GR[x];
        break;
    default: offset ← GR[b] + GR[x]; /*m=0*/
        break;
}

space ← SR[assemble_3(s)];
Icache_flush_entries(space,offset);
```

Exceptions: None

Notes: In a multiprocessor system, this instruction is not broadcast to other processors. This instruction does not necessarily flush the entry specified by “space” and “offset”.

For systems which do not have a cache, this instruction executes as a null instruction.
Halfword Parallel Add

**Format:**  \( \text{HADD}, \text{cmplt} \quad r1, r2, t \)

### (8)

<table>
<thead>
<tr>
<th></th>
<th>r2</th>
<th>r1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>sat</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>2</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To add multiple halfwords in parallel with optional saturation.

**Description:** The corresponding halfwords of GR \( r1 \) and GR \( r2 \) are added together in parallel. Optional saturation is performed, which forces each halfword result to either the maximum or the minimum value, if the result would have been out of the range of the target format. The halfword results are placed in GR \( t \).

The completer, \( \text{cmplt} \), determines whether modular, signed-saturation, or unsigned-saturation arithmetic is performed. The completer is encoded in the \( \text{sat} \) field of the instruction. (Table 7-4 defines the assembly language completer mnemonics.) For signed saturation, all operands are treated as signed numbers, and the results are signed numbers. For unsigned saturation, the first operands, from GR \( r1 \), are treated as unsigned numbers, the second operands, from GR \( r2 \), are treated as signed numbers, and the results are unsigned numbers.

**Table 7-4. Halfword Arithmetic Completers**

<table>
<thead>
<tr>
<th>cmplt</th>
<th>Description</th>
<th>sat</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>modular arithmetic</td>
<td>3</td>
</tr>
<tr>
<td>SS</td>
<td>Signed Saturation</td>
<td>1</td>
</tr>
<tr>
<td>US</td>
<td>Unsigned Saturation</td>
<td>0</td>
</tr>
</tbody>
</table>

**Operation:**

```plaintext
parallel for (start ← 0; start <= 48; start += 16) {
    end ← start + 15;
    GR[t]{start..end} ← (GR[r1]{start..end} + GR[r2]{start..end});
    switch (cmplt) {
        case SS: if (maximum_signed_saturation) /* sat=1 */
            GR[t]{start..end} ← 0x7FFF;
        else if (minimum_signed_saturation)
            GR[t]{start..end} ← 0x8000;
        break;
        case US: if (maximum_unsigned_saturation) /* sat=0 */
            GR[t]{start..end} ← 0xFFF;
        else if (minimum_unsigned_saturation)
            GR[t]{start..end} ← 0x0000;
        break;
        default: /* sat=3 */
            break;
    }
}
```

**Exceptions:** None.
Halfword Parallel Average

**Format:**

```
HAVG r1,r2,t
```

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

**Purpose:**
To average multiple halfwords in parallel.

**Description:**
The corresponding halfwords of GR $r1$ and GR $r2$ are averaged in parallel. The average is obtained by adding the corresponding halfwords, and shifting the result right by one bit, to perform a divide by 2, with the halfword carry bit from the addition shifted back into the leftmost position of each result. The halfword results are placed in GR $t$.

Unbiased rounding is performed on the results to reduce the accumulation of rounding errors with cascaded operations.

**Operation:**
Parallel for (start ← 0; start <= 48; start += 16){
    end ← start + 15;
    sum ← GR[r1][start..end] + GR[r2][start..end];
    new_lsb ← sum{14} | sum{15}; /*unbiased rounding*/
    GR[t][start..end] ← cat(carry,sum{0..13},new_lsb);
}

**Exceptions:** None
Halfword Parallel Shift Left

<table>
<thead>
<tr>
<th>Format:</th>
<th>HSHL r,sa,t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3E 0 r 1 0 0 2 sa 0 t</td>
</tr>
<tr>
<td></td>
<td>6 5 5 1 2 1 2 4 1 5</td>
</tr>
</tbody>
</table>

Purpose: To perform multiple parallel halfword shift left operations.

Description: Each of the halfwords in GR $r$ is shifted left $sa$ bits. The shift amount is between 0 and 15, and is encoded in the $sa$ field in the instruction. The halfword results are placed in GR $t$.

Operation:

```
parallel for (start ← 0; start <= 48; start += 16) {
    end ← start + 15;
    GR[t]{start..end} ← lshift(GR[r1]{start..end},sa);
}
```

Exceptions: None.
Halfword Parallel Shift Left and Add

**Format:**

```
HSHLADD    r1,sa,r2,t
```

**Purpose:**

To perform multiple halfword shift left and add operations in parallel with saturation.

**Description:**

Each halfword of GR \( r1 \) is shifted left by \( sa \) bits, and then added to the corresponding halfword of GR \( r2 \). Signed saturation is performed, which forces each halfword result to either the maximum or the minimum value, if the result would have been out of range. The halfword results are placed in GR \( t \). The shift amount is either 1, 2, or 3, and is encoded in the \( sa \) field of the instruction.

All operands are treated as signed numbers, and the results are signed numbers. Signed saturation is performed.

For this instruction, signed saturation is based both on the shift operation and the add operation. That is, if the result of the shift operation is not representable in 16 bits, signed saturation occurs. If GR \( r1 \) was positive, maximum saturation occurs. If GR \( r2 \) was negative, minimum saturation occurs. If the result of the shift operation is representable in 16 bits, then saturation is determined by the add operation in the normal fashion.

**Operation:**

Parallel for (start ← 0; start <= 48; start += 16) {
  end ← start + 15;
  GR[t]{start..end} ← lshift(GR[r1]{start..end},sa) + GR[r2]{start..end};
  if (maximum_signed_saturation)
    GR[t]{start..end} ← 0x7FFF;
  else if(minimum_signed_saturation)
    GR[t]{start..end} ← 0x8000;
}

**Exceptions:**

None
Halfword Parallel Shift Right

**Format:**

```
HSHR,cmplt r,sa,t
```

*(10)*

**Purpose:** To perform multiple parallel halfword signed or unsigned shift right operations.

**Description:** Each of the halfwords in GR \( r \) is shifted right \( sa \) bits. The completer, \( cmplt \), determines whether a signed or unsigned shift is performed. The completer is encoded in the \( se \) field of the instruction. (Table 7-5 defines the assembly language completer mnemonics.) The shift amount is between 0 and 15, and is encoded in the \( sa \) field in the instruction. The halfword results are placed in GR \( t \).

**Table 7-5. Halfword Parallel Shift Right Completers**

<table>
<thead>
<tr>
<th>cmplt</th>
<th>Description</th>
<th>se</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Unsigned Shift</td>
<td>2</td>
</tr>
<tr>
<td>&lt;none&gt; or S</td>
<td>Signed Shift</td>
<td>3</td>
</tr>
</tbody>
</table>

**Operation:**

```
parallel for (start ← 0; start <= 48; start += 16) {
    end ← start + 15;
    if (cmplt == U) /*se=2 (unsigned)*/
        GR[t]{start..end} ← rshift(GR[r1]{start..end},sa);
    else /*se=3 (signed)*/
        GR[t]{start..end} ← sign_ext_16(rshift(GR[r1]{start..end},sa),16–sa);
}
```

**Exceptions:** None.
Halfword Parallel Shift Right and Add

HSHRADD

Format: HSHRADD r1,sa,r2,t

(8)

<table>
<thead>
<tr>
<th>02</th>
<th>r2</th>
<th>r1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>sa</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>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To perform multiple halfword shift right and add operations in parallel with saturation.

Description: Each halfword of GR r1 is shifted right by sa bits, and then added to the corresponding halfword of GR r2. The bits shifted in equal the sign bit for each halfword. Signed saturation is performed, which forces each halfword result to either the maximum or the minimum value, if the result would have been out of range. The halfword results are placed in GR t. The shift amount is either 1, 2, or 3, and is encoded in the sa field of the instruction.

All operands are treated as signed numbers, and the results are signed numbers. Signed saturation is performed.

Operation: parallel for (start ← 0; start <= 48; start += 16) {
    end ← start + 15;
    GR[t]{start..end} ← sign_ext_16(rshift(GR[r1]{start..end},sa),16–sa) +
    GR[r2]{start..end};
    if (maximum_signed_saturation)
        GR[t]{start..end} ← 0x7FFF;
    else if(minimum_signed_saturation)
        GR[t]{start..end} ← 0x8000;
}

Exceptions: None.
Halfword Parallel Subtract

Format: \( \text{HSUB,cmplt} \ r1,r2,t \)

<table>
<thead>
<tr>
<th></th>
<th>02</th>
<th>r2</th>
<th>r1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>sat</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>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To subtract multiple halfwords in parallel with optional saturation.

Description: The corresponding halfwords of GR \( r2 \) are subtracted from the halfwords of GR \( r1 \) in parallel. Optional saturation is performed, which forces each halfword result to either the maximum or the minimum value, if the result would have been out of the range of the target format. The halfword results are placed in GR \( t \).

The completer, \( \text{cmplt} \), determines whether modular, signed-saturation, or unsigned-saturation arithmetic is performed. The completer is encoded in the sat field of the instruction (see Table 7-4 on page 7-58 for the assembly language completer mnemonics.) For signed saturation, all operands are treated as signed numbers, and the results are signed numbers. For unsigned saturation, the first operands, from GR \( r1 \), are treated as unsigned numbers, the second operands, from GR \( r2 \), are treated as signed numbers, and the results are unsigned numbers.

Operation: parallel for (start ← 0; start <= 48; start += 16) {
    end ← start + 15;
    GR[t]{start..end} ← (GR[r1]{start..end} + ~GR[r2]{start..end} + 1);
    switch (cmplt) {
        case SS: if (maximum_signed_saturation) /* sat=1 */
            GR[t]{start..end} ← 0x7FFF;
        else if (minimum_signed_saturation)
            GR[t]{start..end} ← 0x8000;
        break;
        case US: if (maximum_unsigned_saturation) /* sat=0 */
            GR[t]{start..end} ← 0xFFFF;
        else if (minimum_unsigned_saturation)
            GR[t]{start..end} ← 0x0000;
        break;
        default: /* sat=3 */
            break;
    }
}

Exceptions: None.
Insert Data TLB Translation

**IDTLBT**

**Format:**

<table>
<thead>
<tr>
<th>Format</th>
<th>IDTLBT</th>
<th>r1.r2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>r1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Purpose:** To add an entry to the data TLB.

**Description:** A slot is found in the data or combined TLB and the new translation is placed there. If the data or combined TLB already contains one or more entries whose virtual address ranges overlap the virtual address range of the new translation, the old entries are removed. The virtual address is specified by the IOR and ISR control registers. The contents of the ISR are concatenated with the lower 32 bits of the IOR to form the virtual address. The upper 32-bits of the IOR are ignored.

The physical address and the page size for the translation are specified by GR $r1$. The flags and access control bits are specified by GR $r2$.

**Operation:**

