Assembly Language Syntax

The notations given in this section are taken from Sun’s SPARC Assembler and are used to describe the suggested assembly language syntax for the instruction definitions explained on page 5.

Understanding the use of type fonts is crucial to understanding the assembly language syntax in the instruction definitions. Items in typewriter font are literals, to be entered exactly as they appear. Items in italic font are metasymbols that are to be replaced by numeric or symbolic values when actual assembly language code is written. For example, asi would be replaced by a number in the range of 0 to 255 (the value of the bits in the binary instruction), or by a symbol that has been bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, regrs2 is a reg (i.e., register name) whose binary value will end up in the rs2 field of the resulting instruction.
Register Names

reg

A reg is an integer unit register. It can have a value of:

- `%0` through `%31` all integer registers
- `%g0` through `%g7` global registers—same as `%0` through `%7`
- `%o0` through `%o7` out registers—same as `%8` through `%15`
- `%l0` through `%l7` local registers—same as `%16` through `%23`
- `%i0` through `%i7` in registers—same as `%24` through `%31`

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

- `reg_r1` — rs1 field
- `reg_r2` — rs2 field
- `reg_rd` — rd field

freg

A freg is a floating-point register. It can have a value from `%f0` through `%f31`. Subscripts further identify the placement of the operand in the binary instruction as one of the following:

- `freg_r1` — rs1 field
- `freg_r2` — rs2 field
- `freg_rd` — rd field

creg

A creg is a coprocessor register. It can have a value from `%c0` through `%c31`. Subscripts further identify the placement of the operand in the binary instruction as one of the following:

- `creg_r1` — rs1 field
- `creg_r2` — rs2 field
- `creg_rd` — rd field

Special Symbol Names

Certain special symbols need to be written exactly as they appear in the syntax table. These appear in typewriter font, and are preceded by a percent sign (%). The percent sign is part of the symbol name; it must appear as part of the literal value.

The symbol names are:

- `%psr` Processor State Register
- `%wim` Window Invalid Mask register
- `%tbr` Trap Base Register
Values

Some instructions use operands comprising values as follows:

- `simm13`—A signed immediate constant that fits in 13 bits
- `const22`—A constant that fits in 22 bits
- `asi`—An alternate address space identifier (0 to 255)

Label

A label is a sequence of characters comprised of alphabetic letters (a-z, A-Z (upper and lower case distinct)), underscore (_), dollar sign ($), period (.), and decimal digits (0-9), but which does not begin with a decimal digit.

Some instructions offer a choice of operands. These are grouped as follows:

- **regaddr:**
  - `regrs1`
  - `regrs1 + regrs2`

- **address:**
  - `regrs1`
  - `regrs1 + regrs2`
  - `regrs1 + simm13`
  - `regrs1 - simm13`
  - `simm13`
  - `simm13 + regrs1`

- **reg_or_imm:**
  - `regrs1`
  - `simm13`

Instruction Mnemonics

*Figure 1.* illustrates the mnemonics used to describe the SPARC instruction set. Note that some combinations possible in Figure 1. do not correspond to valid instructions (such as store signed or floating-point convert extended to extended). Refer to the instruction summary on PageBreak 7 for a list of valid SPARC instructions.
Figure 1. SPARC Instruction Mnemonic Summary

Data Transfer

```
<table>
<thead>
<tr>
<th>Load</th>
<th>STore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Single</td>
<td>Double</td>
</tr>
<tr>
<td>Byte</td>
<td>Halfword</td>
</tr>
<tr>
<td>Normal</td>
<td>Alternate</td>
</tr>
<tr>
<td>Word</td>
<td>Doubleword</td>
</tr>
<tr>
<td>Floating-point</td>
<td>Coprocessor</td>
</tr>
<tr>
<td>Register</td>
<td>Status</td>
</tr>
<tr>
<td>Register</td>
<td>Queue</td>
</tr>
</tbody>
</table>
```

atomic SWAP word  atomic Load-Store Unsigned Byte

Integer Operations

```
<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>Not</td>
<td>normal</td>
</tr>
<tr>
<td>set CC</td>
<td>set CC</td>
<td></td>
</tr>
</tbody>
</table>

Shift

```
<table>
<thead>
<tr>
<th>Left</th>
<th>Right</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tagged

```
<table>
<thead>
<tr>
<th>ADD</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>set CC</td>
<td>Trap of Verflow</td>
</tr>
</tbody>
</table>

MULtiply

```
<table>
<thead>
<tr>
<th>Step set CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETHI</td>
</tr>
<tr>
<td>SAVE</td>
</tr>
<tr>
<td>RESTORE</td>
</tr>
</tbody>
</table>

Floating-Point Operations

```
<table>
<thead>
<tr>
<th>Fp</th>
<th>convert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Single</td>
</tr>
<tr>
<td>Single</td>
<td>Double</td>
</tr>
<tr>
<td>eXtended</td>
<td>eXtended</td>
</tr>
</tbody>
</table>

Fp | MOVe | NEGaTe | ABSolute | Single |

Fp |

```

Control Transfer

```
<table>
<thead>
<tr>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
</tr>
<tr>
<td>Floating-point</td>
</tr>
</tbody>
</table>

normal

| Anull delay |
| instruction |

Jump and Link

```
<table>
<thead>
<tr>
<th>RETurn from Trap</th>
</tr>
</thead>
</table>

CALL | Trap on Integer CC |
Definitions

This section provides a detailed definition for each ERC 32 instruction. Each definition includes: the instruction operation; suggested assembly language syntax; a description of the salient features, restrictions and trap conditions; a list of synchronous or floating-point coprocessor traps which can occur as a consequence of executing the instruction; and the instruction format and op codes. Instructions are defined in alphabetical order with the instruction mnemonic shown in large bold type at the top of the PageBreak for easy reference. The instruction set summary that precedes the definitions, (Table 2), groups the instructions by type.

*Table 1.* identifies the abbreviations and symbols used in the instruction definitions. An example of how some of the description notations are used is given below in Figure 2. Register names, labels and other aspects of the syntax used in these instructions are described in the previous section.

**Figure 2.** Instruction Description

![Instruction Description Diagram]