```plaintext
define privileged_operation_trap:
    if (priv != 0)
        privileged_operation_trap;
    else {
        space ← ISR;
        offset ← cat(ISR{32..63}, IOR{32..63});
        page_size ← 4096 << (2 * GR[r1]{60..63});
        for (i ← 0; i < page_size/4096; i++) {
            if (entry ← DTLB_search(space, offset + i*4096))
                DTLB_purge_local(entry);
        }
        entry ← DTLB_alloc(space, offset);
        DTLB[entry].VIRTUAL_ADDR ← (space<<32) | (offset);
        DTLB[entry].PHY_PAGE_NO ← GR[r1]{7..58};
        DTLB[entry].PAGE_SIZE ← GR[r1]{60..63};
        DTLB[entry].ACCESS_RIGHTS ← GR[r2]{5..11};
        DTLB[entry].ACCESS_ID ← GR[r2]{32..62};
        DTLB[entry].T ← GR[r2]{2}
        DTLB[entry].D ← GR[r2]{3}
        DTLB[entry].B ← GR[r2]{4}
        DTLB[entry].U ← GR[r2]{12}
        DTLB[entry].O ← GR[r2]{13}
        if (combined_TLB) {
            ITLB[entry].P ← GR[r2]{14}
        }
    }
```

**Exceptions:** Privileged operation trap
Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries into a combined TLB. The P bit is set to the appropriate bit of GR r in that case.

Note that no OR function is performed in creating the virtual address, since the IIA queue already contains a global address.

If smaller than 31-bit access IDs are implemented, only the appropriate number of the rightmost bits of GR[r][32..62] are stored in the TLB.
Insert Instruction TLB Translation

**Format:**

<table>
<thead>
<tr>
<th>01</th>
<th>r2</th>
<th>r1</th>
<th>0</th>
<th>20</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:**

To add an entry to the instruction TLB.

**Description:**

A slot is found in the instruction or combined TLB and the new translation is placed there. If the instruction or combined TLB already contains one or more entries whose virtual address ranges overlap the virtual address range of the new translation, the old entries are removed. The virtual address is specified by the front entry in the IIA queue. The contents of the front element of the IIASQ are concatenated with the lower 32 bits of the front element of the IIAOQ to form the virtual address. The upper 32-bits of the IIAOQ are ignored.

The physical address and the page size for the translation are specified by GR r1. The flags and access control bits are specified by GR r2.

**Operation:**

if (priv != 0)  
  privileged_operation_trap;
else {
  space ← IIASQ_Front;
  offset ← cat(IIASQ_Front{32..63},IIAOQ_Front{32..63});
  page_size ← 4096 << (2 * GR[r1]{60..63});
  for (i ← 0; i < page_size/4096; i++) {
    if (entry ← ITLB_search(space, offset + i*4096))
      ITLB_purge_local(entry);
  }
  entry ← ITLB_alloc(space,offset);
  ITLB[entry].VIRTUAL_ADDR ← (space<<32) | (offset);
  ITLB[entry].PHY_PAGE_NO ← GR[r1]{7..58};
  ITLB[entry].PAGE_SIZE ← GR[r1]{60..63};
  ITLB[entry].ACCESS_RIGHTS ← GR[r2]{5..11};
  ITLB[entry].ACCESS_ID ← GR[r2]{32..62};
  ITLB[entry].P ← GR[r2]{14}
  if (combined_TLB) {
    ITLB[entry].T ← GR[r2]{2}
    ITLB[entry].D ← GR[r2]{3}
    ITLB[entry].B ← GR[r2]{4}
    ITLB[entry].U ← GR[r2]{12}
    ITLB[entry].O ← GR[r2]{13}
  }
}

**Exceptions:**

Privileged operation trap
Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to insert both instruction entries and data entries into a combined TLB. The T, D, B, U, and O bits are set to the appropriate bits of GR r in that case.

Note that no OR function is performed in creating the virtual address, since the IIA queue already contains a global address.

If smaller than 31-bit access IDs are implemented, only the appropriate number of the rightmost bits of GR[r][32..62] are stored in the TLB.
Load Coherence Index

Format: \text{LCI} \ x(s,b),t

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
6 & 5 & 5 & 2 & 8 & 1 & 5 \\
\hline
\end{tabular}

Purpose: To determine the coherence index corresponding to a virtual address.

Description: The effective address is calculated. GR \text{t} receives the coherence index corresponding to the given virtual address.

In systems with separate data and instruction caches, the coherence index is obtained from the data cache.

The coherence index function is independent of the state of the PSW D-bit.

Operation: if (priv \neq 0)
\hspace{1em} \text{privileged_operation_trap;}
else {
\hspace{1em} space \leftarrow \text{space_select}(s,GR[b],\text{INDEXED});
\hspace{1em} offset \leftarrow GR[b] + GR[x];
\hspace{1em} GR[t] \leftarrow \text{coherence_index}(\text{space},\text{offset});
\}

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: All addresses within a page have the same coherence index.

The coherence index corresponding to a physical address can be determined by performing LCI on the equivalently-mapped virtual address. Also, in order to allow I/O modules to have coherent access to equivalently-mapped addresses without knowing the coherence index, the coherence index for equivalently-mapped addresses must be an implementation-defined function of the physical address bits only.

Two virtual addresses having the same coherence index are not guaranteed to alias unless they also meet the virtual aliasing rules.

For systems that do not have a cache, the target register receives an undefined value.

For system that do not support coherent I/O, this instruction is undefined.
Load Byte

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

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>b</th>
<th>t</th>
<th>s</th>
<th>im14</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>(5)</td>
<td>03</td>
<td>b</td>
<td>im5</td>
<td>s</td>
<td>a1</td>
</tr>
<tr>
<td>(4)</td>
<td>03</td>
<td>b</td>
<td>x</td>
<td>s</td>
<td>u0</td>
</tr>
</tbody>
</table>

**Purpose:** To load a byte into a general register.

**Description:**

The byte at the effective address is zero-extended and loaded into GR \( t \). The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 4), or a displacement \( d \). The displacement can be either long (Format 1) or short (Format 5). 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. 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, \( \text{cc} \), specifies the cache control hint (see Table 6-7 on page 6-10.)

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.

If base register modification is specified and \( b = t \), GR \( t \) receives the aligned byte at the effective address.
Operation: if (indexed_load)
    dx ← GR[x]; /* indexed (Format 4)*/
else if (d > 15 || d < -16) {
    dx ← sign_ext(assemble_16(s,im14),16); /* (Format 1) */
    cc ← NO_HINT;
} else
    dx ← low_sign_ext(im5,5); /* short displacement */

space ← space_select(s,GR[b],format);
space = 16

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;
}

GR[t] ← zero_ext(mem_load(space,offset,0,17,cc),8);
if (cmplt == O)
    enforce_ordered_load;

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

Restrictions: If the completer $O$ is specified, the displacement must be 0.
## Load and Clear Doubleword

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

<table>
<thead>
<tr>
<th></th>
<th>03</th>
<th>b</th>
<th>im5</th>
<th>s</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>5</th>
<th>m</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>(4)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To read and lock a doubleword semaphore in main memory.

**Description:**  
The effective address is calculated. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 4), or a displacement \( d \) (Format 5). 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, and optional index prescaling (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 6-9 on page 6-11.)

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. If base register modification is specified and \( b = t \), the value loaded is the aligned doubleword at the effective address.

The address must be 16-byte aligned. If the address is unaligned, the operation of the instruction is undefined.

The remaining steps of the instruction are indivisible and non-interruptible. The semaphore operation is strongly ordered.

If a cache control hint is not specified, the instruction is performed as follows:

- If the cache line containing the effective address is not present in the cache or is present but not dirty, and the system is not fully coherent, the line is flushed, the addressed doubleword is copied into GR \( t \), and then set to zero in memory. If the line is retained in the cache, it must not be marked as dirty.

- If the cache line containing the effective address is present in the cache and is dirty, or the system is fully coherent, the semaphore operation may be handled as above or may be optimized by copying the addressed doubleword into GR \( t \) and then setting the addressed doubleword to zero in the cache.

If a cache control hint is specified, the semaphore operation may be handled as if a cache control hint had not been specified, or, preferably, the addressed doubleword is copied into GR \( t \) and then the addressed doubleword is set to zero in the cache. The cleared
doubleword need not be flushed to memory.

Operation:

```c
if (indexed_load) /* indexed (Format 4)*/
    switch (cmplt) {
        case S:
        case SM:  dx ← lshift(GR[x],3);
                   break;
        case M:  dx ← GR[x];
                   break;
        default: dx ← GR[x];
    }
else /* short displacement */ /* (Format 5) */
    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;
    }
```

(indivisible)

```c
if (cache line is present and dirty || coherent_system || cc != 0) {
    GR[t] ← mem_load(space,offset,0,63,NO_HINT);
    mem_store(space,offset,0,63,NO_HINT,0);
} else {
    Dcache_flush(space, offset);
    GR[t] ← mem_load(space,offset,0,63,NO_HINT);
    store_in_memory(space,offset,0,63,NO_HINT,0);
}
```

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

Restrictions: All software users of a semaphore must access the semaphore using the same cache control hint. Sharing a semaphore using different cache control hints is undefined.
Load and Clear Word  

**LDCW**

<table>
<thead>
<tr>
<th>Format:</th>
<th>LDCW,cmplt,cc  xjd(s,b),t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>03</td>
</tr>
<tr>
<td>(5)</td>
<td>6</td>
</tr>
<tr>
<td>(4)</td>
<td>6</td>
</tr>
</tbody>
</table>

**Purpose:**
To read and lock a word semaphore in main memory.

**Description:**
The effective address is calculated. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 4), or a displacement \( d \) (Format 5.) 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. The completer also specifies base register modification, and optional index prescaling (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 6-9 on page 6-11.)

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. If base register modification is specified and \( b = t \), the value loaded is the aligned word at the effective address.

The address must be 16-byte aligned. If the address is unaligned, the operation of the instruction is undefined.

The remaining steps of the instruction are indivisible and non-interruptible. The semaphore operation is strongly ordered.

If a cache control hint is not specified, the instruction is performed as follows:

- If the cache line containing the effective address is not present in the cache or is present but not dirty, and the system is not fully coherent, the line is flushed, the addressed word is zero extended and copied into GR \( t \), and then set to zero in memory. If the line is retained in the cache, it must not be marked as dirty.

- If the cache line containing the effective address is present in the cache and is dirty, or the system is fully coherent, the semaphore operation may be handled as above or may be optimized by copying the addressed word into GR \( t \) (zero extended) and then setting the addressed word to zero in the cache.

If a cache control hint is specified, the semaphore operation may be handled as if a cache control hint had not been specified, or, preferably, the addressed word is zero extended and copied into GR \( t \) and then the addressed word is set to zero in the cache. The cleared word
need not be flushed to memory.

**Operation:**

```c
if (indexed_load) /* indexed (Format 4)*/
   switch (cmplt) {
      case S:          /* indexed (Format 4) */
         case SM:  dx ← lshift(GR[x],3);
                     break;
      case M:          /* indexed (Format 4) */
         default:  dx ← GR[x];
                     break;
   }
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;
   }
```

**(indivisible)**

```c
if (cache line is present and dirty || coherent_system || cc != 0) {
   GR[t] ← zero_ext(mem_load(space,offset,0,31,NO_HINT),32);
   mem_store(space,offset,0,31,NO_HINT,0);
} else {
   Dcache_flush(space, offset);
   GR[t] ← zero_ext(mem_load(space,offset,0,31,NO_HINT),32);
   store_in_memory(space,offset,0,31,NO_HINT,0);
}
```

**Exceptions:**
- Data TLB miss fault/data page fault
- TLB dirty bit trap
- Data memory access rights trap
- Page reference trap
- Data memory protection ID trap
- Data memory break trap

**Restrictions:**
- All software users of a semaphore must access the semaphore using the same cache control hint. Sharing a semaphore using different cache control hints is undefined.

**Notes:**
- Note that the “index shift” option for this instruction shifts by three, not two.
Load Doubleword

Format: \text{LDD,cmplt,cc} \ xjd(s,b),t

<table>
<thead>
<tr>
<th>Format</th>
<th>Operation</th>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3)</td>
<td></td>
<td>14</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>im10a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>(5)</td>
<td></td>
<td>03</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>im5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td>03</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t</td>
</tr>
</tbody>
</table>

Purpose: To load a doubleword into a general register.

Description: The aligned doubleword, at the effective address, is loaded into GR \( t \) from the effective address. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 4), or a displacement \( d \). The displacement can be either long (Format 3) or short (Format 5). 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. 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, \text{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.

If base register modification is specified and \( b = t \), GR \( t \) receives the aligned doubleword at the effective address.
Operation:  
    if (indexed_load) /* indexed (Format 4)*/
      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);
      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;
  }
  GR[t] ← mem_load(space,offset,0,63,cc);
  if (cmplt == O)
    enforce_ordered_load;

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

Restrictions:  For 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.
### Load Doubleword Absolute

**Format:**

\[
\text{LDDA,cmp}, cc \quad x|d(b), t
\]

<table>
<thead>
<tr>
<th>(5)</th>
<th>03</th>
<th>b</th>
<th>im5</th>
<th>0</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>4</th>
<th>m</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>11</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| (4) | 03 | b | x | 0 | u | 0 | cc | 4 | m | t |
|-----|----|----|---|---|---|----|---|---|---|
|     | 6  | 5  | 2 | 1 | 1 | 2 | 4  | 1 | 5 |   |

**Purpose:**

To load a doubleword into a general register from an absolute address.

**Description:**

The aligned doubleword at the effective absolute address is loaded into GR \( t \). The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 4), or a displacement \( d \) (Format 5). The displacement is encoded into the immediate field. Optional base modification can also be performed.

The completer, \( \text{cmp} \), 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 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.

If base register modification is specified and \( b = t \), GR \( t \) receives the aligned doubleword at the effective address. Protection is not checked when this instruction is executed. This operation is only defined if the address is aligned on an 8-byte boundary.
Operation: if (priv != 0)
    privileged_operation_trap;
else {
    if (indexed_load)
        switch (cmplt) {
            case S:
                case SM: dx ← lshift(GR[x],3);
                     break;
            case M:
            default: dx ← GR[x];
                     break;
        }
    else /* short displacement */
        dx ← low_sign_ext(im5,5);
        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;
        }
    GR[t] ← phys_mem_load(offset,0,63,cc);
    if (cmplt == O)
        enforce_ordered_load;
}

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level. If the completer O is specified, the displacement must be 0.
**Load Halfword**

**Format:**