**Description:** The LDD instruction moves a doubleword from memory into a destination register pair, r[rd] and r[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the...
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Instruction field that controls instruction annulling during control transfers</td>
</tr>
<tr>
<td>AND, OR XOR, etc.</td>
<td>AND, OR, XOR, etc operators</td>
</tr>
<tr>
<td>asr_reg</td>
<td>Any implemented ASR (Ancillary State)</td>
</tr>
<tr>
<td>c</td>
<td>The icc carry bit</td>
</tr>
<tr>
<td>ccc</td>
<td>The coprocessor condition code field of the CCSR</td>
</tr>
<tr>
<td>CONCAT</td>
<td>Concatenate</td>
</tr>
<tr>
<td>cond</td>
<td>Instruction field that selects the condition code test for branches</td>
</tr>
<tr>
<td>creg</td>
<td>Communication Coprocessor Register: can be %ccsr, %ccfr, %ccpr, %ccrc</td>
</tr>
<tr>
<td>CWP</td>
<td>PSR's Current Window Pointer field</td>
</tr>
<tr>
<td>disp22</td>
<td>Instruction field that contains the 22-bit sign-extended displacement for branches</td>
</tr>
<tr>
<td>ET</td>
<td>PSR's Enable Traps bit</td>
</tr>
<tr>
<td>i</td>
<td>Instruction field that selects rs2 or sign_extend(simm13) as the second operand</td>
</tr>
<tr>
<td>icc</td>
<td>The integer condition code field of the PSR</td>
</tr>
<tr>
<td>imm22</td>
<td>Instruction field that contains the 22-bit constant used by SETHI</td>
</tr>
<tr>
<td>n</td>
<td>The icc negative bit</td>
</tr>
<tr>
<td>not</td>
<td>Logical complement operator</td>
</tr>
<tr>
<td>nPC</td>
<td>next Program Counter</td>
</tr>
<tr>
<td>opc</td>
<td>Instruction field that specifies the count for Coprocessor-operate instructions</td>
</tr>
<tr>
<td>operand2</td>
<td>Either r[rs2] or sign_extend(simm13)</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>pS</td>
<td>PSR's previous Supervisor bit</td>
</tr>
<tr>
<td>PSR</td>
<td>Processor State Register</td>
</tr>
<tr>
<td>r[15]</td>
<td>A directly addressed register (could be floating-point or coprocessor)</td>
</tr>
<tr>
<td>rd</td>
<td>Instruction field that specifies the destination register (except for store)</td>
</tr>
<tr>
<td>r[rd]</td>
<td>Depending on context, the integer register (or its contents) specified by the instruction field, e.g., rd, rs1, rs2</td>
</tr>
<tr>
<td>r[rd]&lt;31&gt;</td>
<td>&lt;&gt; are used to specify bit fields of a particular register or I/O signal</td>
</tr>
<tr>
<td>[r[rs1] + r[rs2]]</td>
<td>The contents of the address specified by r[rs1] + r[rs2]</td>
</tr>
<tr>
<td>rs1</td>
<td>Instruction field that specifies the source 1 register</td>
</tr>
<tr>
<td>rs2</td>
<td>Instruction field that specifies the source 2 register</td>
</tr>
<tr>
<td>S</td>
<td>PSR's Supervisor bit</td>
</tr>
<tr>
<td>shcnt</td>
<td>Instruction field that specifies the count for shift instructions</td>
</tr>
<tr>
<td>sign_extend(simm13)</td>
<td>Instruction field that contains the 13-bit, sign-extended immediate value</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>TBR</td>
<td>Trap Base Register</td>
</tr>
<tr>
<td>tt</td>
<td>TBR’s trap type field</td>
</tr>
<tr>
<td>uf</td>
<td>Floating-point exception: underflow</td>
</tr>
<tr>
<td>v</td>
<td>The icc overflow bit</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>WIM</td>
<td>Window Invalid Mask register</td>
</tr>
<tr>
<td>Y</td>
<td>Y Register</td>
</tr>
<tr>
<td>z</td>
<td>The icc zero bit</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>x</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td>&lt;--</td>
<td>Replaced by</td>
</tr>
<tr>
<td>7FFFFFF H</td>
<td>Hexadecimal number representation</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
</tr>
</tbody>
</table>
### Table 2. Instruction Set Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDSB(LDSBA*)</td>
<td>Load Signed Byte</td>
<td>(from Alternate Space)</td>
</tr>
<tr>
<td>LDSH(LDSHA*)</td>
<td>Load Signed Halfword</td>
<td>(from Alternate Space)</td>
</tr>
<tr>
<td>LDUB(LDUBA*)</td>
<td>Load Unsigned Byte</td>
<td>(from Alternate Space)</td>
</tr>
<tr>
<td>LDUH(LDUHA*)</td>
<td>Load Unsigned Halfword</td>
<td>(from Alternate Space)</td>
</tr>
<tr>
<td>LD(LDA*)</td>
<td>Load Word</td>
<td>(from Alternate Space)</td>
</tr>
<tr>
<td>LDD(LDDA*)</td>
<td>Load Doubleword</td>
<td>(from Alternate Space)</td>
</tr>
<tr>
<td>LDF</td>
<td>Load Floating Point</td>
<td></td>
</tr>
<tr>
<td>LDFD</td>
<td>Load Double Floating Point</td>
<td></td>
</tr>
<tr>
<td>LDCSR</td>
<td>Load Coprocessor State Register</td>
<td></td>
</tr>
<tr>
<td>STB(STBA*)</td>
<td>Store Byte</td>
<td>(into Alternate Space)</td>
</tr>
<tr>
<td>STH(STHA*)</td>
<td>Store Halfword</td>
<td>(into Alternate Space)</td>
</tr>
<tr>
<td>STD(STDA*)</td>
<td>Store Word</td>
<td>(into Alternate Space)</td>
</tr>
<tr>
<td>STDFD</td>
<td>Store Doubleword</td>
<td></td>
</tr>
<tr>
<td>STF</td>
<td>Store Floating Point</td>
<td></td>
</tr>
<tr>
<td>STFD</td>
<td>Store Double Floating Point</td>
<td></td>
</tr>
<tr>
<td>STFSR</td>
<td>Store Floating Point State Register</td>
<td></td>
</tr>
<tr>
<td>STDSR</td>
<td>Store Double Floating Point State Register</td>
<td></td>
</tr>
<tr>
<td>STC</td>
<td>Store Coprocessor</td>
<td></td>
</tr>
<tr>
<td>STDC</td>
<td>Store Double Coprocessor</td>
<td></td>
</tr>
<tr>
<td>STCSR</td>
<td>Store Coprocessor State Register</td>
<td></td>
</tr>
<tr>
<td>STDCSR</td>
<td>Store Double Coprocessor State Register</td>
<td></td>
</tr>
<tr>
<td>LDSTUB(LDSTUBA*)</td>
<td>Atomic Load/Store Signed Byte (from Alternate Space)</td>
<td></td>
</tr>
<tr>
<td>SWAP(SWAPA*)</td>
<td>Swap r Register with Memory (into Alternate Space)</td>
<td></td>
</tr>
<tr>
<td>ADD(ADDcc)</td>
<td>Add</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>ADDX(ADDXcc)</td>
<td>Add with Carry</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>TADDcc(TADDccTV)</td>
<td>Tagged Add and modify icc</td>
<td>(and Trap on overflow)</td>
</tr>
<tr>
<td>SUB(SUBcc)</td>
<td>Subtract</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>TSUBcc(TSUBccTV)</td>
<td>Tagged Subtract and modify icc</td>
<td>(and Trap on overflow)</td>
</tr>
<tr>
<td>MULScc</td>
<td>Multiply Step and modify icc</td>
<td></td>
</tr>
<tr>
<td>AND(ANDcc)</td>
<td>And</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>ANDN(ANDNcc)</td>
<td>And Not</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>OR(ORcc)</td>
<td>Inclusive Or</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>ORN(ORNcc)</td>
<td>Inclusive Or Not</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>XOR(XORcc)</td>
<td>Exclusive Or</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>XNOR(XNORcc)</td>
<td>Exclusive Nor</td>
<td>(and modify icc)</td>
</tr>
<tr>
<td>SLL</td>
<td>Shift Left Logical</td>
<td></td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Right Logical</td>
<td></td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Right Arithmetic</td>
<td></td>
</tr>
<tr>
<td>SETH</td>
<td>Set High 22 Bits of r Register</td>
<td></td>
</tr>
<tr>
<td>SAVE</td>
<td>Save caller’s window</td>
<td></td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore caller’s window</td>
<td></td>
</tr>
<tr>
<td>Bcc</td>
<td>Branch on Integer Condition Codes</td>
<td></td>
</tr>
<tr>
<td>FBcc</td>
<td>Branch on Floating Point Condition Codes</td>
<td></td>
</tr>
<tr>
<td>CBccc</td>
<td>Branch on Coprocessor Condition Codes</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>Call</td>
<td></td>
</tr>
<tr>
<td>JMPc</td>
<td>Jump and Link</td>
<td></td>
</tr>
<tr>
<td>RETc</td>
<td>Return from Trap</td>
<td></td>
</tr>
<tr>
<td>RDY</td>
<td>Read Y Register</td>
<td></td>
</tr>
<tr>
<td>RDPSR*</td>
<td>Read Processor State Register</td>
<td></td>
</tr>
<tr>
<td>RDWIM*</td>
<td>Read Window Invalid Mask</td>
<td></td>
</tr>
<tr>
<td>RTBR*</td>
<td>Read Trap Base Register</td>
<td></td>
</tr>
<tr>
<td>WRy</td>
<td>Write Y Register</td>
<td></td>
</tr>
<tr>
<td>WRPSR*</td>
<td>Write Processor State Register</td>
<td></td>
</tr>
<tr>
<td>WRWIM*</td>
<td>Write Window Invalid Mask</td>
<td></td>
</tr>
<tr>
<td>WRTRBR*</td>
<td>Write Trap Base Register</td>
<td></td>
</tr>
<tr>
<td>UNIMP*</td>
<td>Unimplemented Instruction</td>
<td></td>
</tr>
<tr>
<td>IFUSH</td>
<td>Instruction Cache Flush</td>
<td></td>
</tr>
<tr>
<td>FP Unit</td>
<td>Floating Point Unit Operations</td>
<td>1 to Launch</td>
</tr>
<tr>
<td>CPop</td>
<td>Coprocessor Operations</td>
<td>1 to Launch</td>
</tr>
</tbody>
</table>

* privileged instruction  
** assuming delay slot is filled with useful instruction
ADD

Add

Operation:

\[ r[rd] \leftarrow r[rs1] + (r[rs2] \text{ or sign extnd(simm13)}) \]

Assembler Syntax:

\text{add} \ vreg1, \ vreg\_or\_imm, \ vregd

Description:

The ADD instruction adds the contents of the register named in the \( rs1 \) field, \( r[rs1] \), to either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or to the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The result is placed in the register specified in the \( rd \) field.

Traps:

none

Format:

\[
\begin{array}{ccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & rd & 000 & 000 & rs1 & i=0 & ignored & & & rs2 \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & rd & 000 & 000 & rs1 & i=1 & & & simm13 \\
\end{array}
\]
### ADDcc

**Add and modify icc**

**Operation:**
\[ r[rd] \rightarrow r[rs1] + \text{operand2} \text{, where operand2} = (r[rs2] \text{ or sign extnd(simm13)}) \]
\[ n \rightarrow r[rd]<31> \]
\[ z \rightarrow \text{if } r[rd] = 0 \text{ then } 1, \text{ else } 0 \]
\[ v \rightarrow (r[rs1]<31> \text{ AND operand2}<31> \text{ AND not } r[rd]<31>) \]
\[ \text{OR (not } r[rs1]<31> \text{ AND not operand2}<31> \text{ AND } r[rd]<31>) \]
\[ c \rightarrow (r[rs1]<31> \text{ AND operand2}<31>) \]
\[ \text{OR (not } r[rd]<31> \text{ AND (r[rs1]<31> OR operand2}<31>) \]

**Assembler Syntax:**
\[
\text{addcc } \text{reg}rs1, \text{ reg}_or_\text{imm}, \text{ reg}rd
\]

**Description:**
ADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction’s i bit equals zero, or to a 13-bit, sign-extended immediate operand if i equals one. The result is placed in the register specified in the rd field. In addition, ADDcc modifies all the integer condition codes in the manner described above.

**Traps:**
none

**Format:**

```
  31 30 29 25 24 19 18 14 13 12 5 4 0
  1 0  rd 0 1 0 0 0 0  rs1 i=0  ignored  rs2

  31 30 29 25 24 19 18 14 13 12 0
  1 0  rd 0 1 0 0 0 0  rs1 i=1  simm13
```
ADDX  

**Add with Carry**

**Operation:**

\[ r[rd] \leftarrow r[rs1] + (r[rs2] \text{ or sign extnd(simm13)}) + c \]

**Assembler Syntax:**

```
addx regrs1, reg_or_imm, regrd
```

**Description:**

ADDX adds the contents of \( r[rs1] \) to either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or to a 13-bit, sign-extended immediate operand if \( i \) equals one. It then adds the PSR’s carry bit \( (c) \) to that result. The final result is placed in the register specified in the \( rd \) field.