```
LDH,cmplt,cc x|d(s,b),t
```

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>11 b t s im14</td>
</tr>
<tr>
<td>(5)</td>
<td>03 b im5 s a 1 cc 1 m t</td>
</tr>
<tr>
<td>(4)</td>
<td>03 b x s u 0 cc 1 m t</td>
</tr>
</tbody>
</table>

**Purpose:**

To load a halfword into a general register.

**Description:**

The aligned halfword, at the effective address, is zero-extended and loaded into GR \( t \) from the effective address. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 4), or a displacement \( d \). The displacement can be either long (Format 1) or short (Format 5). 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 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.

If base register modification is specified and \( b = t \), GR \( t \) receives the aligned halfword at the effective address.
Operation: if (indexed_load) switch (cmplt) { /* indexed (Format 4)*/
  case S:
  case SM:  dx ← lshift(GR[x],1);  
            break;
  case M:  default: dx ← GR[x];  
            break;
  }
else if (d > 15 || d < -16) { /* long displacement */
  dx ← sign_ext(assemble_16(s,im14),16);  /* (Format 1) */
  cc ← NO_HINT;
} else { /* short displacement */
  dx ← low_sign_ext(im5,5);  /* (Format 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;
  }
  GR[t] ← zero_ext(mem_load(space,offset,0,15,cc),16);
  if (cmplt == O)
    enforce_ordered_load;
}

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

Restrictions: If the completer \( O \) is specified, the displacement must be 0.
Load Immediate Left

Format: \texttt{LDIL \ i,t}

\begin{center}
\begin{tabular}{ccc}
    08 & t & im21 \\
    6 & 5 & 21
\end{tabular}
\end{center}

Purpose: To load the upper portion of a 32-bit immediate value into a general register.

Description: The 21-bit immediate value, \(i\), is assembled, shifted left 11 bits, sign extended, and placed in GR \(t\).

Operation: \(GR[t] \leftarrow \text{sign_ext(lshift(assemble_21(im21),11),32)}\);

Exceptions: None

Notes: Memory is not referenced.

---

Programming Note

LOAD IMMEDIATE LEFT can be used to generate a 32-bit literal in an arbitrary general register \(t\) by the following sequence of assembly language code:

\begin{verbatim}
LDIL  l%literal,GRt
LDO   r%literal(GRt),GRt
\end{verbatim}
Load Offset

Format: \( \text{LDO} \ d(b),t \)

(1)

<table>
<thead>
<tr>
<th>0D</th>
<th>b</th>
<th>t</th>
<th>i</th>
<th>im14</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>

Purpose: To load an offset into a general register.

Description: The effective address is calculated, and its offset part is loaded into \( \text{GR} \ t \). The displacement \( d \) is encoded into the immediate field.

Operation: \( \text{GR}[t] \leftarrow \text{GR}[b] + \text{sign_ext(assemble_16}(i,\text{im14}),16) \);

Exceptions: None

Notes: Memory is not referenced.

The LDI \( i,t \) pseudo-operation generates an LDO \( i(0),t \) instruction to load an immediate value into a register.

The COPY pseudo-operation allows for the movement of data from one register to another by generating the instruction LDO \( 0(r),t \).
Load Space Identifier  LDSID

Format:  LDSID (s,b),t

<table>
<thead>
<tr>
<th>00</th>
<th>b</th>
<th>rv</th>
<th>s</th>
<th>0</th>
<th>85</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To calculate the space register number referenced by an implicit pointer and copy the space register into a general register.

Description: If $s$ is zero, the space identifier referenced by GR $b$ is copied into GR $t$. If $s$ is not zero, SR $s$ is copied into GR $t$.

Operation: $GR[t] \leftarrow \text{space_select}(s, GR[b], \text{INDEXED})$;

Exceptions: None

Notes: Unimplemented space register bits must read as zero.

The target register receives an undefined value if LDSID with a nonzero $s$-field is executed at a nonzero privilege level when the PSW W-bit is 1.

This instruction provides no useful function when the PSW W-bit is 1, since the operating system is free to change the space register contents at any time.
**Load Word**

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

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>t</th>
<th>s</th>
<th>im14</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>12/13</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(2)</td>
<td>17</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(5)</td>
<td>03</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(4)</td>
<td>03</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Purpose:** To load a word into a general register.

**Description:** The aligned word, at the effective address, is zero-extended and loaded into GR \( t \) from the effective address. The offset is formed as the sum of a base register, \( b \), and either an index register, \( x \) (Format 4), or a displacement \( d \). The displacement can be either long (Formats 1 and 2) or short (Format 5.) 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. 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, \( \text{cc} \), specifies the cache control hint (see Table 6-7 on page 6-10.)

For long displacements with pre-decrement or post-increment, Format 1 (opcode 13) is used. For long displacements with post-decrement or pre-increment, Format 2 is used. For long displacements with no base modification, Format 1 (opcode 12) 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.

If base register modification is specified and \( b = t \), GR \( t \) receives the aligned word at the effective address.
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 */  
   if ((cmplt == MB && d >= 0) || (cmplt == MA && d < 0))  
      dx ← sign_ext(assemble_16a(s,im11a,i),16); /* (Format 2) */  
   else  
      dx ← sign_ext(assemble_16(s,im14),16); /* (Format 1) */  
   cc ← NO_HINT;  
} else /* short displacement */  
   dx ← low_sign_ext(im5,5); /* (Format 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;  
}  
GR[t] ← zero_ext(mem_load(space,offset,0,31,cc),32);  
if (cmplt == O)  
   enforce_ordered_load;

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

Restrictions:  
For post-decrement and pre-increment with long displacements (Format 2), only displacements which are multiples of four may be used.  
If the completer O is specified, the displacement must be 0.
### Load Word Absolute

**Purpose:** To load a word into a general register from an absolute address.

**Description:** The aligned word at the effective absolute address is zero-extended and loaded into GR $t$. The offset is formed as the sum of a base register, $b$, and either an index register, $x$ (Format 4), or a displacement $d$ (Format 5.) The displacement is encoded into the immediate field. Optional base modification can also be performed.

The completer, $\text{complt}$, 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 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.

If base register modification is specified and $b = t$, GR $t$ receives the aligned word at the effective address. Protection is not checked when this instruction is executed. This operation is only defined if the address is aligned on a 4-byte boundary.

<table>
<thead>
<tr>
<th>Format</th>
<th>LDWA,complt,cc x</th>
<th>d(b),t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5)</td>
<td>03</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>(4)</td>
<td>03</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
Operation: if (priv != 0)
    privileged_operation_trap;
else {
    if (indexed_load)
        switch (cmplt) {
            case S:
            case SM:
                dx ← lshift(GR[x], 2);
                break;
            case M:
            default:
                dx ← GR[x];
                break;
        } /* indexed (Format 4) */
    else /* short displacement */
        dx ← low_sign_ext(im5, 5); /* (Format 5) */
        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;
        }
    GR[t] ← zero_ext(phys_mem_load(offset, 0, 31, cc), 32);
    if (cmplt == O)
        enforce_ordered_load;
}

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level. If the completer O is specified, the displacement must be 0.
Load Physical Address

**Format:**

LPA,cmplt  x(s,b),t

(24)

<table>
<thead>
<tr>
<th>01</th>
<th>b</th>
<th>x</th>
<th>s</th>
<th>4D</th>
<th>m</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:**

To determine the absolute address of a mapped virtual page.

**Description:**

The effective address is calculated. The completer, cmplt, determines if the offset is the base register, b, or the base register plus index register x. The completer, encoded in the m field of the instruction, also specifies base register modification (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.) GR t receives the absolute address corresponding to the given virtual address. If the page is not present, a fault is taken and software sets the target register to 0. If base register modification is specified and b = t, the value loaded is the absolute address of the item indicated by the effective address.

In systems with separate data and instruction TLBs, the absolute address is obtained from the data TLB. This instruction performs data address translation regardless of the state of the PSW D-bit.

**Operation:**

if (priv != 0)
   privileged_operation_trap;
else {
   space ← space_select(s,GR[b],INDEXED);
   switch (cmplt) {
      case M: offset ← GR[b]; /*m=1*/
               GR[b] ← GR[b] + GR[x];
               break;
      default: offset ← GR[b] + GR[x]; /*m=0*/
               break;
   }
   if (DTLB_search(space,offset))
      GR[t] ← absolute_address(space,offset);
   else
      non-access_data_TLB_miss_fault();
}

**Exceptions:**

Non-access data TLB miss fault
Privileged operation trap

**Restrictions:**

The result of LPA is ambiguous for an address which maps to absolute address 0. This instruction may be executed only at the most privileged level.

**Notes:**

If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system’s handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the
IPSW[N] bit to 1, and return to the interrupting instruction.
Move From Control Register  

MFCTL

| Format:       | MFCTL r,t                             |
|              | MFCTL,W cr11,t                         |

<table>
<thead>
<tr>
<th>(32)</th>
<th>00</th>
<th>r</th>
<th>0</th>
<th>r</th>
<th>v</th>
<th>e</th>
<th>r</th>
<th>v</th>
<th>45</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Purpose: To move a value to a general register from a control register.

Description: CR r is copied into GR t.

Operation:
- if (r >= 1 && r <= 7)
  undefined;
- else if (priv != 0 && !(r == 11 || r == 26 || r == 27 || (r == 16 && !PSW[S])))
  privileged_register_trap;
- else if (r >= 17 && r <= 22)
  if (PSW[Q])
    undefined;
  else
    GR[t] ← CR[r]; /* IA Queues, IPRs */
- else if (r == 0)
  if (PSW[R])
    undefined;
  else
    GR[t] ← CR[r]; /* Recovery Counter */
- else if (r == 11)
  if (cmplt == W)
    GR[t] ← CR[r]; /* e=1*/
  else
    GR[t] ← CR[r][1..5]; /* SAR */
- else if (r >= 8)
  GR[t] ← CR[r]; /* other control registers */

Exceptions: Privileged register trap

Restrictions: System control registers other than the Shift Amount Register (CR 11), the Interval Timer (CR 16), and temporary registers CR 26 and CR 27, may be read only at the most privileged level. CR 11, CR 26, and CR 27 may be read at any privilege level. CR 16 may be read at any privilege level only if the PSW S-bit is 0; otherwise, CR 16 may be read only at the most privileged level. The Interruption Instruction Address Queues (CRs 17 and 18) and Interruption Parameter Registers (CRs 19, 20, and 21) and the Interruption Processor Status Word (CR 22) may be read reliably only when the PSW[Q] bit is 0.

If the completer W is specified, the control register source must specify CR11 (SAR.)
Move From Instruction Address

Format: MFIA t

<table>
<thead>
<tr>
<th>(32)</th>
<th>00</th>
<th>rv</th>
<th>0</th>
<th>rv</th>
<th>A5</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Purpose: To move the current instruction address to a general register.

Description: IAOQ_FRONT is copied into GR t. The rightmost two bits of GR t, corresponding to the privilege level, are zeroed.

Operation: GR[t] ← cat(IAOQ_FRONT[0..61],0{62..63});

Exceptions: None
Move From Space Register

<table>
<thead>
<tr>
<th>Format:</th>
<th>MFSP sr.t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(29)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>rv</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>s</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To move a value to a general register from a space register.

Description: SR sr (which is assembled from the s field in the instruction) is copied into GR t.

Operation: sr ← assemble_3(s);
GR[t] ← SR[sr];

Exceptions: None

Notes: Unimplemented space register bits must read as zero.
Mix Halfwords

Format: \[ \text{MIXH,cmplt} \quad r1,r2,t \]

(10)

<table>
<thead>
<tr>
<th>3E</th>
<th>r2</th>
<th>r1</th>
<th>1</th>
<th>ea</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To combine four halfwords from two source registers, and merge them in a result register.

Description: Two halfwords from GR \( r1 \) are merged with two halfwords from GR \( r2 \) and the result is placed in GR \( t \).

The completer, \( \text{cmplt} \), determines which halfwords are selected. The completer is encoded in the \( \text{ea} \) field of the instruction. (Table 7-6 defines the assembly language completer mnemonics.)

Table 7-6. Mix Instruction Completers

<table>
<thead>
<tr>
<th>cmplt</th>
<th>Description</th>
<th>ea</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Left Halfwords/Words are combined</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>Right Halfwords/Words are combined</td>
<td>2</td>
</tr>
</tbody>
</table>

If \( \text{cmplt} \) is “L”, the left halfword of each of the four input words is merged into the result. If \( \text{cmplt} \) is “R”, the right halfword of each of the four input words is merged into the result.

The two cases are shown in the following diagram:

![Diagram](/path/to/diagram.png)

Operation: switch (cmplt) {
  case L:  \[ \text{GR}[t] \leftarrow \text{cat}(	ext{GR}[r1][0..15], \text{GR}[r2][0..15], \text{GR}[r1][32..47], \text{GR}[r2][32..47]); \] /*\( ea=0 */
  break;
  case R:  \[ \text{GR}[t] \leftarrow \text{cat}(	ext{GR}[r1][16..31], \text{GR}[r2][16..31], \text{GR}[r1][48..63], \text{GR}[r2][48..63]); \] /*\( ea=2 */
  break;
}

Exceptions: None.
Mix Words

**Format:** MIXW.cmplt r1,r2,t

(10)

<table>
<thead>
<tr>
<th>3E</th>
<th>r2</th>
<th>r1</th>
<th>1</th>
<th>ea</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

**Purpose:** To combine two words from two source registers, and merge them in a result register.

**Description:** A word from GR \( r1 \) is merged with a word from GR \( r2 \) and the result is placed in GR \( t \).

The completer, \( cmplt \), determines which words are selected. The completer is encoded in the \( ea \) field of the instruction (see Table 7-6 on page 7-94 for the assembly language completer mnemonics.) If \( cmplt \) is “L”, the left word of each of the two input doublewords is merged into the result. If \( cmplt \) is “R”, the right word of each of the two input doublewords is merged into the result. The two cases are shown in the following diagram:

![Diagram of Mix Words operation]

**Operation:**

\[
\text{switch (cmplt) }
\begin{array}{l}
\text{case L:} \\
\quad \text{GR}[t] \leftarrow \text{cat} (\text{GR}[r1]{0..31}, \text{GR}[r2]{0..31}); \\
\quad \text{/*ea=0*/ break;}
\end{array}
\begin{array}{l}
\text{case R:} \\
\quad \text{GR}[t] \leftarrow \text{cat} (\text{GR}[r1]{32..63}, \text{GR}[r2]{32..63}); \\
\quad \text{/*ea=2*/ break;}
\end{array}
\]

**Exceptions:** None.
Move and Branch

MOVB

Format: MOVB,cond,n r1,r2,target

(17)

<table>
<thead>
<tr>
<th></th>
<th>r2</th>
<th>r1</th>
<th>c</th>
<th>w1</th>
<th>n</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To copy one register to another and perform an IA-relative branch conditionally based on the value moved.

Description: GR r1 is copied into GR r2. If the value moved satisfies the specified condition, cond, the word displacement is assembled from the w and w1 fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The condition is encoded in the c field of the instruction. The branch target, target, in the assembly language format is encoded in the w and w1 fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The ,N completer, encoded in the n field of the instruction, specifies nullification.

Conditions: The condition, cond, is any of the extract/deposit 32-bit conditions shown in Table D-13 on page D-9 (never, =, <, OD, TR, <>, >=, EV). When a condition completer is not specified, the “never” condition is used. The boolean variable “cond_satisfied” in the operation section is set to 1 when the value moved satisfies the specified condition and set to 0 otherwise.

Operation: GR[r2] ← GR[r1];
disp ← lshift(sign_ext(assemble_12(w1,w),12),2);
if (cond_satisfied)
   IAOQ_Next ← IAOQ_Front + disp + 8;
if (n)
   if (disp < 0)
      PSW[N] ← !cond_satisfied;
   else
      PSW[N] ← cond_satisfied;

Exceptions: Taken branch trap
Move Immediate and Branch

Format: MOVIB,cond,n i,r,target

<table>
<thead>
<tr>
<th></th>
<th>33</th>
<th>r</th>
<th>im5</th>
<th>c</th>
<th>w1</th>
<th>n</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Purpose: To copy an immediate value into a register and perform an IA-relative branch conditionally based on the value moved.

Description: The immediate value \( im5 \) is sign extended and copied into GR \( r \). If the value moved satisfies the specified condition, \( cond \), the word displacement is assembled from the \( w \) and \( w1 \) fields, sign extended, and added to the current instruction offset plus 8 to form the target offset. The condition is encoded in the \( c \) field of the instruction. The branch target, \( target \), in the assembly language format is encoded in the \( w \) and \( w1 \) fields.

If nullification is not specified, the following instruction is not nullified. If nullification is specified, the instruction following a taken forward branch or a failing backward branch is nullified. The \( .N \) completer, encoded in the \( n \) field of the instruction, specifies nullification.

Conditions: The condition, \( cond \), is any of the extract/deposit 32-bit conditions shown in Table D-13 on page D-9 (never, =, <, OD, TR, <> , >=, EV ). When a condition completer is not specified, the “never” condition is used. The boolean variable “cond_satisfied” in the operation section is set to 1 when the value moved satisfies the specified condition and set to 0 otherwise.

Operation: \[
\text{GR}[r] \leftarrow \text{low\_sign\_ext}(im5,5); \\
disp \leftarrow \text{lshift}(\text{sign\_ext}(\text{assemble\_12}(w1,w),12),2); \\
\text{if (cond\_satisfied)} \\
\quad \text{IAOQ\_Next} \leftarrow \text{IAOQ\_Front} + \text{disp} + 8; \\
\text{if (n)} \\
\quad \text{if (disp < 0)} \\
\quad \quad \text{PSW}[N] \leftarrow \neg \text{cond\_satisfied}; \\
\quad \text{else} \\
\quad \quad \text{PSW}[N] \leftarrow \text{cond\_satisfied};
\]

Exceptions: Taken branch trap

Programming Note
Since \( i \) is known at the time a MOVE IMMEDIATE AND BRANCH instruction is written, conditions other than always and never (the \( .TR \) and \( <\text{none}> \) completers) are of no use.
Move To Control Register

Format:  MTCTL  r,t

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>t</th>
<th>r</th>
<th>rv</th>
<th>C2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Purpose:  To move a value from a general register to a control register.

Description:  GR r is copied into CR t. If CR 23 is specified, then the value is first complemented and ANDed with the original value.

Notes:  The MTSAR r pseudo-operation generates an MTCTL r,%SAR instruction to copy a general register to the Shift Amount Register.
Operation:
if (t \geq 1 \&\& t \leq 7)
    undefined;
else if (t \neq 11 \&\& priv \neq 0)
    privileged_register_trap;
else
    switch(t) {
        case 0:
            if (PSW[R])
                undefined;
            else
                CR[t] \leftarrow GR[r]{32..63}; /* Recovery Counter */
            break;
        case 14: case 15: case 16: case 24: case 25: case 26:
            case 27: case 28: case 29: case 30: case 31:
                CR[t] \leftarrow GR[r]; /* other control registers */
            break;
        case 17: case 18: case 20: case 21: case 22:
            if (PSW[Q])
                undefined;
            else
                CR[t] \leftarrow GR[r]; /* IIA Queues, IOR, ISR */
            break;
        case 23:
            CR[23] \leftarrow CR[23] \& \sim GR[r]; /* EIRR */
            break;
        case 10:
            CR[10] \leftarrow GR[r]{48..63}; /* CCR, SCR */
            break;
        case 11:
            CR[11] \leftarrow GR[r]{26..31}; /* SAR */
            break;
        case 8: case 9: case 12: case 13:
            CR[t] \leftarrow GR[r]; /* Protection Identifiers */
            break;
        case 19:
            undefined; /* IIR */
            break;
    }

Exceptions: Privileged register trap

Restrictions: System control registers other than the Shift Amount Register (CR 11) may be written only at the most privileged level. CR 11 may be written at any privilege level. The Recovery Counter (CR 0) may be written reliably only when the PSW[R] bit is 0. Writing into the Interruption Instruction Register (CR 19) is an undefined operation. Writing into the Interruption Instruction Address Queues (CRs 17 and 18), the Interruption Processor Status Word (CR 22), the Interruption Offset Register (CR 21) or the Interruption Space Register (CR 20) when the PSW[Q] bit is 1 is an undefined operation.

Notes: The MTSAR pseudo-operation generates an MTCTL r,CR11 to copy a general register to the Shift Amount Register (CR 11.)
Move To Shift Amount Register Complement  

**Format:**  
MTSARCM  r

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To take the one’s complement of a value from a general register and move it to the Shift Amount Register (CR11.)

**Description:** The one’s complement of GR \( r \) is copied into CR[11].

**Operation:**  
\[ \text{CR}[11] \leftarrow \sim \text{GR}[r][26..31]; \]

**Exceptions:** None.

**Notes:** The upper bits of the SAR are non-existent and so on a MFCTL instruction with the SAR as the specified register, hardware can return either 0’s or what was last written for the upper 58 bits. If hardware returns what was last written, the value written by a MTSARCM instruction must be the complement of GR[\( r \)].
Move To System Mask

Format: \[
\begin{array}{c}
\text{MTSM} \ r \\
\end{array}
\] (33)

Purpose: To set PSW system mask bits to a value from a register.

Description: Bits 36, 37 and 56..63 of GR \(r\) replace the system mask, \(\text{PSW}\{36,37,56..63\}\). Setting the PSW Q-bit, \(\text{PSW}\{60\}\), to 1 with this instruction, if it was not already 1, is an undefined operation.

Operation:
\[
\begin{array}{l}
\text{if (priv \(!=\) 0)}\\
\quad \text{privileged\_operation\_trap;}\\
\text{else} \\
\quad \text{if } ((\text{PSW}[Q] == 0) \&\& (\text{GR}[r][60] == 1)) \\
\qquad \text{undefined;}\\
\quad \text{else} \\
\qquad \text{PSW}[W] \leftarrow \text{GR}[r][36];\\
\qquad \text{PSW}[E] \leftarrow \text{GR}[r][37];\\
\qquad \text{PSW}[O] \leftarrow \text{GR}[r][56];\\
\qquad \text{PSW}[G] \leftarrow \text{GR}[r][57];\\
\qquad \text{PSW}[F] \leftarrow \text{GR}[r][58];\\
\qquad \text{PSW}[R] \leftarrow \text{GR}[r][59];\\
\qquad \text{PSW}[Q] \leftarrow \text{GR}[r][60];\\
\qquad \text{PSW}[P] \leftarrow \text{GR}[r][61];\\
\qquad \text{PSW}[D] \leftarrow \text{GR}[r][62];\\
\qquad \text{PSW}[I] \leftarrow \text{GR}[r][63];\\
\end{array}
\]

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: The state of the IPRs, IIA queues, and the IPSW is undefined when this instruction is used to set the Q-bit to 0, if it was not already 0.
Move To Space Register

Format: MTSP r, sr

(29)  

<table>
<thead>
<tr>
<th></th>
<th>rv</th>
<th>r</th>
<th>s</th>
<th>C1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To move a value from a general register to a space register.

Description: GR r is copied into SR sr (which is assembled from the s field in the instruction.)

Operation:
\[
\text{sr} \leftarrow \text{assemble}_3(s); \\
\text{if (sr >= 5 && priv != 0)} \\
\quad \text{privileged_register_trap}; \\
\text{else} \\
\quad \text{if (PSW}[W]) \\
\quad \quad \text{SR}[sr] \leftarrow \text{GR}[r]; \\
\text{else} \\
\quad \text{SR}[sr]\{32..63\} \leftarrow \text{GR}[r]\{32..63\};
\]

Exceptions: Privileged register trap

Restrictions: SRs 5, 6 and 7 may be changed only by software running at the most privileged level.

Notes: The values written to unimplemented space register bits must be ignored.

Bits 0..31 of the target space register, if implemented, are unchanged by this instruction if the PSW W-bit is 0.
Inclusive OR

Format: OR,cond r1,r2,t

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

Purpose: To do a 64-bit, bitwise inclusive OR.

Description: GR r1 and GR r2 are ORed and the result is placed in GR t. The following instruction is nullified if the values ORed satisfy the specified condition, cond. The condition is encoded in the c, d, and f fields of the instruction.

Conditions: The condition is any of the 32-bit logical conditions shown in Table D-9 on page D-7 or any of the 64-bit logical conditions shown in Table D-10 on page D-7. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values ORed satisfy the specified condition.

Operation: GR[t] ← GR[r1] | GR[r2];
if (cond_satisfied) PSW[N] ← 1;

Exceptions: None

Notes: The NOP pseudo-operation generates the instruction OR 0,0,0.
Purge Data Cache

**Format:**

PDC.cmplt \( x(s,b) \)

<table>
<thead>
<tr>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

**Purpose:** To invalidate a data cache line.

**Description:** The cache line (if present) specified by the effective address generated by the instruction is invalidated from the data cache. If the privilege level is non-zero and the cache line is dirty then it is written back to memory before being invalidated. If the privilege level is zero and the line is dirty then the implementation may optionally write back the line to memory.

The completer, \( cmplt \), determines if the offset is the base register, \( b \), or the base register plus the index register \( x \). The completer, encoded in the \( m \) field of the instruction, specifies base register modification (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.)

If a cache purge operation is performed, write access to the data is required and a special access rights check is performed. See “Access Control” on page 3-11. The PSW D-bit (Data address translation enable) determines whether a virtual or absolute address is used.

In a multiprocessor system, a purge or flush request is broadcast to all data and combined caches.

**Operation:**

space ← space_select(s,GR\([b]\),INDEXED);