**Traps:**

none

**Format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>
```
**ADDXcc**

**Add with Carry and modify icc**

**Operation:**

\[
r[rd] \leftarrow r[rs1] + \text{operand2} + c, \text{ where operand2} = (r[rs2] \text{ or sign extnd(simm13)})
\]

\[
n \leftarrow r[rd]_{<31>}
\]

\[
z \leftarrow \text{if } r[rd] = 0 \text{ then } 1, \text{ else } 0
\]

\[
v \leftarrow (r[rs1]_{<31>} \text{ AND operand2}_{<31>} \text{ AND not } r[rd]_{<31>})
\]

\[
\text{OR (not } r[rs1]_{<31>} \text{ AND not operand2}_{<31>} \text{ AND } r[rd]_{<31>})
\]

\[
c \leftarrow (r[rs1]_{<31>} \text{ AND operand2}_{<31>})
\]

\[
\text{OR (not } r[rd]_{<31>} \text{ AND } r[rs1]_{<31>} \text{ OR operand2}_{<31>})
\]

**Assembler Syntax:**

```plaintext
addxcc regrs1, reg_or_imm, regrd
```

**Description:**

ADDXcc adds the contents of \(r[rs1]\) to either the contents of \(r[rs2]\) if the instruction's \(i\) bit equals zero, or to a 13-bit, sign-extended immediate operand if \(i\) equals one. It then adds the PSR's carry bit (c) to that result. The final result is placed in the register specified in the \(rd\) field. ADDXcc also modifies all the integer condition codes in the manner described above.

**Traps:**

none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0 1 0 0 0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0 1 0 0 0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**SPARC**

12

4168C–AERO–08/01
AND

Operation:  \( r[rd] \leftarrow r[rs1] \text{ AND } (r[rs2] \text{ or sign extnd(simmm13)}) \)

Assembler Syntax:  

\[ \text{and } \text{regrs1, reg_or_imm, regrd} \]

Description:  
This instruction does a bitwise logical AND of the contents of register \( r[rs1] \) with either
the contents of \( r[rs2] \) (if if bit field \( i=0 \)) or the 13-bit, sign-extended immediate value con-
tained in the instruction (if if bit field \( i=1 \)). The result is stored in register \( r[rd] \).

Traps:  
none

Format:

\[
\begin{array}{ccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 0 & 1 & rs1 & i=0 & ignored & rs2 \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 0 & 1 & rs1 & i=1 & simm13 \\
\end{array}
\]
ANDcc  

**And and modify icc**

**Operation:**

\[ r[rd] \leftarrow r[rs1] \text{ AND } (r[rs2] \text{ or sign extnd(simm13)}) \]

\[ n \leftarrow r[rd] < 31 > \]

\[ z \leftarrow \text{if } r[rd] = 0 \text{ then } 1, \text{ else } 0 \]

\[ v \leftarrow 0 \]

\[ c \leftarrow 0 \]

**Assembler Syntax:**

`andcc regrs1, reg_or_imm, regrd`

**Description:**

This instruction does a bitwise logical AND of the contents of register `r[rs1]` with either the contents of `r[rs2]` (if if bit field `i=0`) or the 13-bit, sign-extended immediate value contained in the instruction (if if bit field `i=1`). The result is stored in register `r[rd]`. ANDcc also modifies all the integer condition codes in the manner described above.

**Traps:**

none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>

SPARC
ANDN

And Not

Operation:

\[ r[rd] \leftarrow r[rs1] \text{ AND } (r[rs2] \text{ or sign extnd(simm13)}) \]

Assembler Syntax:

`andn regrs1, reg_or_imm, regrd`

Description:

ANDN does a bitwise logical AND of the contents of register \( r[rs1] \) with the logical complement (not) of either \( r[rs2] \) (if bit field \( i=0 \)) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field \( i=1 \)). The result is stored in register \( r[rd] \).

Traps:

none

Format:

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 1 & 0 & 1 & rs1 & i=0 & ignored & rs2 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 1 & 0 & 1 & rs1 & i=1 & simm13 \\
\end{array}
\]
ANDNcc

And Not and modify icc

Operation:

\[
\begin{align*}
\text{r}[\text{rd}] & \rightarrow \text{r}[\text{rs1}] \ \text{AND} \ \left(\text{r}[\text{rs2}] \text{ or sign extnd(simm13)}\right) \\
n & \rightarrow \text{r}[\text{rd}]<31> \\
z & \rightarrow \text{if r}[\text{rd}] =0 \text{ then 1, else 0} \\
v & \rightarrow 0 \\
c & \rightarrow 0
\end{align*}
\]

Assembler Syntax:

\text{andncc regrs1, reg_or_imm, regrd}

Description:

ANDNcc does a bitwise logical AND of the contents of register r[rs1] with the logical compliment (not) of either r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ANDNcc also modifies all the integer condition codes in the manner described above.

Traps:

none

Format:

\[
\begin{array}{cccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 0 & 1 & 0 & 1 & 0 & 1 & \text{rs1} & \text{i=0 ignored} & \text{rs2} \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & \text{rd} & 0 & 1 & 0 & 1 & 0 & 1 & \text{rs1} & \text{i=1} & \text{simm13} \\
\end{array}
\]
Bicc

Integer Conditional Branch

Operation:

PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
else nPC ← nPC + 4

Assembler Syntax:

ba{},a \textit{label}
bn{},a \textit{label}
\textit{bne}{},a label synonym: \textit{bnz}
be{},a \textit{label} synonym: \textit{bz}
bg{},a \textit{label}
ble{},a \textit{label}
\textit{bge}{},a \textit{label}
bl{},a \textit{label}
\textit{bgu}{},a \textit{label}
bleu{},a \textit{label}
\textit{bcc}{},a \textit{label} synonym: \textit{bgeu}
\textit{bcs}{},a \textit{label} synonym: \textit{blu}
\textit{bpos}{}\textit{a} \textit{label}
\textit{bneg}{}\textit{a} \textit{label}
\textit{bvc}{}\textit{a} \textit{label}
\textit{bvs}{}\textit{a} \textit{label}

Note: The instruction’s annul bit field, \( a \), is set by appending “,a” after the branch name. If it is not appended, the \( a \) field is automatically reset. “,a” is shown in braces because it is optional.

Description:

The Bicc instructions (except for BA and BN) evaluate specific integer condition code combinations (from the PSR’s \textit{icc} field) based on the branch type as specified by the value in the instruction’s \textit{cond} field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address \((PC + 4) + (\text{sign extnd}(\text{disp22}) \times 4)\). If the condition codes evaluate as false, the branch is not taken.

If the branch is not taken, the annul bit field \( a \) is checked. If \( a \) is set, the instruction immediately following the branch instruction (the delay instruction) is not executed (i.e., it is annulled). If the annul field is zero, the delay instruction is executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed.

Branch Never (BN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch Always (BA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if \( a=1 \), the delay instruction is annulled; if \( a=0 \), the delay instruction is executed.

The delay instruction following a Bicc (other than BA) should not be a delayed-control-transfer instruction. The results of following a Bicc with another delayed control transfer instruction are implementation-dependent and therefore unpredictable.
Traps:  none

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Cond.</th>
<th>Operation</th>
<th>icc Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>BN</td>
<td>0000</td>
<td>Branch Never</td>
<td>No test</td>
</tr>
<tr>
<td>BE</td>
<td>0001</td>
<td>Branch on Equal</td>
<td>z</td>
</tr>
<tr>
<td>BLE</td>
<td>0010</td>
<td>Branch on Less or Equal</td>
<td>z OR (n XOR v)</td>
</tr>
<tr>
<td>BL</td>
<td>0011</td>
<td>Branch on Less</td>
<td>n XOR v</td>
</tr>
<tr>
<td>BLEU</td>
<td>0100</td>
<td>Branch on Less or Equal, Unsigned</td>
<td>c OR z</td>
</tr>
<tr>
<td>BCS</td>
<td>0101</td>
<td>Branch on Carry Set</td>
<td>c</td>
</tr>
<tr>
<td>BNEG</td>
<td>0110</td>
<td>Branch on Negative</td>
<td>n</td>
</tr>
<tr>
<td>BVS</td>
<td>0111</td>
<td>Branch on Overflow Set</td>
<td>v</td>
</tr>
<tr>
<td>BA</td>
<td>1000</td>
<td>Branch Always</td>
<td>No test</td>
</tr>
<tr>
<td>BNE</td>
<td>1001</td>
<td>Branch on Not Equal</td>
<td>not z</td>
</tr>
<tr>
<td>BG</td>
<td>1010</td>
<td>Branch on Greater</td>
<td>not(z OR (n XOR v))</td>
</tr>
<tr>
<td>BGE</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>not(n XOR v)</td>
</tr>
<tr>
<td>BGU</td>
<td>1100</td>
<td>Branch on Greater, Unsigned</td>
<td>not(c OR z)</td>
</tr>
<tr>
<td>BCC</td>
<td>1101</td>
<td>Branch on Carry Clear</td>
<td>not c</td>
</tr>
<tr>
<td>BPOS</td>
<td>1110</td>
<td>Branch on Positive</td>
<td>not n</td>
</tr>
<tr>
<td>BVC</td>
<td>1111</td>
<td>Branch on Overflow Clear</td>
<td>not v</td>
</tr>
</tbody>
</table>

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>25</th>
<th>24</th>
<th>22</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a</td>
<td>cond.</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>disp22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

31-30 29 28 25 24 22 21 0
CALL

Operation:
- r[15] \rightarrow PC
- PC \rightarrow nPC
- nPC \rightarrow PC + (disp30 \times 4)

Assembler Syntax:
call label

Description:
The CALL instruction causes a delayed, unconditional, PC-relative control transfer to the address (PC + 4) + (disp30 \times 4). The CALL instruction does not have an annul bit, therefore the delay slot instruction following the CALL instruction is always executed. CALL first writes its return address (PC) into the outs register, r[15], and then adds 4 to the PC. The 32-bit displacement which is added to the new PC is formed by appending two low-order zeros to the 30-bit word displacement contained in the instruction. Consequently, the target address can be anywhere in the ERC 32's user or supervisor address space.

If the instruction following a CALL uses register r[15] as a source operand, hardware interlocks add a one cycle delay.

Programming note: a register-indirect CALL can be constructed using a JMPL instruction with rd set to 15.

Traps:
- none

Format:

<table>
<thead>
<tr>
<th>31 30 29</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>disp30</td>
</tr>
</tbody>
</table>
CBccc

Coprocessor Conditional Branch

Operation:

PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
else nPC ← nPC + 4

Assembler Syntax:

cba{,a}label
cbn{,a}label
cb3{,a}abel
cb2{,a}abel
cb23{,a}label
cb1{,a}label
cb13{,a}label
cb12{,a}label
cb123{,a}label
cb0{,a}abel
cb03{,a}abel
cb02{,a}abel
cb023{,a}label
cb01{,a}label
cb013{,a}label
cb012{,a}label

Note: The instruction’s annul bit field, a, is set by appending “,a” after the branch name. If it is not appended, the a field is automatically reset. “,a” is shown in braces because it is optional.

Description:

The CBccc instructions (except for CBA and CBN) evaluate specific coprocessor condition code combinations (from the CCC<1:0> inputs) based on the branch type as specified by the value in the instruction’s cond field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken.

If the branch is not taken, the annul bit field (a) is checked. If a is set, the instruction immediately following the branch instruction (the delay instruction) is not executed (i.e., it is annulled). If the annul field is zero, the delay instruction is executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed.

Branch Never (CBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch Always (CBA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if a=1, the delay instruction is annulled; if a=0, the delay instruction is executed.

To prevent misapplication of the condition codes, a non-coprocessor instruction must immediately precede a CBccc instruction.

A CBccc instruction generates a cp_disabled trap (and does not branch or annul) if the PSR’s EC bit is reset or if no coprocessor is present.
Traps:

- `cp_disabled`
- `cp_exception`

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>cond.</th>
<th>CCC&lt;1:0&gt; test</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBN</td>
<td>0000</td>
<td>Never</td>
</tr>
<tr>
<td>CB123</td>
<td>0001</td>
<td>1 or 2 or 3</td>
</tr>
<tr>
<td>CB12</td>
<td>0010</td>
<td>1 or 2</td>
</tr>
<tr>
<td>CB13</td>
<td>0011</td>
<td>1 or 3</td>
</tr>
<tr>
<td>CB1</td>
<td>0100</td>
<td>1</td>
</tr>
<tr>
<td>CB23</td>
<td>0101</td>
<td>2 or 3</td>
</tr>
<tr>
<td>CB2</td>
<td>0110</td>
<td>2</td>
</tr>
<tr>
<td>CB3</td>
<td>0111</td>
<td>3</td>
</tr>
<tr>
<td>CBA</td>
<td>1000</td>
<td>Always</td>
</tr>
<tr>
<td>CB0</td>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>CB03</td>
<td>1010</td>
<td>0 or 3</td>
</tr>
<tr>
<td>CB02</td>
<td>1011</td>
<td>0 or 2</td>
</tr>
<tr>
<td>CB023</td>
<td>1100</td>
<td>0 or 2 or 3</td>
</tr>
<tr>
<td>CB01</td>
<td>1101</td>
<td>0 or 1</td>
</tr>
<tr>
<td>CB013</td>
<td>1110</td>
<td>0 or 1 or 3</td>
</tr>
<tr>
<td>CB012</td>
<td>1111</td>
<td>0 or 1 or 2</td>
</tr>
</tbody>
</table>

Format:

```
31 30 29 28 25 24 22 21 0
0 0 a cond. 1 1 1 disp22
```
**CPop**

**Coprocessor Operate**

**Operation:**
Dependent on Coprocessor implementation

**Assembler Syntax:**
Unspecified

**Description:**
CPop1 and CPop2 are the instruction formats for coprocessor operate instructions. The op3 field for CPop1 is 110110; for CPop2 it’s 110111. The coprocessor operations themselves are encoded in the opc field and are dependent on the coprocessor implementation. Note that this does not include load/store coprocessor instructions, which fall into the integer unit’s load/store instruction category.

All CPop instructions take all operands from, and return all results to, the coprocessor’s registers. The data types supported, how the operands are aligned, and whether a CPop generates a cp_exception trap are Coprocessor dependent.

A CPop instruction causes a cp_disabled trap if the PSR’s EC bit is reset or if no coprocessor is present.

**Traps:**
cp_disabled  
cp_exception

**Format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>110110</td>
<td>rs1</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>110111</td>
<td>rs1</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
FABSs

Absolute Value Single
(FPU Instruction Only)

Operation:

\[ f[rd]s \leftarrow f[rs2]s \text{ AND } 7FFFFFFF H \]

Assembler Syntax:

\text{fabss \ fregrs2, fregrd}

Description:

The FABSs instruction clears the sign bit of the word in \( f[rs2] \) and places the result in \( f[rd] \). It does not round.

Since \( rs2 \) can be either an even or odd register, FABSs can also operate on the high-order words of double and extended operands, which accomplishes sign bit clear for these data types.

Traps:

\text{fp\_disabled}

\text{fp\_exception*}

Format:

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 1 & 1 & 0 & 1 & 0 & 0 & \text{ignored} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \text{rs2} \\
\end{array}
\]

Note: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
FADDd

Add Double
(FPU Instruction Only)

Operation:
f[rd]d ← f[rs1]d + f[rs2]d

Assembler Syntax:
fadd d fregrs1, fregrs2, fregrd

Description:
The FADDd instruction adds the contents of f[rs1] CONCAT f[rs1+1] to the contents of
f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard and places
the results in f[rd] and f[rd+1].

Traps:
fp_disabled
fp_exception (of, uf, nv, nx)

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>rs1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
FADDs

Add Single
(FPU Instruction Only)

Operation:
\[ f[rd]s \leftarrow f[rs1]s + f[rs2]s \]

Assembler Syntax:
\textit{fadd} \ freg\textit{rs1}, \ freg\textit{rs2}, \ freg\textit{rd}

Description:
The FADDs instruction adds the contents of \( f[rs1] \) to the contents of \( f[rs2] \) as specified by the ANSI/IEEE 754-1985 standard and places the results in \( f[rd] \).

Traps:
\textit{fp\_disabled}
\textit{fp\_exception (of, uf, nv, nx)}

Format:

\begin{array}{cccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
1 & 0 & rd & 110100 & rs1 & 001000001 & rs2
\end{array}
FADDx

Add Extended
(FPU Instruction Only)

Operation:
\[ f[rd]x \leftarrow f[rs1]x + f[rs2]x \]

Assembler Syntax:
faddx fregrs1, fregrs2, fregrd

Description:
The FADDx instruction adds the contents of \( f[rs1] \) CONCAT \( f[rs1+1] \) CONCAT \( f[rs1+2] \) to the contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) CONCAT \( f[rs2+2] \) as specified by the ANSI/IEEE 754-1985 standard and places the results in \( f[rd] \), \( f[rd+1] \), and \( f[rd+2] \).

Traps:
- fp_disabled
- fp_exception (of, uf, nv, nx)

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
FBfcc

Floating-Point Conditional Branch

Operation:

PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
else nPC ← nPC + 4

Assembler Syntax:

fba{,a}label
fbd{,a}label
fbc{,a}label
fbug{,a}label
fbl{,a}label
fblu{,a}label
fblg{,a}label
fn{,a}label
synonym: fbnz
fbel{,a}label
synonym: fbe
fbue{,a}label
fbge{,a}label
fbuge{,a}label
fbune{,a}label
fbule{,a}label
fbo{,a}label

Note: The instruction’s annul bit field, a, is set by appending “,a” after the branch name. If it is not appended, the a field is automatically reset. “,a” is shown in braces because it is optional.

Description:

The FBfcc instructions (except for FBA and FBN) evaluate specific floating-point condition code combinations (from the FCC<1:0> inputs) based on the branch type, as specified by the value in the instruction’s cond field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken.

If the branch is not taken, the annul bit field (a) is checked. If a is set, the instruction immediately following the branch instruction (the delay instruction) is not executed (i.e., it is annulled). If the annul field is zero, the delay instruction is executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed.

Branch Never (FBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch Always (FBA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if a=1, the delay instruction is annulled; if a=0, the delay instruction is executed.

To prevent misapplication of the condition codes, a non-floating-point instruction must immediately precede an FBfcc instruction.

An FBfcc instruction generates an fp_disabled trap (and does not branch or annul) if the PSR’s EF bit is reset or if no Floating-Point Unit is present.
Traps:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Cond.</th>
<th>Operation</th>
<th>fcc Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBN</td>
<td>0000</td>
<td>Branch Never</td>
<td>no test</td>
</tr>
<tr>
<td>FBNE</td>
<td>0001</td>
<td>Branch on Not Equal</td>
<td>U or L or G</td>
</tr>
<tr>
<td>FBLG</td>
<td>0010</td>
<td>Branch on Less or Greater</td>
<td>L or G</td>
</tr>
<tr>
<td>FBUL</td>
<td>0011</td>
<td>Branch on Unordered or Less</td>
<td>U or L</td>
</tr>
<tr>
<td>FBL</td>
<td>0100</td>
<td>Branch on Less</td>
<td>L</td>
</tr>
<tr>
<td>FBUG</td>
<td>0101</td>
<td>Branch on Unordered or Greater</td>
<td>U or G</td>
</tr>
<tr>
<td>FBG</td>
<td>0110</td>
<td>Branch on Greater</td>
<td>G</td>
</tr>
<tr>
<td>FBU</td>
<td>0111</td>
<td>Branch on Unordered</td>
<td>U</td>
</tr>
<tr>
<td>FBA</td>
<td>1000</td>
<td>Branch Always</td>
<td>no test</td>
</tr>
<tr>
<td>FBE</td>
<td>1001</td>
<td>Branch on Equal</td>
<td>E</td>
</tr>
<tr>
<td>FBUE</td>
<td>1010</td>
<td>Branch on Unordered or Equal</td>
<td>U or E</td>
</tr>
<tr>
<td>FBGE</td>
<td>1011</td>
<td>Branch on Greater or Equal</td>
<td>G or E</td>
</tr>
<tr>
<td>FBUGE</td>
<td>1100</td>
<td>Branch on Unordered or Greater or Equal</td>
<td>U or G or E</td>
</tr>
<tr>
<td>FBLE</td>
<td>1101</td>
<td>Branch on Less or Equal</td>
<td>L or E</td>
</tr>
<tr>
<td>FBULE</td>
<td>1110</td>
<td>Branch on Unordered or Less or Equal</td>
<td>U or L or E</td>
</tr>
<tr>
<td>FBO</td>
<td>1111</td>
<td>Branch on Ordered</td>
<td>L or G or E</td>
</tr>
</tbody>
</table>

Format:

```
31 30 29 28 25 24 22 21 0
0 0 a cond. 1 1 0 disp22
```

Note: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
FCMPd

Compare Double
(FPU Instruction Only)

Operation:
fcc ← f[rs1]d COMPARE f[rs2]d

Assembler Syntax:
fcmpd fregrs1, fregrs2

Description:
FCMPd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR’s fcc bits are set accordingly, and then the result is discarded. The codes are set as follows:

<table>
<thead>
<tr>
<th>fcc</th>
<th>relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fs1 = fs2</td>
</tr>
<tr>
<td>1</td>
<td>fs1 &lt; fs2</td>
</tr>
<tr>
<td>2</td>
<td>fs1 &gt; fs2</td>
</tr>
<tr>
<td>3</td>
<td>fs1 ? fs2 (unordered)</td>
</tr>
</tbody>
</table>

In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPd causes an invalid exception (nv) if either operand is a signaling NaN.

Traps:
fp_disabled
fp_exception (nv)

Format:

```
31 30 29 25 24 19 18 14 13 5 4 0
1 0 ignored 1 1 0 1 0 1 rs1 0 0 1 0 1 0 0 1 0 rs2
```
FCMPEd

Compare Double and Exception if Unordered
(FPU Instruction Only)

**Operation:**

```
fcc ← f[rs1]d COMPARE f[rs2]d
```

**Assembler Syntax:**

```
fcmped fregrs1, fregrs2
```

**Description:**

FCMPEd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's fcc bits are set accordingly, and then the result is discarded. The codes are set as follows:

<table>
<thead>
<tr>
<th>fcc</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fs1 = fs2</td>
</tr>
<tr>
<td>1</td>
<td>fs1 &lt; fs2</td>
</tr>
<tr>
<td>2</td>
<td>fs1 &gt; fs2</td>
</tr>
<tr>
<td>3</td>
<td>fs1 ? fs2 (unordered)</td>
</tr>
</tbody>
</table>

In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEd causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

**Traps:**

- fp_disabled
- fp_exception (nv)

**Format:**

```
31 30 29 25 24 19 18 14 13 5 4 0
1 0 ignored 1 1 0 1 0 1 rs1 0 0 1 0 1 0 1 0 rs2
```
**FCMPEs**

**Compare Single and Exception if Unordered (FPU Instruction Only)**

**Operation:**

\[ \text{fcc} \leftarrow f[\text{rs1}] \text{ COMPARE } f[\text{rs2}] \]

**Assembler Syntax:**

\[ \text{fcmpes fregrs1, fregrs2} \]

**Description:**

FCMPEs subtracts the contents of \( f[\text{rs2}] \) from the contents of \( f[\text{rs1}] \) following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR’s \( \text{fcc} \) bits are set accordingly, and then the result is discarded. The codes are set as follows:

<table>
<thead>
<tr>
<th>fcc</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( f_s1 = f_s2 )</td>
</tr>
<tr>
<td>1</td>
<td>( f_s1 &lt; f_s2 )</td>
</tr>
<tr>
<td>2</td>
<td>( f_s1 &gt; f_s2 )</td>
</tr>
<tr>
<td>3</td>
<td>( f_s1 \ ? \ f_s2 ) (unordered)</td>
</tr>
</tbody>
</table>

In this table, \( f_s1 \) stands for the contents of \( f[\text{rs1}] \) and \( f_s2 \) represents the contents of \( f[\text{rs2}] \).

Compare instructions are used to set up the floating-point condition codes for a subsequent FBFcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBFcc instruction.

FCMPEs causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

**Traps:**

- fp_disabled
- fp_exception (nv)

**Format:**

<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( \text{ignored} ) ( 1 \ 0 \ 1 \ 0 \ 1 ) ( \text{rs1} ) ( 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 )</td>
</tr>
</tbody>
</table>
FCMPEx  

Compare Extended and Exception if Unordered  
(FPU Instruction Only)

Operation:  

fcc ← f[rs1]x COMPARE f[rs2]x

Assembler Syntax:  

fcmpeq fregrs1, fregrs2

Description:  

FCMPEx subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR’s fcc bits are set accordingly, and then the result is discarded. The codes are set as follows:

<table>
<thead>
<tr>
<th>fcc</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fs1 = fs2</td>
</tr>
<tr>
<td>1</td>
<td>fs1 &lt; fs2</td>
</tr>
<tr>
<td>2</td>
<td>fs1 &gt; fs2</td>
</tr>
<tr>
<td>3</td>
<td>fs1 ? fs2 (unordered)</td>
</tr>
</tbody>
</table>

In this table, fs1 stands for the contents of f[rs1], f[rs1+1], f[rs1+2] and fs2 represents the contents of f[rs2], f[rs2+1], f[rs2+2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEx causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps:  

fp_disabled
fp_exception (nv)

Format:  

<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13</th>
<th>5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 ignored</td>
<td>1 1 0 1 0 1</td>
</tr>
</tbody>
</table>
FCMPs

**Compare Single**
**(FPU Instruction Only)**

**Operation:**

fcc ← f[rs1]s COMPARE f[rs2]s

**Assembler Syntax:**

fcmps fregrs1, fregrs2

**Description:**

FCMPs subtracts the contents of f[rs2] from the contents of f[rs1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR’s fcc bits are set accordingly, and then the result is discarded. The codes are set as follows:

<table>
<thead>
<tr>
<th>fcc</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fs1 = fs2</td>
</tr>
<tr>
<td>1</td>
<td>fs1 &lt; fs2</td>
</tr>
<tr>
<td>2</td>
<td>fs1 &gt; fs2</td>
</tr>
<tr>
<td>3</td>
<td>fs1 ? fs2 (unordered)</td>
</tr>
</tbody>
</table>

In this table, fs1 stands for the contents of f[rs1] and fs2 represents the contents of f[rs2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPs causes an invalid exception (nv) if either operand is a signaling NaN.

**Traps:**

fp_disabled
fp_exception (nv)

**Format:**

31 30 29 25 24 19 18 14 13 5 4 0

1 1 ignored 1 1 0 1 0 1 rs1 0 0 1 0 1 0 0 1 rs2
FCMPx

Compare Extended
(FPU Instruction Only)

Operation:

fcc ← f[rs1]x COMPARE f[rs2]x

Assembler Syntax:

fcmpx fregrs1, fregrs2

Description:

FCMPx subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's fcc bits are set accordingly, and then the result is discarded. The codes are set as follows:

<table>
<thead>
<tr>
<th>fcc</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fs1 = fs2</td>
</tr>
<tr>
<td>1</td>
<td>fs1 &lt; fs2</td>
</tr>
<tr>
<td>2</td>
<td>fs1 &gt; fs2</td>
</tr>
<tr>
<td>3</td>
<td>fs1 ≠ fs2 (unordered)</td>
</tr>
</tbody>
</table>

In this table, fs1 stands for the contents of f[rs1], f[rs1+1], f[rs1+2] and fs2 represents the contents of f[rs2], f[rs2+1], f[rs2+2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPx causes an invalid exception (nv) if either operand is a signaling NaN.

Traps:

fp_disabled
fp_exception (nv)

Format:

<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 ignored 1 1 1 0 1 0 0 0 1 1 1</td>
</tr>
</tbody>
</table>

rs1 rs2
FDIVd

Divide Double
(FPU Instruction Only)

Operation:
\[ f[rd]d \leftarrow f[rs1]d / f[rs2]d \]

Assembler Syntax:
\[ \text{fdivd } fregrs1, fregrs2, fregrd \]

Description:
The FDIVd instruction divides the contents of \( f[rs1] \) CONCAT \( f[rs1+1] \) by the contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) as specified by the ANSI/IEEE 754-1985 standard and places the results in \( f[rd] \) and \( f[rd+1] \).

Traps:
- fp_disabled
- fp_exception (of, uf, dz, nv, nx)

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>rd</td>
<td>1 1 0 1 0 0</td>
<td>rs1</td>
<td>0 0 1 0 0 1 1 0</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
FDIVs  

Divide Single  
(FPU Instruction Only)

Operation:  
f[rd]s ← f[rs1]s / f[rs2]s

Assembler Syntax:  
fddivs fregrs1, fregrs2, fregrd

Description:  
The FDIVs instruction divides the contents of f[rs1] by the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps:  
fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:

```
  31  30  29  25  24  19  18  14  13  5  4  0
  1  0  rd  1 1 0 1 0 0  rs1  0 0 1 0 0 1 0 1  rs2
```
FDIVx

**Divide Extended**
*(FPU Instruction Only)*

**Operation:**

f[rd]x ← f[rs1]x / f[rs2]x

**Assembler Syntax:**

fdix \ fregrs1, fregrs2, fregrd

**Description:**

The FDIVx instruction divides the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] by the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], and f[rd+2].

**Traps:**

- fp_disabled
- fp_exception (of, uf, dz, nv, nx)

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>110100</td>
<td>rs1</td>
<td>001001111</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FdTOi  

Convert Double to Integer  
(FPU Instruction Only)

Operation:  
f[rd]i ← f[rs2]d

Assembler Syntax:  
fdtoi fregrs2, fregrd

Description:  
FdTOi converts the floating-point double contents of f[rs2] CONCAT f[rs2+1] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding direction field (RD) of the FSR is ignored.

Traps:  
fp_disabled  
fp_exception (nv, nx)

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>


FdTOs

Convert Double to Single
(FPU Instruction Only)

Operation: \[ f[rd] \leftarrow f[rs2]d \]

Assembler Syntax: \texttt{fdtos \ fregrs2, fregrd}

Description: FdTOs converts the floating-point double contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[rd] \). Rounding is performed according to the rounding direction field (RD) of the FSR.

Traps:
- \texttt{fp\_disabled}
- \texttt{fp\_exception (of, uf, nv, nx)}

Format:

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 1 & 1 & 0 & 1 & 0 & 0 & \text{ignored} & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \text{rs2}
\end{array}
\]
FdTOx

Convert Double to Extended
(FPU Instruction Only)

Operation:

\[ f[\text{rd}]x \leftarrow f[\text{rs2}]d \]

Assembler Syntax:

\text{fdtox} \; f\text{regrs2}, \; f\text{regrd}

Description:

FdTOx converts the floating-point double contents of \( f[\text{rs2}] \) CONCAT \( f[\text{rs2}+1] \) to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[\text{rd}], \; f[\text{rd}+1], \) and \( f[\text{rd}+2] \). Rounding is performed according to the rounding direction \( (RD) \) and rounding precision \( (RP) \) fields of the FSR.

Traps:

- \text{fp\_disabled}
- \text{fp\_exception (nv)}

Format:

\[
\begin{array}{cccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
\hline
1 & 0 & \text{rd} & 1 & 1 & 0 & 1 & 0 & 0 & \text{ignored} & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \text{rs2}
\end{array}
\]
FiTOd  

**Convert Integer to Double**  
*(FPU Instruction Only)*

**Operation:**  
\[ f[rd]d \leftarrow f[rs2]i \]

**Assembler Syntax:**  
`fitod fregrs2, fregrd`

**Description:**  
FiTOd converts the 32-bit, signed integer contents of `f[rs2]` to a floating-point, double-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in `f[rd]` and `f[rd+1]`.

**Traps:**  
- `fp_disabled`
- `fp_exception*`

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
FiTOs

Convert Integer to Single
(FPU Instruction Only)

Operation:
\[ f[rd] \leftarrow f[rs2]i \]

Assembler Syntax:
fitos \textit{fregrs2}, \textit{fregrd}

Description:
FiTOs converts the 32-bit, signed integer contents of \( f[rs2] \) to a floating-point, single-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[rd] \). Rounding is performed according to the rounding direction field, \( RD \).

Traps:

\begin{itemize}
  \item fp\_disabled
  \item fp\_exception (nx)
\end{itemize}

Format:

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
\hline
1 & 0 & \text{rd} & 1 & 1 & 0 & 1 & 0 & 0 & \text{ignored} & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & \text{rs2}
\end{array}
\]
FiTOx

Convert Integer to Extended
(FPU Instruction Only)

Operation:

\[ f[rd]x \leftarrow f[rs2]i \]

Assembler Syntax:

fitox fregrs2, fregrd

Description:

FiTOx converts the 32-bit, signed integer contents of \( f[rs2] \) to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[rd] \), \( f[rd+1] \), and \( f[rd+2] \).

Traps:

fp_disabled
fp_exception*

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
FMOVs

Move
(FPU Instruction Only)

Operation:
\[ f[rd]s \leftarrow f[rs2]s \]

Assembler Syntax:
fmovs fregrs2, fregrd

Description:
The FMOVs instruction moves the word content of register \( f[rs2] \) to the register \( f[rd] \). Multiple FMOVs’s are required to transfer multiple-precision numbers between \( f \) registers.

Traps:
- fp_disabled
- fp_exception*

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>110100</td>
<td>ignored</td>
<td>0000000001</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
FMULd

Multiply Double
(FPU Instruction Only)

Operation:
\[ f[rd] \leftarrow f[rs1] \times f[rs2] \]

Assembler Syntax:
\[ \text{fmuld } fregrs1, fregrs2, fregrd \]

Description:
The FMULd instruction multiplies the contents of \( f[rs1] \) CONCAT \( f[rs1+1] \) by the contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) as specified by the ANSI/IEEE 754-1985 standard and places the results in \( f[rd] \) and \( f[rd+1] \).

Traps:
- fp_disabled
- fp_exception (of, uf, nv, nx)

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
FMULs

Multiply Single
(FPU Instruction Only)

Operation:
\[ f[rd]s \leftarrow f[rs1]s \times f[rs2]s \]

Assembler Syntax:
\texttt{fmuls \ fregrs1, fregrs2, fregrd}

Description:
The FMULs instruction multiplies the contents of \( f[rs1] \) by the contents of \( f[rs2] \) as specified by the ANSI/IEEE 754-1985 standard and places the results in \( f[rd] \).

Traps:
\texttt{fp\_disabled}
\texttt{fp\_exception (of, uf, nv, nx)}

Format:

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
1 & 0 & rd & 1 & 1 & 0 & 1 & 0 & 0 & rs1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & rs2
\end{array}
\]
FMULx  Multiply Extended
        (FPU Instruction Only)

Operation:
        f[rd]x ← f[rs1]x × f[rs2]x

Assembler Syntax:
        fmulp fregrs1, fregrs2, fregrd

Description:
        The FMULx instruction multiplies the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] by the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], and f[rd+2].

Traps:
        fp_disabled
        fp_exception (of, uf, nv, nx)

Format:
        \[
        \begin{array}{ccccccccccc}
        31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
        1 & 0 & \text{rd} & 1 & 1 & 0 & 1 & 0 & 0 & \text{rs1} & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \text{rs2} \\
        \end{array}
        \]
FNEGs

**Negate**
*(FPU Instruction Only)*

**Operation:**
\[ f[rd] \leftarrow f[rs2] \text{ XOR } 80000000 \text{ H} \]

**Assembler Syntax:**
\[ \text{fnegs } \text{fregrs2, fregrd} \]

**Description:**
The FNEGs instruction complements the sign bit of the word in \( f[rs2] \) and places the result in \( f[rd] \). It does not round.

Since this FPop can address both even and odd \( f \) registers, FNEGs can also operate on the high-order words of double and extended operands, which accomplishes sign bit negation for these data types.

**Traps:**
- \( \text{fp\_disabled} \)
- \( \text{fp\_exception}^* \)

**Format:**

<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
</tr>
</tbody>
</table>

**Note:** An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
FSQRTd

**Square Root Double**
*(FPU Instruction Only)*

**Operation:**
\[ f[rd]d \leftarrow \text{SQRT } f[rs2]d \]

**Assembler Syntax:**
`fsqrtd fregrs2, fregrd`

**Description:**
FSQRTd generates the square root of the floating-point double contents of `f[rs2] CONCAT f[rs2+1]` as specified by the ANSI/IEEE 754-1985 standard. The result is placed in `f[rd]` and `f[rd+1]`. Rounding is performed according to the rounding direction field (`RD`) of the FSR.

**Traps:**
- `fp_disabled`
- `fp_exception (nv, nx)`

**Format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>rd</td>
<td>1 1 0 1 0 0</td>
<td>ignored</td>
<td>0 0 0 1 0 1 0 0</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
FSQRTs

Square Root Single
(FPU Instruction Only)

Operation:
f[rd]s ← SQRT f[rs2]s

Assembler Syntax:
fsqrts fregrs2, fregrd

Description:
FSQRTs generates the square root of the floating-point single contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field (RD) of the FSR.

Traps:
fp_disabled
fp_exception (nv, nx)

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**FSQRTx**

**Square Root Extended**

(FPU Instruction Only)

**Operation:**

\[ f[rd]x \leftarrow \text{SQRT } f[rs2]x \]

**Assembler Syntax:**

`fsqrt x fregrs2, fregrd`

**Description:**

FSQRTx generates the square root of the floating-point extended contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) CONCAT \( f[rs2+2] \) as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[rd] \), \( f[rd+1] \), and \( f[rd+2] \). Rounding is performed according to the rounding direction (RD) and rounding precision (RP) fields of the FSR.

**Traps:**

- `fp_disabled`
- `fp_exception (nv, nx)`

**Format:**

\[
\begin{array}{cccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
1 & 0 & rd & 1 & 1 & 0 & 1 & 0 & 0 & ignored & 0 & 0 & 0 & 1 & 0 & 1 & 1 & rs2
\end{array}
\]
FsTOd  Convert Single to Double  (FPU Instruction Only)

Operation:  f[rd]d ← f[rs2]s

Assembler Syntax:  fstod fregrs2, fregrd

Description:  FsTOd converts the floating-point single contents of f[rs2] to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1]. Rounding is performed according to the rounding direction field (RD) of the FSR.

Traps:  
fp_disabled
fp_exception (nv)

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
FsTOi  
Convert Single to Integer  
(FPU Instruction Only)

Operation:  
f[rd]i ← f[rs2]s

Assembler Syntax:  
fstoi  tregrs2, tregrd

Description:  
FsTOi converts the floating-point single contents of f[rs2] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding field (RD) of the FSR is ignored.

Traps:  
fp_disabled  
fp_exception (nv, nx)

Format:

```
31 30 29 25 24 19 18 14 13 5 4 0
1  0  rd  1 1 0 1 0 0  ignored  0 1 1 0 1 0 0 1  rs2
```
FsTOx

Convert Single to Extended
(FPU Instruction Only)

Operation: \( f[rd]x \leftarrow f[rs2]s \)

Assembler Syntax: `fstox fregrs2, fregrd`

Description:
FsTOx converts the floating-point single contents of \( f[rs2] \) to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[rd] \), \( f[rd+1] \), and \( f[rd+2] \). Rounding is performed according to the rounding direction (RD) and rounding precision (RP) fields of the FSR.

Traps:
- `fp_disabled`
- `fp_exception (nv)`

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
FSUBd
Subtract Double
(FPU Instruction Only)

Operation: $f[rd] \leftarrow f[rs1]d - f[rs2]d$

Assembler Syntax: `fsubd fregrs1, fregrs2, fregrd`

Description: The FSUBd instruction subtracts the contents of $f[rs2]$ CONCAT $f[rs2+1]$ from the contents of $f[rs1]$ CONCAT $f[rs1+1]$ as specified by the ANSI/IEEE 754-1985 standard and places the results in $f[rd]$ and $f[rd+1]$.

Traps: `fp_disabled`
`fp_exception (of, uf, nx, nv)`

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
FSUBs

Subtract Single
(FPU Instruction Only)

Operation:
\[ f[rd]s \leftarrow f[rs1]s - f[rs2]s \]

Assembler Syntax:
\texttt{fsubs fregs1, fregrs2, fregrd}

Description:
The FSUBs instruction subtracts the contents of \( f[rs2] \) from the contents of \( f[rs1] \) as specified by the ANSI/IEEE 754-1985 standard and places the results in \( f[rd] \).

Traps:
- \texttt{fp\_disabled}
- \texttt{fp\_exception (of, uf, nx, nv)}

Format:

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 1 & 1 & 0 & 1 & 0 & 0 & \text{rs1} & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & \text{rs2} \\
\end{array}
\]
FSUBx

Subtract Extended
(FPU Instruction Only)

Operation:

\[ f[rd]x \leftarrow f[rs1]x - f[rs2]x \]

Assembler Syntax:

```
fsubx fregrs1, fregrs2, fregrd
```

Description:

The FSUBx instruction subtracts the contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) CONCAT \( f[rs2+2] \) from the contents of \( f[rs1] \) CONCAT \( f[rs1+1] \) CONCAT \( f[rs1+2] \) as specified by the ANSI/IEEE 754-1985 standard and places the results in \( f[rd] \), \( f[rd+1] \), and \( f[rd+2] \).

Traps:

- fp_disabled
- fp_exception (of, uf, nv, nx)

Format:

```
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
\hline
1 & 0 & rd & 1 & 1 & 0 & 1 & 0 & 0 & rs1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & rs2 \\
\hline
\end{array}
```
FxTOd

Convert Extended to Double
(FPU Instruction Only)

Operation:

\[ f[rd]d \leftarrow f[rs2]x \]

Assembler Syntax:

\[ fxtod \ fregrs2, fregrd \]

Description:

FxTOd converts the floating-point extended contents of \([rs2] CONCAT [rs2+1] CONCAT [rs2+2]\) to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \([rd]\) and \([rd+1]\). Rounding is performed according to the rounding direction (RD) field of the FSR.

Traps:

- fp_disabled
- fp_exception (of, uf, nv, nx)

Format:

```
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
FxTOi  Convert Extended to Integer  
(FPU Instruction Only)

Operation:  \( f[rd]i \leftarrow f[rs2]x \)

Assembler Syntax:  
\texttt{ftoi \ iregrs2, iregrd}

Description:  
FxTOi converts the floating-point extended contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) CONCAT \( f[rs2+2] \) to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[rd] \). The rounding field \( (RD) \) of the FSR is ignored.