```c
switch (cmplt) {
    case M: offset ← GR[b]; /*m=1*/
        GR[b] ← GR[b] + GR[x];
        break;
    default: offset ← GR[b] + GR[x]; /*m=0*/
        break;
}
```

if (priv != 0)
    Dcache_flush(space,offset);
else
    Dcache_flush_or_purge(space,offset);

**Exceptions:**

- Non-access data TLB miss fault
- Data memory break trap
- Data memory access rights trap
- Data memory protection ID trap

**Notes:**

- For systems that do not have a cache, this instruction executes as a null instruction.
- At privilege level zero, implementations are encouraged to purge the cache line for performance reasons.
- This instruction may be executed out of sequence but must satisfy the instruction ordering.
constraints. The SYNC instruction enforces program order with respect to the instructions following the SYNC.

It is an undefined operation to execute a PDC with a nonzero s-field at a nonzero privilege level when the PSW W-bit is 1.
Purge Data TLB

**Format:**

\[
\text{PDTLB,scope,cmpl}
\]

\[
x(s,b)
\]

(24)

\[
\begin{array}{cccccccc}
01 & b & x & s & 2 & e1 & 8 & m & rv \\
6 & 5 & 5 & 2 & 3 & 1 & 4 & 1 & 5 \\
\end{array}
\]

**Purpose:**
To invalidate a data TLB entry.

**Description:**
The data or combined TLB entries (if any) which match the effective address generated by the instruction are removed. The completer, **scope**, encoded in the **e1** field of the instruction, specifies whether the purge is global to all processors in a multiprocessor system (no completer, **e1=0**) or limited to the local processor (**L** completer, **e1=1**.) The completer, **cmplt**, encoded in the **m** field of the instruction, determines if the offset is the base register, **b**, or the base register plus index register, **x**, and whether base register modification is performed (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.)

TLB purges are strongly ordered. In a multiprocessor system, a global TLB purge causes a purge request to be broadcast to all data and combined TLBs. The other processors must remove all matching entries before the issuing processor continues.

**Operation:**
\[
\text{if (priv !}=0) \\
\quad \text{privileged_operation_trap;} \\
\text{else }
\]

\[
\quad \text{space} \leftarrow \text{space_select}(s,GR[b],INDEXED); \\
\quad \text{switch (cmplt)} \\
\quad \quad \text{case M: offset} \leftarrow \text{GR}[b]; \\
\quad \quad \text{GR}[b] \leftarrow \text{GR}[b] + \text{GR}[x]; \\
\quad \quad \text{break;} \\
\quad \quad \text{default: offset} \leftarrow \text{GR}[b] + \text{GR}[x]; \\
\quad \quad \text{break;}
\]

\[
\text{page_size} \leftarrow 4096 \times (2 \times \text{GR}[b](60..63)); \\
\text{for (i} \leftarrow 0; i < \text{page_size/4096; i}++) \\
\quad \text{if (entry} \leftarrow \text{DTLB_search(space, offset + i*4096))} \\
\quad \quad \text{DTLB_purge_local(entry);} \\
\quad \text{if (scope} \neq \text{L}) \\
\quad \quad \text{DTLB_purge_broadcast(space,offset,page_size);} \\
\]

**Exceptions:**
Privileged operation trap.

**Restrictions:**
This instruction may be executed only at the most privileged level.

**Notes:**
This instruction may be used to purge both instruction entries and data entries from a combined TLB.
Purge Data TLB Entry

Format: PDTLBE,cmplt x(s,b)

Purpose: To invalidate a data TLB entry without matching the address portion.

Description: The data or combined TLB entries (if any) specified by an implementation-dependent function of the effective address generated by the instruction are removed. All the fields of these entries may be changed to arbitrary values as long as these entries do not validate any subsequent accesses. The completer, cmplt, determines if the offset is the base register, b, or the base register plus the index register x. The completer, encoded in the m field of the instruction, specifies base register modification (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.)

This is an implementation-dependent instruction that can be used to purge the entire data TLB without knowing the translations in the TLB. No broadcast occurs in a multiprocessor system.

Operation: if (priv != 0)

privileged_operation_trap;
else {
    space ← space_select(s,GR[b],INDEXED);
    switch (cmplt) {
        case M: offset ← GR[b]; /*m=1*/
            GR[b] ← GR[b] + GR[x];
            break;
        default: offset ← GR[b] + GR[x]; /*m=0*/
            break;
    }
    DTLB_purge_entries(space,offset);
}

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries from a combined TLB. This instruction does not necessarily purge the entry specified by “space” and “offset”.

PA-RISC 2.0 Architecture Instruction Descriptions 7-107
Permuted Halfwords

**Format:**

PERMH,c r,t

---

**Purpose:**

To select any combination of four halfwords from a source register, and place that combination in a result register.

**Description:**

The source register, GR\(r\), is treated as four 16-bit fields. A 64-bit result is generated, consisting of four 16-bit fields. Each field in the result is independently selected from one of the fields in GR\(r\). The result is placed in GR\(t\).

The choice of which fields are selected for each result field is specified by the completer \(c\), which is given as a four-digit number, where each digit is either 0, 1, 2, or 3. Each digit controls the selection for one result field. For example, a digit value of 0 specifies that the result field receives the value from the leftmost source field, and a digit value of 2 selects the next-to-rightmost source field. \(c\) is encoded in the \(c0\), \(c1\), \(c2\), and \(c3\) fields of the instruction. \(c0\) encodes the first digit, \(c1\) the second, etc. Thus, \(c0\) encodes which source field will appear in the leftmost result field, \(c1\) which source field will appear in the next-to-leftmost result field, etc.

The array of boolean variables “\(c[]\)” in the operation section represent the completer \(c\). The variable “\(c[0]\)” represents the first digit in \(c\), “\(c[1]\)” the second digit in \(c\), etc.

Any combination or permutation of the four source fields can be generated.

**Operation:**

parallel for (i ← 0; i <= 3; i++) {
    start ← 16 * i;
    end ← start + 15;
    field_select ← c[i];
    switch (field_select) {
        case 0: GR[t]{start..end} ← GR[r]{0..15};
            break;
        case 1: GR[t]{start..end} ← GR[r]{16..31};
            break;
        case 2: GR[t]{start..end} ← GR[r]{32..47};
            break;
        case 3: GR[t]{start..end} ← GR[r]{48..63};
            break;
    }
}

**Exceptions:**

None.

**Notes:**

The source register specifier, \(r\), must appear in both source operand fields of the instruction, as shown in the format. If it does not, the operation is undefined.
Programming Note
The following figures illustrate examples of how the source fields are specified and how the result is generated.

\[
\begin{array}{cccc}
\text{r} & 0 & 1 & 2 & 3 \\
& a & b & c & d \\
\text{c= 0000} & \\
\text{t} & a & a & a & a \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{r} & 0 & 1 & 2 & 3 \\
& a & b & c & d \\
\text{c= 2031} & \\
\text{t} & c & a & d & b \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{r} & 0 & 1 & 2 & 3 \\
& a & b & c & d \\
\text{c= 3210} & \\
\text{t} & d & c & b & a \\
\end{array}
\]
Purge Instruction TLB

**Format:**

\[
\text{PITLB}, \text{scope}, \text{cmplt} \quad x(s,r,b)
\]

(26)

\[
\begin{array}{cccccccc}
01 & b & x & s & 0 & e1 & 8 & m & rv \\
6 & 5 & 5 & 3 & 2 & 1 & 4 & 1 & 5
\end{array}
\]

**Purpose:**

To invalidate an instruction TLB entry.

**Description:**

The instruction or combined TLB entry (if any) for the page specified by the effective address generated by the instruction is removed. The completer, `scope`, encoded in the `el` field of the instruction, specifies whether the purge is global to all processors in a multiprocessor system (no completer, `el=0`) or limited to the local processor (`L`, completer, `el=1`). The completer, `cmplt`, encoded in the `m` field of the instruction, determines if the offset is the base register, `b`, or the base register plus index register, `x`, and whether base register modification is performed (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.)

TLB purges are strongly ordered. In a multiprocessor system, a global TLB purge causes a purge request to be broadcast to all instruction and combined TLBs. The other processors must remove the entry before the issuing processor continues.

**Operation:**

if (priv != 0)

privileged_operation_trap;
else {

space ← SR[assemble_3(s)];

switch (cmplt) {

    case M: offset ← GR[b]; /* m=1 */
        GR[b] ← GR[b] + GR[x];
        break;

    default: offset ← GR[b] + GR[x]; /* m=0 */
        break;

}

page_size ← 4096 << (2 * GR[b]{60..63});

for (i ← 0; i < page_size/4096; i++) {

    if (entry ← ITLB_search(space, offset + i*4096))
        ITLB_purge_local(entry);

}

if (scope != L)

    ITLB_purge_broadcast(space,offset,page_size);
}

**Exceptions:**

Privileged operation trap

**Restrictions:**

This instruction may be executed only at the most privileged level.

**Notes:**

This instruction may be used to purge both instruction entries and data entries from a combined TLB.

7-110 Instruction Descriptions PA-RISC 2.0 Architecture
Purge Instruction TLB Entry

Format:     PITLBE,cmplt x(sr,b)

(26)

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>x</th>
<th>s</th>
<th>09</th>
<th>m</th>
<th>rv</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To invalidate an instruction TLB entry without matching the address portion.

Description: The instruction or combined TLB entries (if any) specified by an implementation-dependent function of the effective address generated by the instruction are removed. All the fields of these entries may be changed to arbitrary values as long as these entries do not validate any subsequent accesses. The completer, \textit{cmplt}, determines if the offset is the base register, \(b\), or the base register plus the index register \(x\). The completer, encoded in the \(m\) field of the instruction, specifies base register modification (see Table 7-3 on page 7-52 for the assembly language completer mnemonics.) The space register, \(sr\), is encoded in the \(s\) field of the instruction.

This is an implementation-dependent instruction that can be used to purge the entire instruction TLB without knowing the translations in the TLB. No broadcast occurs in a multiprocessor system.

Operation: if (priv != 0)

\[
\text{privileged_operation_trap;}
\]
else {

\[
\text{space } \leftarrow \text{SR}[\text{assemble}_3(s)];
\]
switch (cmplt) {

\[
\text{case M: offset } \leftarrow \text{GR}[b]; \quad /*m=1*/
\]
\[
\text{GR}[b] \leftarrow \text{GR}[b] + \text{GR}[x];
\]
break;

\[
\text{default: offset } \leftarrow \text{GR}[b] + \text{GR}[x]; \quad /*m=0*/
\]
break;

\]

\[
\text{ITLB_purge_entries(space,offset);}
\]
}

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Notes: This instruction may be used to purge both instruction entries and data entries from a combined TLB. This instruction does not necessarily purge the entry specified by “space” and “offset”. 

PA-RISC 2.0 Architecture

Instruction Descriptions 7-111
Pop Branch Target Stack

**Format:**

<table>
<thead>
<tr>
<th>POPBTS</th>
<th>i</th>
</tr>
</thead>
</table>

(23)

<table>
<thead>
<tr>
<th>3A</th>
<th>0</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>i</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Purpose:** To pop one or more entries off the branch target stack.

**Description:** The top i entries of the branch target stack are popped.

If this instruction is nullified, the results are undefined.

This instruction is executed as a NOP on machines that do not implement the branch target stack.

**Operation:**

```c
for (j=i; j>0; j--) {
    pop_from_BTS();
}
```

**Exceptions:** None
Probe Access

**Format:** PROBE.cmplt (s,b),r,t

<table>
<thead>
<tr>
<th>01</th>
<th>b</th>
<th>r</th>
<th>s</th>
<th>23</th>
<th>e1</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To determine whether read or write access to a given address is allowed.

**Description:** A test is performed to determine if access to the address computed by the instruction is permitted at the privilege level given by the two rightmost bits of the GR r. GR t is set to 1 if the test succeeds and 0 otherwise.

The completer, cmplt, encoded in the sub-operation field e1, specifies whether the instruction checks for read (cmplt == R) or write (cmplt == W) access (e1=0: check for read access, e1=1: check for write access.) If the PSW P-bit is 1, the protection IDs are also checked. The instruction performs data address translation regardless of the state of the PSW D-bit.

**Operation:**

```plaintext
space ← space_select(s,GR[b],INDEXED);
offset ← GR[b];
if (DTLB_search(space,offset))
    switch (cmplt) {
        case W: if (write_access_allowed(space,offset,GR[r])) /*e1=1*/
                   GR[t] ← 1;
            else
                   GR[t] ← 0;
            break;
        case R:
            default: if (read_access_allowed(space,offset,GR[r])) /*e1=0*/
                      GR[t] ← 1;
                   else
                      GR[t] ← 0;
            break;
    }
else
    non-access_data_TLB_miss_fault();
```

**Exceptions:** Non-access data TLB miss fault/non-access data page fault

**Notes:** If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system’s handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the IPSW[N] bit to 1, and return to the interrupting instruction.

It is an undefined operation to execute a PROBE with a nonzero s-field at a nonzero privilege level when the PSW W-bit is 1.
Probe Access Immediate

**Format:**

```
PROBEI,cmplt (s,b),i,t
```

<table>
<thead>
<tr>
<th>01</th>
<th>b</th>
<th>i</th>
<th>s</th>
<th>63</th>
<th>e1</th>
<th>0</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To determine whether read or write access to a given address is allowed.

**Description:** A test is performed to determine if access to the address computed by the instruction is permitted at the privilege level given by the two rightmost bits of the immediate value \( i \). GR \( t \) is set to 1 if the test succeeds and 0 otherwise.

The completer, \( cmplt \), encoded in the sub-operation field \( e1 \), specifies whether the instruction checks for read (\( cmplt == R \)) or write (\( cmplt == W \)) access (\( e1 == 0 \): check for read access, \( e1 == 1 \): check for write access.) If the PSW P-bit is 1, the protection IDs are also checked. This instruction performs data address translation regardless of the state of the PSW D-bit.

**Operation:**

```
space ← space_select(s,GR[b],INDEXED);
offset ← GR[b];
if (DTLB_search(space,offset))
  switch (cmplt) {
    case W: if (write_access_allowed(space,offset,i)) /*e1=1*/
      GR[t] ← 1;
      else
      GR[t] ← 0;
      break;
    case R:
      default: if (read_access_allowed(space,offset,i)) /*e1=0*/
      GR[t] ← 1;
      else
      GR[t] ← 0;
      break;
  }
else
  non-access_data_TLB_miss_fault();
```

**Exceptions:** Non-access data TLB miss fault/non-access data page fault

**Notes:** If this instruction causes a non-access data TLB miss fault/non-access data page fault, the operating system’s handler is required to search its page tables for the given address. If found, it does the appropriate TLB insert and returns to the interrupting instruction. If not found, the handler must decode the target field of the instruction, set that GR to 0, set the IPSW[N] bit to 1, and return to the interrupting instruction.

It is an undefined operation to execute a PROBEI with a nonzero \( s \)-field at a nonzero privilege level when the PSW W-bit is 1.
Push Branch Target Stack  

**Format:**  
\[
\text{PUSHBTS } \ r \\
\]

(23)  

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3A</td>
<td>0</td>
<td>r</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:** To push a value from a GR onto the branch target stack.

**Description:** Either the value in GR \( r \), or an “invalid” value is pushed onto the branch target stack.

If this instruction is nullified, the results are undefined.

This instruction is executed as a NOP on machines that do not implement the branch target stack.

**Operation:**  
\[
\text{push\_onto\_BTS(GR[b]{0..61});} \\
\]

**Exceptions:** None.
Push Nominated

Format: PUSHNOM

(23) 3A 0 0 2 0 0 0 0 1

Purpose: To push the currently nominated address onto the branch target stack.

Description: If there is a current nominated value, it is pushed onto the branch target stack. Otherwise, an “invalid” value is pushed.

If this instruction is nullified, the results are undefined.

This instruction is executed as a NOP on machines that do not implement the branch target stack.

Operation: push_onto_BTS(BNR);

Exceptions: None.
Return From Interruption

Format: RFI, cmplt

<table>
<thead>
<tr>
<th></th>
<th>rv</th>
<th>rv</th>
<th>rv</th>
<th>6</th>
<th>e1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

(33)

Purpose: To restore processor state and restart execution of an interrupted instruction stream and optionally restore GRs 1, 8, 9, 16, 17, 24, and 25 from the shadow registers.

Description: The PSW register contents are restored from the IPSW register but are not modified by this instruction. The IA queues are restored from the IIA queues. Execution continues at the locations loaded into the IA queues.

The completer, cmplt, encoded in the sub-operation field e1, specifies whether the contents of GRs 1, 8, 9, 16, 17, 24, and 25 are restored from the shadow registers (e1=5: restore from shadow registers, e1=0: GRs are unchanged.) Execution of an RFI with the .R completer when the contents of the shadow registers are undefined leaves the contents of GRs 1, 8, 9, 16, 17, 24, and 25 undefined. After execution of an RFI with the .R completer, the SHRs are undefined.

Execution of an RFI with the IPSW Q-bit equal to 0 returns to the location specified by the IIA queues, but leaves the IIAOQ, IIASQ, and IPRs undefined. Software is responsible for avoiding interruptions during the execution of an RFI. Execution of an RFI instruction when any of the PSW Q, I, or R bits are ones is an undefined operation. Execution of an RFI instruction when the PSW L-bit is a one is an undefined operation if the new privilege level after execution of the RFI is non zero.
Operation:  
if (priv != 0)
    privileged_operation_trap;
else {
    if (cmplt == R) {
        GR[1] ← SHR[0]; /*e1=5*/
        GR[8] ← SHR[1];
        GR[9] ← SHR[2];
        GR[16] ← SHR[3];
        GR[17] ← SHR[4];
        GR[24] ← SHR[5];
        GR[25] ← SHR[6];
    } else /* do nothing */ /*e1=0*/
    PSW ← IPSW;
    IAOQ_Back ← IIAOQ_Back;
    IAOQ_Front ← IIAOQ_Front;
    if (!level_0) {
        IASQ_Back ← IIASQ_Back & ~zero_ext(IIAOQ_Back{0..31},32);
        IASQ_Front ← IIASQ_Front & ~zero_ext(IIAOQ_Front{0..31},32);
    }
}

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.

Because this instruction restores the state of the execution pipeline, it is possible for software to place the processor in states which could not result from the execution of any sequence of instructions not involving interruptions. For example, it could set the PSW B-bit to 0 even though the addresses in the IA queues are not contiguous. The operation of the machine is undefined in such cases, and it is the responsibility of software to avoid them.

To avoid improper processor states, software must not set the PSW B-bit to 0 with different privilege levels in the IAOQ.

Notes: When this instruction returns to an instruction which executes at a lower privilege level, a lower-privilege transfer trap is not taken.

This instruction is the only instruction that can set the PSW Q-bit to 1.
**Reset System Mask (RSM)**

**Format:**

\[
\begin{array}{c|c|c|c|c}
0 & 0 & 0 & 73 & t \\
6 & 10 & 3 & 8 & 5 \\
\end{array}
\]

**Purpose:**
To selectively reset bits in the system mask to 0.

**Description:**
The current value of the system mask, PSW\{36,37,56..63\}, is saved in GR \(t\) and then the complement of the immediate value \(i\) is ANDed with the system mask.

**Operation:**

```c
if (priv != 0)
    privileged_operation_trap;
else {
    GR[t] ← 0;
    GR[t][36] ← PSW[W];
    GR[t][37] ← PSW[E];
    GR[t][56] ← PSW[O];
    GR[t][57] ← PSW[G];
    GR[t][58] ← PSW[F];
    GR[t][59] ← PSW[R];
    GR[t][60] ← PSW[Q];
    GR[t][61] ← PSW[P];
    GR[t][62] ← PSW[D];
    GR[t][63] ← PSW[I];
    PSW[W]← PSW[W] & (~i{0});
    PSW[E] ← PSW[E] & (~i{1});
    PSW[O] ← PSW[O] & (~i{2});
    PSW[G] ← PSW[G] & (~i{3});
    PSW[F] ← PSW[F] & (~i{4});
    PSW[R] ← PSW[R] & (~i{5});
    PSW[Q] ← PSW[Q] & (~i{6});
    PSW[P] ← PSW[P] & (~i{7});
    PSW[D] ← PSW[D] & (~i{8});
    PSW[I] ← PSW[I] & (~i{9});
}
```

**Exceptions:**
Privileged operation trap

**Restrictions:**
This instruction may be executed only at the most privileged level.

**Notes:**
The state of the IPRs, IIA queues, and the IPSW is undefined when this instruction is used to set the Q-bit to 0, if it was not already 0.
Shift Left and Add

**Format:**

\[ \text{SHLADD,cmplt,cond } r1,sa,r2,t \]

<table>
<thead>
<tr>
<th>02</th>
<th>r2</th>
<th>r1</th>
<th>c</th>
<th>f</th>
<th>e1</th>
<th>1</th>
<th>0</th>
<th>sa</th>
<th>d</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:**
To provide a primitive operation for multiplication.

**Description:**
GR \( r1 \) is shifted left \( sa \) bit positions and added to GR \( r2 \). If no trap occurs, the result is placed in GR \( t \). The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the add operation. The completer, \( \text{cmplt} \), encoded in the \( e1 \) field, specifies whether the carry/borrow bits in the PSW are updated and whether a trap is taken on signed overflow as shown in the table below. The shift amount is either 1, 2, or 3, and is encoded in the \( sa \) field of the instruction.

The following instruction is nullified if the values added satisfy the specified condition, \( \text{cond} \). The condition is encoded in the \( c, d, \) and \( f \) fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a 32-bit signed overflow (\( d=0 \)) or a 64-bit signed overflow (\( d=1 \)). For addition with carry in, the \( d \) field encodes whether the word carry (PSW C/B\{8\}, \( d=0 \)) or the doubleword carry (PSW C/B\{0\}, \( d=1 \)) is used.

For this instruction, signed overflow condition means that either the bit(s) shifted out differ from the leftmost bit following the shift or an ordinary signed overflow occurred during the addition. Unsigned overflow means that one or more of the bit(s) shifted out are 1 or an ordinary unsigned overflow occurred during the addition. For 32-bit overflows, it is the bits shifted out of the lower word that are checked. The conditions take on special interpretations since the shift operation participates in overflow determination.

The \( e1 \) field encodes whether the carry/borrow bits in the PSW are updated and whether a trap is taken on overflow (\( e1=1 \): carries updated, no trap, \( e1=2 \): carries not updated, no trap, \( e1=3 \): carries updated, trap on overflow.)

**Conditions:**
The condition is any of the 32-bit add conditions shown in Table D-6 on page D-5 or any of the 64-bit add conditions shown in Table D-7 on page D-6. When a condition completer is not specified, the "never" condition is used. The boolean variable "\( \text{cond\_satisfied} \)" in the operation section is set when the values added satisfy the specified condition.
Operation: \[\text{res} \leftarrow \text{lshift}(	ext{GR}[r1], sa) + \text{GR}[r2];\]
if (cmplt == TSV && overflow)
  overflow_trap;
else {
  \[\text{GR}[t] \leftarrow \text{res};\]
  if (cmplt != 'L')
    PSW[C/B] \leftarrow \text{carry_borrows};
    if (cond_satisfied) PSW[N] \leftarrow 1;
}

Exceptions: Overflow trap

Notes: When the ,L completer is specified, no trapping on overflow is available.
**Shift Right Pair Doubleword**

**SHRPD**

<table>
<thead>
<tr>
<th>Format:</th>
<th>SHRPD, cond r1, r2, sa, t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(11)</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td>(14)</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

**Purpose:** To shift a pair of registers by a fixed or variable amount and conditionally nullify the following instruction.

**Description:** The rightmost 63 bits of GR \( r1 \) are concatenated with the 64 bits of GR \( r2 \) and shifted right the number of bits given by the shift amount, \( sa \). The rightmost 64 bits of the result are placed in GR \( t \).

The shift amount, \( sa \), can either be a constant (specifying a fixed shift), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable shift.) Format 11 is used for variable shifts; Format 14 is used for fixed shifts. For fixed shifts, the shift amount \( sa \) in the assembly language format is represented by \( \text{cat}(c_p,c_{pos}) \) in the machine instruction, whose value is \( 63 - sa \).

The following instruction is nullified if the result of the operation satisfies the specified condition, \( cond \). The condition is encoded in the \( c \) field of the instruction.

**Conditions:** The condition is any of the 64-bit extract/deposit conditions shown in Table D-14 on page D-9. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

**Operation:**

if (fixed_shift) /* (Format 14) */

\[ \text{shamt} \leftarrow sa; \]

else /* (Format 11) */

\[ \text{shamt} \leftarrow \text{CR}[11]; \]

\[ \text{GR}[t] \leftarrow \text{rshift(\text{cat}(\text{GR}[r1][1..63],\text{GR}[r2]),\text{shamt})[63..126]}; \]

if (cond_satisfied) \( \text{PSW}[N] \leftarrow 1; \)

**Exceptions:** None.
Programming Note

A logical right shift of GR \( r \) by a variable amount contained in GR \( p \) leaving the result in GR \( t \) may be done by the following sequence:

\[
\text{MTSAR} \quad p \\
\text{SHRPD} \quad 0,r,sar,t
\]

An arithmetic right shift can be done with an extract instruction. See EXTRACT DOUBLEWORD for an example.

If \( r1 \) and \( r2 \) name the same register, its contents are rotated and placed in GR \( t \). For example, the following rotates the contents of \( ra \) right by 8 bits:

\[
\text{SHRPD} \quad ra,ra,8,ra
\]
Shift Right Pair Word

**Format:**

```
SHRPW,cond  r1,r2,sa,t
```

**(11)**

```
gr2  r1  c  0  0  0  0  t
  6  5  5  3  2  1  1  4  5
```

**(14)**

```
gr2  r1  c  1  0  cpos  t
  6  5  5  3  1  1  1  5  5
```

**Purpose:** To shift the rightmost 32 bits of a pair of registers by a fixed or variable amount and conditionally nullify the following instruction.

**Description:** The rightmost 31 bits of GR\( r1 \) are concatenated with the rightmost 32 bits of GR\( r2 \) and shifted right the number of bits given by the shift amount, \( sa \). The rightmost 32 bits of the result are placed in GR\( t \). The leftmost 32 bits of GR\( t \) are undefined.

The shift amount, \( sa \), can either be a constant (specifying a fixed shift), or can be SAR, the Shift Amount Register (CR 11) (specifying a variable shift.) Format 11 is used for variable shifts; Format 14 is used for fixed shifts. For variable shifts, the leftmost bit of the SAR is ignored, so the shift amount is between 0 and 31. For fixed shifts, the shift amount \( sa \) in the assembly language format is represented by \( cpos \) in the machine instruction, whose value is \( 31 - sa \).

The following instruction is nullified if the result of the operation satisfies the specified condition, \( cond \). The condition is encoded in the \( c \) field of the instruction.

**Conditions:** The condition is any of the 32-bit extract/deposit conditions shown in Table D-13 on page D-9. When a condition completer is not specified, the “never” condition is used. The boolean variable "cond_satisfied" in the operation section is set when the result of the operation satisfies the specified condition.

**Operation:**

```c
if (fixed_shift) /* (Format 14) */
   shamt ← sa;
else /* (Format 11) */
   shamt ← CR[11][1..5];
GR[t]{32..63} ← rshift(cat(GR[r1]{33..63},GR[r2]{32..63}),shamt){31..62};
GR[t]{0..31} ← undefined;
if (cond_satisfied) PSW[N] ← 1;
```

**Exceptions:** None.
Programming Note
A logical right shift of GR \( r \) by a variable amount contained in GR \( p \) leaving the result in GR \( t \) may be done by the following sequence:

\[
\begin{align*}
\text{MTSAR} & \quad p \\
\text{SHRPW} & \quad 0, r, \text{sar}, t
\end{align*}
\]

An arithmetic right shift can be done with an extract instruction. See EXTRACT WORD for an example.

If \( r1 \) and \( r2 \) name the same register, its contents are rotated and placed in GR \( t \). For example, the following rotates the contents of \( ra \) right by 8 bits:

\[
\text{SHRPW} \quad ra, ra, 8, ra
\]
## Special Operation Zero

### SPOP0

**Format:** SPOP0,sfu,sop,n

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td></td>
<td>sop1</td>
<td>0</td>
<td>sfu</td>
<td>n</td>
</tr>
</tbody>
</table>

(34)

**Purpose:** To invoke a special function unit operation.

**Description:** The SFU identified by `sfu` is directed to perform the operation specified by the information supplied to it. If nullification is specified, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

The `sop` field in the assembly language format is the concatenation of the `sop1` and `sop2` fields in the machine instruction, `sop = cat(sop1,sop2)`.

**Operation:**

sfu_operation0(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31});

if (n && sfu_condition0(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31}))

PSW[N] ← 1;

**Exceptions:**

- Assist emulation trap
- Assist exception trap
**Special Operation One**

**SPOP1**

**Format:**

SPOP1,sfu,sop,n t

<table>
<thead>
<tr>
<th>04</th>
<th>sop</th>
<th>1</th>
<th>sfu</th>
<th>n</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To copy a special function unit register or a result to a general register.

**Description:** A single word is sent from the SFU identified by sfu to GR t. The SFU uses its internal state and the instruction fields supplied to it to compute or select the result. If nullification is specified, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

**Operation:**

GR[t] ← sfu_operation1(cat(sop,sfu,n),IAOQ_Front[30..31]);
if (n && sfu_condition1(cat(sop,sfu,n),IAOQ_Front[30..31]))
    PSW[N] ← 1;

**Exceptions:**

Assist emulation trap
Assistant exception trap

**Notes:**

The SPECIAL OPERATION ONE instruction is used to implement the IDENTIFY SFU pseudo-operation. This operation returns a 32-bit identification number from the special function unit sfu to general register t. The value returned is implementation dependent and is useful for configuration, diagnostics, and error recovery. The state of the SFU is undefined after this instruction.

Each implementation must choose an identification number that identifies the version of the SFU. The values all zeros and all ones are reserved. The assist emulation trap handler returns zero when executing this instruction. An assist exception trap is not allowed and this instruction must be implemented by all SFUs. The IDENTIFY SFU pseudo-operation is coded as: SPOP1,sfu,0 t
Special Operation Two

**SPOP2**

**Format:** \( \text{SPOP2}, \text{sfu}, \text{sop}, n \ r \)

\[
\begin{array}{cccc|ccc|c}
  & 04 & r & 2 & \text{sfu} & n & \text{sop2} & \\
\hline
(36) & 6 & 5 & 10 & 2 & 3 & 1 & 5 \\
\end{array}
\]

**Purpose:** To perform a parameterized special function unit operation.

**Description:** GR \( r \) is passed to the SFU identified by \( \text{sfu} \). The SFU uses its internal state, the contents of the register, and the instruction fields supplied to it to compute a result. If nullification is specified, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

The \( \text{sop} \) field in the assembly language format is the concatenation of the \( \text{sop1} \) and \( \text{sop2} \) fields in the machine instruction, \( \text{sop} = \text{cat}(\text{sop1}, \text{sop2}) \).

**Operation:**

\[
\begin{align*}
\text{sfu\_operation2(} & \text{cat(sop1, sfu, n, sop2), IAOQ\_Front\{30..31\}, GR[r]}); \\
\text{if (} & n \text{ && sfu\_condition2(} \text{cat(sop1, sfu, n, sop2), IAOQ\_Front\{30..31\}, GR[r]})) \\
& \text{PSW}[N] \leftarrow 1;
\end{align*}
\]

**Exceptions:** Assist emulation trap

Assist exception trap
### Special Operation Three

**SPOP3**

**Format:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>r2</td>
<td>r1</td>
<td>sop1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>sfu</td>
<td>n</td>
<td>sop2</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:**

To perform a parameterized special function unit operation.

**Description:**

Gr r1 and GR r2 are passed to the SFU identified by sfu. The SFU uses its internal state, the contents of the two registers, and the instruction fields supplied to it to compute a result. If nullification is specified, the SFU also computes a 1-bit condition that causes the following instruction to be nullified if the condition is satisfied.

The sop field in the assembly language format is the concatenation of the sop1 and sop2 fields in the machine instruction, sop = cat(sop1,sop2).

**Operation:**

```c
sfu_operation3(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r1],GR[r2]);
if (n && sfu_condition3(cat(sop1,sfu,n,sop2),IAOQ_Front{30..31},GR[r1],GR[r2]))
    PSW[N] ← 1;
```

**Exceptions:**

Assist emulation trap

Assist exception trap
Set System Mask

<table>
<thead>
<tr>
<th>Format:</th>
<th>SSM  i,t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(33)</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6B</td>
</tr>
<tr>
<td></td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To selectively set bits in the system mask to 1.

Description: The current value of the system mask, PSW\{36,37,56..63\}, is saved in GR \(t\) and then the immediate value \(i\) is ORed with the system mask. Setting the PSW Q-bit, PSW\{60\}, to 1 with this instruction, if it was not already 1, is an undefined operation.

Operation:

if (priv != 0)
    privileged_operation_trap;
else {
    if ((PSW[Q] == 0) && (i{6}))
        undefined;
    else {
        GR[t] ← 0;
        GR[t]{36} ← PSW[W];
        GR[t]{37} ← PSW[E];
        GR[t]{56} ← PSW[O];
        GR[t]{57} ← PSW[G];
        GR[t]{58} ← PSW[F];
        GR[t]{59} ← PSW[R];
        GR[t]{60} ← PSW[Q];
        GR[t]{61} ← PSW[P];
        GR[t]{62} ← PSW[D];
        GR[t]{63} ← PSW[I];
        PSW[W] ← PSW[W] | i{0};
        PSW[E] ← PSW[E] | i{1};
        PSW[O] ← PSW[O] | i{2};
        PSW[G] ← PSW[G] | i{3};
        PSW[F] ← PSW[F] | i{4};
        PSW[R] ← PSW[R] | i{5};
        PSW[P] ← PSW[P] | i{7};
        PSW[D] ← PSW[D] | i{8};
        PSW[I] ← PSW[I] | i{9};
    }
}

Exceptions: Privileged operation trap

Restrictions: This instruction may be executed only at the most privileged level.
Store Byte

Format: \( \text{STB,cmplt,cc} \ r,d(s,b) \)

<table>
<thead>
<tr>
<th>18</th>
<th>b</th>
<th>r</th>
<th>s</th>
<th>im14</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>03</th>
<th>b</th>
<th>r</th>
<th>s</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>8</th>
<th>m</th>
<th>im5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To store a byte from a general register.

Description: The rightmost byte in GR \( r \) is stored in the aligned byte at the effective address. The offset is formed as the sum of a base register, \( b \), and a displacement \( d \). The displacement can be either long (Format 1) or short (Format 6). 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 displacement. The completer also specifies base register modification 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 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.

If base register modification is specified and \( b = r \), the value stored at the effective address is the byte from the source register before modification.
Operation: if (d > 15 || d < -16) {
    dx ← sign_ext(assemble_16(s,im14),16); /* long displacement */
    cc ← NO_HINT; /* (Format 1) */
} else /* short displacement */
    dx ← low_sign_ext(im5,5);
    cc ← NO_HINT; /* (Format 6) */
space ← space_select(s,GR[b],format);
if (cmplt == O)
    enforce_ordered_store;
switch (cmplt) {
    case MB: offset ← GR[b] + dx;
        mem_store(space,offset,0,7,cc,GR[r]{56..63});
        GR[b] ← GR[b] + dx;
        break;
    case MA: offset ← GR[b];
        mem_store(space,offset,0,7,cc,GR[r]{56..63});
        GR[b] ← GR[b] + dx;
        break;
    default: offset ← GR[b] + dx;
        mem_store(space,offset,0,7,cc,GR[r]{56..63});
        break;
}

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

Restrictions: If the completer O is specified, the displacement must be 0.
Store Bytes

Format: \( \text{STBY}, \text{cmplt}, \text{cc} \ r, d(s, b) \)

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>r</th>
<th>s</th>
<th>a</th>
<th>l</th>
<th>cc</th>
<th>C</th>
<th>m</th>
<th>im5</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To implement the beginning, middle, and ending cases for fast byte moves with unaligned sources and destinations.

Description: If the PSW E-bit is 0 and begin (modifier ",B" corresponding to \( a = 0 \)) is specified, the rightmost bytes of GR \( r \) are stored in memory starting at the byte whose address is given by the effective address. The number of bytes stored is sufficient to fill out the word containing the byte addressed by the effective address.

If the PSW E-bit is 0 and end (modifier ",E" corresponding to \( a = 1 \)) is specified, the leftmost bytes of the rightmost word of GR \( r \) are stored in memory starting at the leftmost byte in the word specified by the effective address, and continuing until (but not including) the byte specified by the effective address. When the effective address specifies the leftmost byte in a word, nothing is stored, but protection is checked and the cache line is marked as dirty.

If the PSW E-bit is 1 and begin (modifier ",B" corresponding to \( a = 0 \)) is specified, the leftmost bytes of the rightmost word of GR \( r \) are stored in memory starting at the byte whose address is given by the effective address. The number of bytes stored is sufficient to fill out the word containing the byte addressed by the effective address.

If the PSW E-bit is 1 and end (modifier ",E" corresponding to \( a = 1 \)) is specified, the rightmost bytes of GR \( r \) are stored in memory starting at the leftmost byte in the word specified by the effective address, and continuing until (but not including) the byte specified by the effective address. When the effective address specifies the leftmost byte in a word, nothing is stored, but protection is checked and the cache line is marked as dirty.

If base register modification is specified through completer ",M", GR \( b \) is updated and then truncated to a word address. (Table 7-7 defines the assembly language completer mnemonics.) If base register modification is specified and \( b = r \), the value stored at the effective address is the bytes from the source register before modification.

Table 7-7. Store Bytes Instruction Completers

<table>
<thead>
<tr>
<th>cmplt</th>
<th>Description</th>
<th>a</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt; or B</td>
<td>Beginning case, don’t modify base register</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B,M</td>
<td>Beginning case, Modify base register</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>Ending case, don’t modify base register</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E,M</td>
<td>Ending case, Modify base register</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The completer, \( cc \), specifies the cache control hint (see Table 6-8 on page 6-10.) If the first byte of the addressed cache line is not written to, the processor must perform the store as if...
the cache control hint had not been specified.

Operation:

```
space ← space_select(s,GR[b],format);
dx ← low_sign_ext(im5,5);
if (cmplt == B,M) */a=0, m=1*/
    offset ← GR[b];
else
    offset ← GR[b] + dx;
pos ← 8*(offset & 0x3);
offset ← offset & ~0x3; /* word aligned */
switch (cmplt) {
    case B:
        /*a=0, m=0*/
        if (PSW[E] == 0)
            mem_store(space,offset,pos,31,cc,GR[r]{pos+32..63});
        else
            mem_store(space,offset,pos,31,cc,GR[r]{32..63-pos});
        break;
    case E:
        /*a=1, m=0*/
        if (PSW[E] == 0)
            mem_store(space,offset,0,pos-1,cc,GR[r]{32..pos+31});
        else
            mem_store(space,offset,0,pos-1,cc,GR[r]{64-pos..63});
        break;
    case B,M:
        /*a=0, m=1*/
        if (PSW[E] == 0)
            mem_store(space,offset,pos,31,cc,GR[r]{pos+32..63});
        else
            mem_store(space,offset,pos,31,cc,GR[r]{32..63-pos});
        GR[b] ← (GR[b] + dx) & ~0x3;
        break;
    case E,M:
        /*a=1, m=1*/
        if (PSW[E] == 0)
            mem_store(space,offset,0,pos-1,cc,GR[r]{32..pos+31});
        else
            mem_store(space,offset,0,pos-1,cc,GR[r]{64-pos..63});
        GR[b] ← (GR[b] + dx) & ~0x3;
        break;
}
```

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

Notes: All bits of the original virtual offset are saved, unmasked, to IOR (CR21) if this instruction traps.
For this instruction, the low 2 bits of the virtual offset are masked to 0 when comparing
against the contents of the data breakpoint address offset registers.

---

Programming Note

The STBY instruction with the ".E" completer and the effective address specifying the leftmost byte of the word may be used to implement a memory scrubbing operation. This is possible because the line is marked *dirty* but the contents are not modified.
### Store Doubleword

**Format:** \( \text{STD,cmplt,cc} \ r, d(s, b) \)

<table>
<thead>
<tr>
<th>03</th>
<th>b</th>
<th>r</th>
<th>s</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>B</th>
<th>m</th>
<th>im5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To store a doubleword from a general register.

**Description:** GR \( r \) is stored in the aligned doubleword at the effective address. The offset is formed as the sum of a base register, \( b \), and a displacement \( d \). The displacement can be either long (Format 3) or short (Format 6.) 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 displacement. The completer also specifies base register modification 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.

If base register modification is specified and \( b = r \), the value stored at the effective address is the word from the source register before modification.
Operation: if (d > 15 || d < -16) {
    dx ← sign_ext(assemble_16a(s,cat(im10a,0),i),16);
    cc ← NO_HINT;
} else
    dx ← low_sign_ext(im5,5);
space ← space_select(s,GR[b],format);
if (cmplt == 0)
    enforce_ordered_store;
switch (cmplt) {
    case MB:  offset ← GR[b] + dx;
               mem_store(space,offset,0,63,cc,GR[r]);
               GR[b] ← GR[b] + dx;
               break;
    case MA:  offset ← GR[b];
               mem_store(space,offset,0,63,cc,GR[r]);
               GR[b] ← GR[b] + dx;
               break;
    default: offset ← GR[b] + dx;
             mem_store(space,offset,0,63,cc,GR[r]);
             break;
}

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

Restrictions: For 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.
Store Doubleword Absolute

**Format:**  
STDA, cmplt, cc  
$r, d(b)$

(6)

<table>
<thead>
<tr>
<th></th>
<th>03</th>
<th>b</th>
<th>r</th>
<th>0</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>F</th>
<th>m</th>
<th>im5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**Purpose:** To store a doubleword from a general register to an absolute address.

**Description:** The register $r$ is stored in the aligned doubleword at the effective absolute address. The offset is formed as the sum of a base register, $b$, and a displacement $d$. 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 displacement. The completer also specifies base register modification and ordering constraints (see Table H-1 on page H-4 for the assembly language completer mnemonics.) The completer, $cc$, specifies the cache control hint (see Table 6-8 on page 6-10.)

If base register modification is specified and $b = r$, the value stored at the effective address is the doubleword from the source register before modification. Protection is not checked when this instruction is executed. This operation is only defined if the address is aligned on an 8-byte boundary.

**Operation:**

```plaintext
if (priv != 0)
    privileged_operation_trap;
else {
    dx ← low_sign_ext(im5,5);
    if (cmplt == O)
        enforce_ordered_store;
    switch (cmplt) {
        case MB: offset ← GR[b] + dx;
            phys_mem_store(offset,0,63,cc,GR[r]);
            GR[b] ← GR[b] + dx;
            break;
        case MA: offset ← GR[b];
            phys_mem_store(offset,0,63,cc,GR[r]);
            GR[b] ← GR[b] + dx;
            break;
        default: offset ← GR[b] + dx;
            phys_mem_store(offset,0,63,cc,GR[r]);
            break;
    }
}
```

**Exceptions:** Privileged operation trap

**Restrictions:** This instruction may be executed only at the most privileged level. If the completer $O$ is specified, the displacement must be 0.
Store Doubleword Bytes

**STDBY**

*Format: STDBY,cmplt,cc r,d(s,b)*

*Purpose: To implement the beginning, middle, and ending cases for fast byte moves with unaligned sources and destinations.*

*Description: If the PSW E-bit is 0 and begin (modifier ",B" corresponding to \(a = 0\)) is specified, the rightmost bytes of GR \(r\) are stored in memory starting at the byte whose address is given by the effective address. The number of bytes stored is sufficient to fill out the doubleword containing the byte addressed by the effective address.

If the PSW E-bit is 0 and end (modifier ",E" corresponding to \(a = 1\)) is specified, the leftmost bytes of GR \(r\) are stored in memory starting at the leftmost byte in the doubleword specified by the effective address, and continuing until (but not including) the byte specified by the effective address. When the effective address specifies the leftmost byte in a doubleword, nothing is stored, but protection is checked and the cache line is marked as *dirty*.

If the PSW E-bit is 1 and begin (modifier ",B" corresponding to \(a = 0\)) is specified, the leftmost bytes of GR \(r\) are stored in memory starting at the byte whose address is given by the effective address. The number of bytes stored is sufficient to fill out the doubleword containing the byte addressed by the effective address.

If the PSW E-bit is 1 and end (modifier ",E" corresponding to \(a = 1\)) is specified, the rightmost bytes of GR \(r\) are stored in memory starting at the leftmost byte in the doubleword specified by the effective address, and continuing until (but not including) the byte specified by the effective address. When the effective address specifies the leftmost byte in a doubleword, nothing is stored, but protection is checked and the cache line is marked as *dirty*.

If base register modification is specified through completer ",M", GR \(b\) is updated and then truncated to a doubleword address (see Table 7-7 on page 7-133 for the assembly language completer mnemonics.) If base register modification is specified and \(b = r\), the value stored at the effective address is the bytes from the source register before modification.

The completer, cc, specifies the cache control hint (see Table 6-8 on page 6-10.) If the first byte of the addressed cache line is not written to, the processor must perform the store as if the cache control hint had not been specified.
Operation: space ← space_select(s, GR[b], format);
    dx ← low_sign_ext(im5, 5);
    if (cmplt == B,M) /* a=0, m=1 */
      offset ← GR[b];
    else
      offset ← GR[b] + dx;
    pos ← 8*(offset & 0x7);
    offset ← offset & ~0x7; /* doubleword aligned */
    switch (cmplt) {
      case B: /* a=0, m=0 */
        if (PSW[E] == 0)
          mem_store(space, offset, pos, 63, cc, GR[r]{pos..63});
        else
          mem_store(space, offset, pos, 63, cc, GR[r]{0..63-pos});
        break;
      case E: /* a=1, m=0 */
        if (PSW[E] == 0)
          mem_store(space, offset, 0, pos-1, cc, GR[r]{0..pos-1});
        else
          mem_store(space, offset, 0, pos-1, cc, GR[r]{64-pos..63});
        break;
      case B,M: /* a=0, m=1 */
        if (PSW[E] == 0)
          mem_store(space, offset, pos, 63, cc, GR[r]{pos..63});
        else
          mem_store(space, offset, pos, 63, cc, GR[r]{0..63-pos});
        GR[b] ← (GR[b] + dx) & ~0x7;
        break;
      case E,M: /* a=1, m=1 */
        if (PSW[E] == 0)
          mem_store(space, offset, 0, pos-1, cc, GR[r]{0..pos-1});
        else
          mem_store(space, offset, 0, pos-1, cc, GR[r]{64-pos..63});
        GR[b] ← (GR[b] + dx) & ~0x7;
        break;
    }

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

Notes: All bits of the original virtual offset are saved, unmasked, to IOR (CR21) if this instruction traps.
For this instruction, the low 3 bits of the virtual offset are masked to 0 when comparing against the contents of the data breakpoint address offset registers.
Programming Note

The STDBY instruction with the ".E" completer and the effective address specifying the leftmost byte of the doubleword may be used to implement a memory scrubbing operation. This is possible because the line is marked dirty but the contents are not modified.
Store Halfword

Format: \texttt{STH,cmplt,cc \ r,d(\$b)}

(1)

\begin{tabular}{ccccccc}
19 & b & r & s & im14 \\
6 & 5 & 5 & 2 & 14
\end{tabular}

(6)

\begin{tabular}{ccccccccccc}
03 & b & r & s & a & 1 & cc & 9 & m & im5 \\
6 & 5 & 5 & 2 & 1 & 1 & 2 & 4 & 1 & 5
\end{tabular}

Purpose: To store a halfword from a general register.

Description: The rightmost halfword in GR \( r \) is stored in the aligned halfword at the effective address. The offset is formed as the sum of a base register, \( b \), and a displacement \( d \). The displacement can be either long (Format 1) or short (Format 6). The displacement is encoded into the immediate field. Optional base modification can also be performed.

The completer, \textit{cmplt}, determines whether the offset is the base register, or the base register plus the displacement. The completer also specifies base register modification 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, \textit{cc}, specifies the cache control hint (see Table 6-8 on page 6-10.)

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.

If base register modification is specified and \( b = r \), the value stored at the effective address is the halfword from the source register before modification.
Operation:  if (d > 15 || d < -16) {
   dx ← sign_ext(assemble_16(s,im14),16); /* (Format 1) */
   cc ← NO_HINT;
} else /* short displacement */
   dx ← low_sign_ext(im5,5);
   cc ← NO_HINT;
else /* long displacement */
   cc ← NO_HINT;
   /* (Format 6) */
   space ← space_select(s,GR[b],format);
   if (cmplt == O)
      enforce_ordered_store;
   switch (cmplt) {
   case MB: offset ← GR[b] + dx;
      mem_store(space,offset,0,15,cc,GR[r]{48..63});
      GR[b] ← GR[b] + dx;
      break;
   case MA: offset ← GR[b];
      mem_store(space,offset,0,15,cc,GR[r]{48..63});
      GR[b] ← GR[b] + dx;
      break;
   default: offset ← GR[b] + dx;
      mem_store(space,offset,0,15,cc,GR[r]{48..63});
      break;
   }

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

Restrictions:  If the completer O is specified, the displacement must be 0.
**Store Word**

<table>
<thead>
<tr>
<th>Format:</th>
<th>STW,cmplt,cc  r, d(s, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>1A/1B  b  r  s  im14</td>
</tr>
<tr>
<td></td>
<td>6      5  5  2  14</td>
</tr>
<tr>
<td>(2)</td>
<td>1F  b  r  s  im11a  2  i</td>
</tr>
<tr>
<td></td>
<td>6      5  2  11  2  1</td>
</tr>
<tr>
<td>(6)</td>
<td>03  b  r  s  a 1  cc  A  m  im5</td>
</tr>
<tr>
<td></td>
<td>6      5  2  2  1  2  4  1  5</td>
</tr>
</tbody>
</table>

**Purpose:** To store a word from a general register.

**Description:** The rightmost word in GR $r$ is stored in the aligned word at the effective address. The offset is formed as the sum of a base register, $b$, and a displacement $d$. The displacement can be either long (Formats 1 and 2) or short (Format 6). 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 displacement. The completer also specifies base register modification 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 pre-decrement or post-increment, Format 1 (opcode 1B) is used. For long displacements with post-decrement or pre-increment, Format 2 is used. For long displacements with no base modification, Format 1 (opcode 1A) 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.

If base register modification is specified and $b = r$, the value stored at the effective address is the word from the source register before modification.
Operation: if (d > 15 || d < -16) { /* long displacement */
    if (((cmplt==MB && d>=0) || (cmplt==MA && d<0))
        dx ← sign_ext(assemble_16a(s,im11a,i),16); /* (Format 2) */
    else
        dx ← sign_ext(assemble_16(s,im14),16); /* (Format 1) */
        cc ← NO_HINT;
    } else /* short displacement */
        dx ← low_sign_ext(im5,5); /* (Format 6) */
    space ← space_select(s,GR[b],format);
    if (cmplt == O)
        enforce_ordered_store;
    switch (cmplt) {
        case MB: offset ← GR[b] + dx;
                mem_store(space,offset,0,31,cc,GR[r]{32..63});
                GR[b] ← GR[b] + dx;
                break;
        case MA: offset ← GR[b];
                mem_store(space,offset,0,31,cc,GR[r]{32..63});
                GR[b] ← GR[b] + dx;
                break;
        default: offset ← GR[b] + dx;
                mem_store(space,offset,0,31,cc,GR[r]{32..63});
                break;
    }

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

Restrictions: For post-decrement and pre-increment with long displacements (Format 2), only
              displacements which are multiples of four may be used.

              If the completer O is specified, the displacement must be 0.
Store Word Absolute

Format: STWA,cmplt,cc r,d(b)

<table>
<thead>
<tr>
<th>03</th>
<th>b</th>
<th>r</th>
<th>0</th>
<th>a</th>
<th>1</th>
<th>cc</th>
<th>E</th>
<th>m</th>
<th>im5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To store a word from a general register to an absolute address.

Description: The rightmost word in GR \( r \) is stored in the aligned word at the effective absolute address. The offset is formed as the sum of a base register, \( b \), and a displacement \( d \). 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 displacement. The completer also specifies base register modification and ordering constraints (see Table H-1 on page H-4 for the assembly language completer mnemonics.) The completer, \( cc \), specifies the cache control hint (see Table 6-8 on page 6-10.)

If base register modification is specified and \( b = r \), the value stored at the effective address is the word from the source register before modification. Protection is not checked when this instruction is executed. This operation is only defined if the address is aligned on a 4-byte boundary.

Operation:

```c
if (priv != 0)
    privileged_operation_trap;
else {
    dx ← low_sign_ext(im5,5);
    if (cmplt == 0)
        enforce_ordered_store;
    switch (cmplt) {
        case MB: offset ← GR[b] + dx;
            phys_mem_store(offset,0,31,cc,GR[r]{32..63});
            GR[b] ← GR[b] + dx;
            break;
        case MA: offset ← GR[b];
            phys_mem_store(offset,0,31,cc,GR[r]{32..63});
            GR[b] ← GR[b] + dx;
            break;
        default: offset ← GR[b] + dx;
            phys_mem_store(offset,0,31,cc,GR[r]{32..63});
            break;
    }
}
```

Exceptions: Privileged operation trap
Restrictions: This instruction may be executed only at the most privileged level. If the completer \( O \) is specified, the displacement must be 0.
Subtract

Format: \texttt{SUB,cmplt,borrow,trapc,cond r1,r2,t}

<table>
<thead>
<tr>
<th>Completer</th>
<th>Description</th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>Subtract</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TC</td>
<td>Subtract and trap on condition</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>B or DB</td>
<td>Subtract with borrow/doubleword borrow</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>TSV</td>
<td>Subtract and trap on signed overflow</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TSV,TC</td>
<td>Subtract and trap on signed overflow or condition</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>B,TSV or DB,TSV</td>
<td>Subtract with borrow/doubleword borrow and trap on signed overflow</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Conditions: The condition is any of the 32-bit compare or subtract conditions shown in Table D-3 on page D-4 or any of the 64-bit compare or subtract conditions shown in Table D-4 on page D-4. When a condition completer is not specified, the “never” condition is used. The boolean variable “cond_satisfied” in the operation section is set when the values subtracted satisfy the specified condition.

Purpose: To do 64-bit integer subtraction, and conditionally nullify the following instruction.

Description: GR r2 is subtracted from GR r1. If no trap occurs, the result is placed in GR t and the carry/borrow bits in the PSW are updated. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the subtract operation. The completer, \texttt{cmplt}, encoded in the \texttt{e1} field, specifies whether a trap is taken on signed overflow. The completer, \texttt{borrow}, encoded in the \texttt{e2} field, specifies whether the subtraction is done with borrow. The completer, \texttt{trapc}, encoded in the \texttt{e3} field, specifies whether a trap is taken if the values subtracted satisfy the condition specified.

The following instruction is nullified if the values subtracted satisfy the specified condition, \texttt{cond}. The condition is encoded in the \texttt{c}, \texttt{d}, and \texttt{f} fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a 32-bit signed overflow (\texttt{d}=0) or a 64-bit signed overflow (\texttt{d}=1.) For subtraction with borrow, the \texttt{d} field encodes whether the word borrow (PSW C/B\{8\}, \texttt{d}=0), or the doubleword borrow (PSW C/B\{0\}, \texttt{d}=1) is used.

The \texttt{e1} field encodes whether the a trap is taken on overflow (\texttt{e1}=1: no trap, \texttt{e1}=3: trap on overflow.) The \texttt{e2} field encodes whether subtraction with borrow in is performed (\texttt{e2}=0: no borrow, \texttt{e2}=1: subtraction performed with borrow.) The \texttt{e3} field encodes whether to trap if the values subtracted satisfy the specified condition (\texttt{e3}=0: no trap, \texttt{e3}=3: trap on condition.)
Operation: switch (borrow) {
    case B: res ← GR[r1] + ¬GR[r2] + PSW[C/B][8];
    break;
    case DB: res ← GR[r1] + ¬GR[r2] + PSW[C/B][0];
    break;
    default: res ← GR[r1] + ¬GR[r2] + 1;
    break;
}
if (cmplt == TSV && overflow)
    overflow_trap;
else if (trapc == TC && cond_satisfied)
    conditional_trap;
else {
    GR[t] ← res;
    PSW[C/B] ← carry_borrows;
    if (cond_satisfied) PSW[N] ← 1;
}

Exceptions: Overflow trap
Conditional trap

Notes: When the ,B completer is specified, only 32-bit conditions are available. When the ,DB completer is specified, only 64-bit conditions are available.
Subtract from Immediate

Format: \texttt{SUBI,cmplt,cond\ i,r,t}

<table>
<thead>
<tr>
<th></th>
<th>25</th>
<th>r</th>
<th>t</th>
<th>c</th>
<th>f</th>
<th>e1</th>
<th>im11</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Purpose: To subtract a register from an immediate value and conditionally nullify the following instruction.

Description: \( GR_r \) is subtracted from the sign-extended immediate value \( i \). If no trap occurs, the result is placed in \( GR_t \) and the carry/borrow bits in the PSW are updated. The immediate value is encoded into the \( im11 \) field. The variable "carry_borrows" in the operation section captures the 4-bit carries resulting from the subtract operation.

The completer, \( cmplt \), encoded in the \( e1 \) field, specifies whether a trap is taken on a 32-bit signed overflow (\( e1=0 \): no trap, \( e1=1 \): trap on 32-bit signed overflow, \( cmplt==TSV \)).

The following instruction is nullified if the values subtracted satisfy the specified condition, \( cond \). The condition is encoded in the \( c \) and \( f \) fields of the instruction. The boolean variable "overflow" in the operation section is set if the operation results in a 32-bit signed overflow.

Conditions: The condition is any of the 32-bit compare or subtract conditions shown in Table D-3 on page D-4. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values subtracted satisfy the specified condition.

Operation: 
\[
\text{res} \leftarrow \text{low\_sign\_ext(im11,11)} + \sim\text{GR}[r] + 1; \\
\text{if (cmplt == TSV \&\& overflow)} \\
\qquad \text{overflow\_trap;}
\]
\[
\text{else} \{
\qquad \text{GR}[t] \leftarrow \text{res;}
\qquad \text{PSW}[C/B] \leftarrow \text{carry\_borrows;}
\qquad \text{if (cond\_satisfied)} \text{PSW}[N] \leftarrow 1;
\text{\} }
\]

Exceptions: Overflow trap

Programming Note

SUBTRACT FROM IMMEDIATE can be used to perform a logical NOT operation when coded as follows:

\[
\text{SUBI -1,r,t} \quad \text{/* GR}[t] \leftarrow \sim\text{GR}[r] \\
\text{all PSW}[C/B] \text{ are set to ones */}
\]
Synchronize Caches

SYNC

Format: SYNC

<table>
<thead>
<tr>
<th></th>
<th>rv</th>
<th>0</th>
<th>rv</th>
<th>0</th>
<th>20</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

Purpose: To enforce program order of instruction execution.

Description: Any load, store, semaphore, cache flush, or cache purge instructions that follow the SYNC instruction get executed only after all such instructions prior to the SYNC instruction have completed executing. On implementations which execute such instructions out of sequence, this instruction enforces program ordering.

Operation: Enforce program order of memory references

Exceptions: None

Notes: In systems in which all memory references are performed in order, this instruction executes as a null instruction.

Programming Note

The minimum spacing that is guaranteed to work for "self-modifying code" is shown in the code segment below. Since instruction prefetching is permitted, any data cache flushes must be separated from any instruction cache flushes by a SYNC. This will ensure that the "new" instruction will be written to memory prior to any attempts at prefetching it as an instruction.

```
LDIL l%newinstr,rnew
LDW r%newinstr(0,rnew),temp
LDIL l%instr,rinstr
STW temp,r%instr(0,rinstr)
FDC r%instr(0,rinstr)
SYNC
FIC r%instr(rinstr)
SYNC
(at least seven instructions)
instr . . .
```

This sequence assumes a uniprocessor system. In a multiprocessor system, software must ensure no processor is executing code which is in the process of being modified.
Synchronize DMA

SYNCDMA

Format: SYNCDMA

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>rv</td>
<td>1</td>
<td>rv</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Purpose: To enforce DMA completion order.

Description: On implementations which can signal DMA completion prior to achieving cache coherence, this instruction enforces ordering. All cache coherence actions which are outstanding as a consequence of prior DMA operations must be completed before the next memory access is performed.

Operation: Enforce DMA completion order

Exceptions: None

Notes: In systems in which all DMA operations are performed in order, this instruction executes as a null instruction.
### Unit Add Complement

**Format:**

<table>
<thead>
<tr>
<th></th>
<th>02</th>
<th>r2</th>
<th>r1</th>
<th>c</th>
<th>f</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>e1</th>
<th>d</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To individually compare corresponding sub-units of a doubleword for a greater-than or less-than-or-equal a relation.

**Description:**

GR \( r1 \) is added to the one’s complement of GR \( r2 \). If no trap occurs, the result is placed in GR \( t \). The following instruction is nullified if the values added satisfy the specified condition, \( cond \). The completer, \( trapc \), encoded in the \( e1 \) field, specifies whether a trap is taken if the condition, \( cond \), is satisfied by the values added (\( e1=2 \): no trap, no \( cmplt \), \( e1=3 \): trap on condition, \( cmplt==TC \)). The condition is encoded in the \( c \), \( d \), and \( f \) fields of the instruction.

**Conditions:** The condition \( cond \) is any of the 32-bit unit conditions shown in Table D-11 on page D-8 or any of the 64-bit conditions shown in Table D-12 on page D-8. When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values added satisfy the specified condition.

**Operation:**

\[
\text{res} \leftarrow \text{GR}[r1] + \sim\text{GR}[r2]; \\
\text{if } (\text{trapc} == \text{TC} \&\& \text{cond}_satisfied) \\
\hspace{1em} \text{conditional_trap}; \\
\text{else } \\
\hspace{1em} \text{GR}[t] \leftarrow \text{res}; \\
\hspace{1.5em} \text{if } (\text{cond}_satisfied) \text{PSW}[N] \leftarrow 1;
\]

**Exceptions:** Conditional trap

---

**Programming Note**

UNIT ADD COMPLEMENT can be used to perform a logical NOT operation when coded as follows:

UADDCM 0,r,t /* GR[t] ← ~GR[r] */

UNIT ADD COMPLEMENT with the TC (Trap on Condition) completer can be used to check decimal validity and to pre-bias decimal numbers. \( ra \) contains the number to be checked and \( rt \)
will contain the number plus the bias as result of the UADDCM operation.

NINES .equ X'99999999
LDIL l%NINES,nines
LDO r%NINES(nines),nines
UADDCM,TC,SDC ra,nines,rt
Unit XOR

Format: UXOR,cond r1,r2,t

Purpose: To individually compare corresponding sub-units of two doublewords for equality.

Description: GR $r1$ and GR $r2$ are XORed and the result is placed in GR $t$. This instruction generates unit conditions unlike XOR which generates logical conditions. The following instruction is nullified if the values XORed satisfy the specified condition, $\text{cond}$. The condition is encoded in the $c$, $d$, and $f$ fields of the instruction.

Conditions: The condition, $\text{cond}$, is any of the 32-bit unit conditions not involving carries shown in Table D-11 on page D-8 ("never", SBZ, SHZ, TR, NBZ, NHZ) or any of the 64-bit unit conditions not involving carries shown in Table D-12 on page D-8 (*, *SBZ, *SHZ, *SWZ, *TR, *NBZ, *NHZ, *NWZ.) When a condition completer is not specified, the "never" condition is used. The boolean variable "cond_satisfied" in the operation section is set when the values XORed satisfy the specified condition.

Operation: $\text{GR}[t] \leftarrow \text{xor}($GR$[r1], \text{GR}[r2])$;
if (cond_satisfied) $\text{PSW}[N] \leftarrow 1$;

Exceptions: None

\[ \begin{array}{cccccccccc}
 02 & | & r2 & | & r1 & | & c & | & f & | & 0 & | & 1 & | & 1 & | & 2 & | & d & | & t \\
 6 & | & 5 & | & 5 & | & 3 & | & 1 & | & 2 & | & 1 & | & 2 & | & 1 & | & 5
\]
Exclusive OR

**XOR,cond r1,r2,t**

<table>
<thead>
<tr>
<th></th>
<th>02</th>
<th>r2</th>
<th>r1</th>
<th>c</th>
<th>f</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>d</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>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Purpose:** To do a 64-bit, bitwise exclusive OR.

**Description:** GR $r_1$ and GR $r_2$ are XORed and the result is placed in GR $t$. The following instruction is nullified if the values XORed satisfy the specified condition, `cond`. The condition is encoded in the $c$, $d$, and $f$ fields of the instruction.

**Conditions:** The condition is any of the 32-bit logical conditions shown in Table D-9 on page D-7 or any of the 64-bit logical conditions shown in Table D-10 on page D-7. The boolean variable "cond_satisfied" in the operation section is set when the values XORed satisfy the specified condition.

**Operation:**

$$\text{GR}[t] \leftarrow \text{xor}($$

GR[$r_1$], GR[$r_2$]);

if (cond_satisfied) PSW[$N$] $\leftarrow$ 1;

**Exceptions:** None