Traps:  
\texttt{fp\_disabled}
\texttt{fp\_exception (nv, nx)}

Format:

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 1 & 1 & 0 & 1 & 0 & 0 & \text{ignored} & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & \text{rs2}
\end{array}
\]
FxTOs
Convert Extended to Single
(FPU Instruction Only)

Operation: 
\[ f[rd]s \leftarrow f[rs2]x \]

Assembler Syntax: 
fxtos fregrs2, fregrd

Description: 
FxTOs converts the floating-point extended contents of \( f[rs2] \) CONCAT \( f[rs2+1] \) CONCAT \( f[rs2+2] \) to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in \( f[rd] \). Rounding is performed according to the rounding direction (\( RD \)) field of the FSR.

Traps: 
fp_disabled
fp_exception (of, uf, nv, nx)

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>


IFLUSH

**Instruction Cache Flush**

**Operation:**

\[\text{FLUSH} \leftarrow [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))]\]

**Assembler Syntax:**

`iflush address`

**Description:**

The IFLUSH instruction causes a word to be flushed from an instruction cache which may be internal to the processor. The word to be flushed is at the address specified by the contents of \(r[rs1]\) plus either the contents of \(r[rs2]\) if the instruction's \(i\) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \(i\) equals one.

Since there is no internal instruction cache in the current ERC 32 family, the result of executing an IFLUSH instruction is dependent on the state of the input signal, Instruction Cache Flush Trap (IFT). If IFT = 1, IFLUSH executes as a NOP, with no side effects. If IFT = 0, execution of IFLUSH causes an illegal_instruction trap.

**Traps:**

illegal_instruction

**Format:**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>ignored</td>
<td>1 1 1 0 1 1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>ignored</td>
<td>1 1 1 0 1 1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** IFT = 0 in TSC 695
JMPL

Jump and Link

Operation:

\[
\begin{align*}
& r[rd] \rightarrow PC \\
& PC \rightarrow nPC \\
& nPC \rightarrow r[rs1] + (r[rs2] \text{ or sign extnd(simm13)})
\end{align*}
\]

Assembler Syntax:

\text{jmpl} \enspace \text{address, regrd}

Description:

JMPL first provides linkage by saving its return address into the register specified in the \textit{rd} field. It then causes a register-indirect, delayed control transfer to an address specified by the sum of the contents of \textit{r[rs1]} and either the contents of \textit{r[rs2]} if the instruction's \textit{i} bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \textit{i} equals one.

If either of the low-order two bits of the jump address is nonzero, a \texttt{memory	extunderscore address	extunderscore not	extunderscore aligned} trap is generated.

\textit{Programming note}: A register-indirect CALL can be constructed using a JMPL instruction with \textit{rd} set to 15. JMPL can also be used to return from a CALL. In this case, \textit{rd} is set to 0 and the return (jump) address would be equal to \textit{r[31]} + 8.

Traps:

\texttt{memory	extunderscore address	extunderscore not	extunderscore aligned}

Format:

\begin{center}
\begin{tabular}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & \textit{rd} & 1 & 1 & 1 & 0 & 0 & 0 & \textit{i}=0 & \textit{ignored} & \textit{rs2} \\
\hline
1 & 0 & \textit{rd} & 1 & 1 & 1 & 0 & 0 & 0 & \textit{i}=1 & \textit{simm13} & \text{ } & \text{ }
\end{tabular}
\end{center}
LD

Load Word

Operation:

\[ r[rd] \rightarrow [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \]

Assembler Syntax:

\[ \text{ld} \ [\text{address}], \text{regrd} \]

Description:

The LD instruction moves a word from memory into the destination register, \( r[rd] \). The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one.

If LD takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load’s \( r[rd] \) register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:

- memory_address_not_aligned
- data_access_exception

Format:

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
\hline
1 & 1 & \text{rd} & 0 & 0 & 0 & 0 & 0 & \text{rs1} & \text{i=0} & \text{ignored} & \text{rs2} \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 1 & \text{rd} & 0 & 0 & 0 & 0 & 0 & \text{rs1} & \text{i=1} & \text{simm13} \\
\end{array}
\]
LDA

Load Word from Alternate space
(Privileged Instruction)

Operation:

address space ← asi
r[rd] ← [r[rs1] + r[rs2]]

Assembler Syntax:

lda [regaddr] asi, regrd

Description:

The LDA instruction moves a word from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2].

If LDA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Traps:

illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>rs1</td>
<td>i=0</td>
<td>asi</td>
<td>rs2</td>
</tr>
</tbody>
</table>
LDC

Load Coprocessor register

Operation:

\[ c[rd] \leftarrow [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \]

Assembler Syntax:

\textbf{ld} [\textit{address}], \textit{cregrd}

Description:

The LDC instruction moves a word from memory into a coprocessor register, \( c[rd] \). The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one.

If the PSR’s EC bit is set to zero or if no coprocessor is present, a \texttt{cp_disabled} trap will be generated. If LDC takes a trap, the state of the coprocessor depends on the particular implementation.

If the instruction following a coprocessor load uses the load’s \( c[rd] \) register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

\textit{Programming note: If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.}

Traps:

\texttt{cp_disabled}
\texttt{cp_exception}
\texttt{memory_address_not_aligned}
\texttt{data_access_exception}

Format:

\begin{verbatim}
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1 1 0 0 0 0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1 1 0 0 0 0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}
LDCSR

Load Coprocessor State Register

Operation:

CSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax:

ld [address], %csr

Description:

The LDCSR instruction moves a word from memory into the Coprocessor State Register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one.

If the PSR’s EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDCSR takes a trap, the state of the coprocessor depends on the particular implementation.

If the instruction following a LDCSR uses the CSR as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon implementation of the coprocessor.

*Programming note:* If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:

cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i=1</td>
<td></td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>
```
LDD  

Load Doubleword

**Operation:**

\[
\begin{align*}
    r[rd] & \leftarrow [r[rs1] + (r[rs2] \text{ or sign extnd}(\text{simm13})] \\
    r[rd + 1] & \leftarrow [(r[rs1] + (r[rs2] \text{ or sign extnd}(\text{simm13})) + 4]
\end{align*}
\]

**Assembler Syntax:**

\[\text{ldd} \ [\text{address}], \ regrd\]

**Description:**

The LDD instruction moves a doubleword from memory into a destination register pair, \(r[rd]\) and \(r[rd+1]\). The effective memory address is derived by summing the contents of \(r[rs1]\) and either the contents of \(r[rs2]\) if the instruction’s \(i\) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \(i\) equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register.

If a data_access_exception trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load’s \(r[rd]\) register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDD, this applies to both destination registers.

**Programming note:** If \(rs1\) is set to 0 and \(i\) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

**Traps:**

memory_address_not_aligned

data_access_exception

**Format:**

| 31 30 29 25 24 19 18 14 13 12 5 4 0 |
|----------------------|--|--|--|--|--|--|--|--|--|--|--|--|
| 1 1 | rd | 0 0 0 0 1 1 | rs1 | i=0 | ignored | rs2 |

| 31 30 29 25 24 19 18 14 13 12 0 |
|----------------------|--|--|--|--|--|--|--|--|--|--|--|
| 1 1 | rd | 0 0 0 0 1 1 | rs1 | i=1 | simm13 |
LDDA

Load Doubleword from Alternate space
(Privileged Instruction)

Operation:

address space ← asi
r[rd] ← [r[rs1] + r[rs2]]

r[rd +1] ← [r[rs1] + r[rs2] + 4]

Assembler Syntax:

ldda [regaddr] asi, regrd

Description:

The LDDA instruction moves a doubleword from memory into the destination registers, r[rd] and r[rd+1]. The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2]. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register.

If a trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load’s r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDA, this applies to both destination registers.

Traps:

illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>rs1</td>
<td>l=0</td>
<td>asi</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>
LDDC

Load Doubleword Coprocessor

Operation:

\[
\begin{align*}
&c[rd] \leftarrow [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \\
&c[rd + 1] \leftarrow [(r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}) + 4]
\end{align*}
\]

Assembler Syntax:

\text{ldd [address], cregrd}

Description:

The LDDC instruction moves a doubleword from memory into the coprocessor registers, \(c[rd]\) and \(c[rd+1]\). The effective memory address is derived by summing the contents of \(r[rs1]\) and either the contents of \(r[rs2]\) if the instruction’s \(i\) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \(i\) equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register.

If the PSR’s EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDDC takes a trap, the state of the coprocessor depends on the particular implementation.

If the instruction following a coprocessor load uses the load’s \(c[rd]\) register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem and coprocessor implementation. For an LDDC, this applies to both destination registers.

Programming note: If \(rs1\) is set to 0 and \(i\) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:

\text{cp_disabled, cp_exception, memory_address_not_aligned, data_access_exception}

Format:

\[
\begin{array}{ccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 1 & rd & 1 & 1 & 0 & 0 & 1 & 1 & rs1 & i=0 & ignored & rs2 \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 1 & rd & 1 & 1 & 0 & 0 & 1 & 1 & rs1 & i=1 & simm13
\end{array}
\]
LDDF

Load Doubleword Floating-Point

Operation:

\[
\begin{align*}
\text{f}[rd] & \leftarrow \text{f}[rs1] + (\text{f}[rs2] \text{ or sign extnd(simm13)])} \\
\text{f}[rd + 1] & \leftarrow (\text{f}[rs1] + (\text{f}[rs2] \text{ or sign extnd(simm13)]) + 4
\end{align*}
\]

Assembler Syntax:

`ldd [address], fregrd`

Description:

The LDDF instruction moves a doubleword from memory into the floating-point registers, \(\text{f}[rd]\) and \(\text{f}[rd+1]\). The effective memory address is derived by summing the contents of \(\text{r}[rs1]\) and either the contents of \(\text{r}[rs2]\) if the instruction’s \(i\) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \(i\) equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register.

If the PSR’s EF bit is set to zero or if no floating-point unit is present, an \(\text{fp\_disabled}\) trap will be generated. If a trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following a floating-point load uses the load’s \(\text{f}[rd]\) register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDF, this applies to both destination registers.

**Programming note:** If \(rs1\) is set to 0 and \(i\) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:

- \(\text{fp\_disabled}\)
- \(\text{fp\_exception}\)
- \(\text{memory\_address\_not\_aligned}\)
- \(\text{data\_access\_exception}\)

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1 0 0 0 1 1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1 0 0 0 1 1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: * An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit
LDF

Load Floating-Point register

Operation:
\[ f[rd] \leftarrow [r[rs1] + (r[rs2] or sign extnd(simm13))] \]

Assembler Syntax:
```
ld [address], fregrd
```

Description:
The LDF instruction moves a word from memory into a floating-point register, \( f[rd] \). The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one.

If the PSR’s EF bit is set to zero or if no Floating-Point Unit is present, an fp_disabled trap will be generated. If LDF takes a trap, the contents of the destination register remain unchanged.

If the instruction following a floating-point load uses the load’s \( f[rd] \) register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

*Programming note:* If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
- fp_disabled
- fp_exception*
- memory_address_not_aligned
- data_access_exception

Format:
```
31 30 29 25 24 19 18 14 13 12 5 4 0
1 1 rd 1 0 0 0 0 0 0 0 rs1 i=0 ignored rs2
```
```
31 30 29 25 24 19 18 14 13 12 0
1 1 rd 1 0 0 0 0 0 0 0 rs1 i=1 simm13
```

Note: *An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.*
LDFSR
Load Floating-Point State Register

Operation:
FSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax:
“ld [address], %fsr

Description:
The LDFSR instruction moves a word from memory into the floating-point state register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one. This instruction will wait for all pending FPops to complete execution before it loads the memory word into the FSR.

If the PSR’s EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If LDFSR takes a trap, the contents of the FSR remain unchanged.

If the instruction following a LDFSR uses the FSR as a source operand, hardware interlocks add one or more cycle delay to the following instruction depending upon the memory subsystem.

Programming note: If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
f_p_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1 0 0 0 1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1 0 0 0 1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
LDSB

**Load Signed Byte**

**Operation:**

\[ r[rd] \leftarrow \text{sign extnd}[r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \]

**Assembler Syntax:**

\[ \text{l dsb } [\text{address}], \text{regrd} \]

**Description:**

The LDSB instruction moves a signed byte from memory into the destination register, \( r[rd] \). The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The fetched byte is right-justified and sign-extended in \( r[rd] \).

If LDSB takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load’s \( r[rd] \) register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

*Programming note:* If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

**Traps:**

- data_access_exception

**Format:**

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 1 & rd & 0 & 0 & 1 & 0 & 0 & 1 & rs1 & i=0 & ignored & rs2 \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 1 & rd & 0 & 0 & 1 & 0 & 0 & 1 & rs1 & i=1 & simm13
\end{array}
\]
LDSBA

Load Signed Byte from Alternate space
(Privileged Instruction)

Operation:
address space ← asi
r[rd] ← sign extnd[rs1 + rs2]

Assembler Syntax:
ldsba [regaddr] asi, regrd

Description:
The LDSBA instruction moves a signed byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and sign-extended in r[rd].

If LDSBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>asi</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>
**LDSH**

**Load Signed Halfword**

**Operation:**

\[ r[rd] \leftarrow \text{sign extnd}[r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \]

**Assembler Syntax:**

`ldsh [address], regrd`

**Description:**

The LDSH instruction moves a signed halfword from memory into the destination register, \( r[rd] \). The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The fetched halfword is right-justified and sign-extended in \( r[rd] \).

If LDSH takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load’s \( r[rd] \) register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

*Programming note:* If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

**Traps:**

- memory_address_not_aligned
- data_access_exception

**Format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>
```
LDSHA
Load Signed Halfword from Alternate space
(Privileged Instruction)

Operation:
address space ← asi
r[rd] ← sign extnd[r[rs1] + r[rs2]]

Assembler Syntax:
lsha [regaddr] asi, regrd

Description:
The LDSHA instruction moves a signed halfword from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched halfword is right-justified and sign-extended in r[rd].

If LDSHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i=0</td>
<td>asi</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>
LDSTUB  Atomic Load/Store Unsigned Byte

Operation:
\[
\begin{align*}
  r[rd] &\leftarrow \text{zero extnd}[r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \\
  [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] &\leftarrow \text{FFFFFFFFFF H}
\end{align*}
\]

Assembler Syntax:
```asm
ldstub [address], regrd
```

Description:
The LDSTUB instruction moves an unsigned byte from memory into the destination register, \( r[rd] \), and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic load/store instructions which address the same byte simultaneously are guaranteed to execute them serially, in some order.

The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The fetched byte is right-justified and zero-extended in \( r[rd] \).

If the instruction following an integer load uses the load’s \( r[rd] \) register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

If LDSTUB takes a trap, the contents of the memory address remain unchanged.

Programming note: If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
data_access_exception

Format:
```
<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13 12 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 rd 0 0 1 1 0 1 rs1 i=0 ignored</td>
</tr>
</tbody>
</table>
```
```
<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13 12 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 rd 0 0 1 1 0 1 rs1 i=1 simm13</td>
</tr>
</tbody>
</table>
```
LDSTUBA  Atomic Load/Store Unsigned Byte in Alternate space (Privileged Instruction)

Operation:
address space  ← asi
r[rd]  ← zero extnd[r[rs1] + r[rs2]]
[r[rs1] + r[rs2]]  ← FFFFFFFFH

Assembler Syntax:
ldstuba [regaddr] asi, regrd

Description:
The LDSTUBA instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic load/store instructions which address the same byte simultaneously are guaranteed to execute them in some serial order.

The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

If the instruction following an integer load uses the load’s r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

If LDSTUBA takes a trap, the contents of the memory address remain unchanged.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>asi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


LDUB

Load Unsigned Byte

Operation:
\[ r[rd] \leftarrow \text{zero extnd}[r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \]

Assembler Syntax:
\text{ldub [address], regrd}

Description:
The LDUB instruction moves an unsigned byte from memory into the destination register, \( r[rd] \). The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The fetched byte is right-justified and zero-extended in \( r[rd] \).

If LDUB takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load’s \( r[rd] \) register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

*Programming note:* If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
\text{data\_access\_exception}

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=1</td>
</tr>
</tbody>
</table>
```
LDUBA

Load Unsigned Byte from Alternate space
(Privileged Instruction)

Operation:
address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]

Assembler Syntax:
lduba [regaddr] asi, regrd

Description:
The LDUBA instruction moves an unsigned byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

If LDUBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>rd</td>
<td>0 1 0 0 0 1</td>
<td>rs1</td>
<td>i=0</td>
<td>asi</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LDUH

Load Unsigned Halfword

Operation:

\[ r[rd] \leftarrow \text{zero extnd}[r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \]

Assembler Syntax:

\text{lduh } [\text{address}], \text{regrd}

Description:

The LDUH instruction moves an unsigned halfword from memory into the destination register, \( r[rd] \). The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The fetched halfword is right-justified and zero-extended in \( r[rd] \).

If LDUH takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load’s \( r[rd] \) register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

*Programming note:* If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:

- memory_address_not_aligned
- data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>25</td>
<td>24</td>
<td>19</td>
<td>18</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>rs1</td>
<td>i=1</td>
<td></td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>

4168C–AERO–08/01
**LDUHA**

**Load Unsigned Halfword from Alternate space**  
*(Privileged Instruction)*

**Operation:**

address space ← asi  

r[rd] ← zero extnd[r[rs1] + r[rs2]]

**Assembler Syntax:**

`lduha [regaddr] asi, regrd`

**Description:**

The LDUHA instruction moves an unsigned halfword from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched halfword is right-justified and zero-extended in r[rd].

If LDUHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

**Traps:**

- illegal_instruction (if i=1)  
- privileged_instruction (if S=0)  
- memory_address_not_aligned  
- data_access_exception

**Format:**

```
  31 30 29 25 24 19 18 14 13 12  5  4  0
  1 1 rd  0 1 0 0 1 0 rs1 i=0 asi rs2
```
MULScc

Multiply Step and modify icc

Operation:

\[ \text{op1} = (n \text{ XOR } v) \text{ CONCAT } r[rs1]<31:1> \]

if \( Y<0> = 0 \) \( \text{op2} = 0 \), else \( \text{op2} = r[rs2] \) or sign extnd(simm13)

\[ r[rd] \leftarrow \text{op1} + \text{op2} \]

\[ Y \leftarrow r[rs1]<0> \text{ CONCAT } Y<31:1> \]

\[ n \leftarrow r[rd]<31> \]

\[ z \leftarrow \text{if } [r[rd]]=0 \text{ then } 1, \text{ else } 0 \]

\[ v \leftarrow ((\text{op1}<31> \text{ AND } \text{op2}<31> \text{ AND } \text{not } r[rd]<31>) \]

\[ \text{OR (not op1<31> AND not op2<31> AND r[rd]<31>))} \]

\[ c \leftarrow ((\text{op1}<31> \text{ AND } \text{op2}<31>) \]

\[ \text{OR (not r[rd] AND (op1<31> OR op2<31>))} \]

Assembler Syntax:

\text{mulscc regrs1, reg_or_imm, regrd}

Description:

The multiply step instruction can be used to generate the 64-bit product of two signed or unsigned words. MULScc works as follows:

1. The “incoming partial product” in \( r[rs1] \) is shifted right by one bit and the high-order bit is replaced by the sign of the previous partial product \( n \text{ XOR v} \). This is operand1.

2. If the least significant bit of the multiplier in the Y register equals zero, then operand2 is set to zero. If the LSB of the Y register equal one, then operand2 becomes the multiplicand, which is either the contents of \( r[rs2] \) if the instruction i field is zero, or sign extnd(simm13) if the i field is one. Operand2 is then added to operand1 and stored in \( r[rd] \) (the outgoing partial product).

3. The multiplier in the Y register is then shifted right by one bit and its high-order bit is replaced by the least significant bit of the incoming partial product in \( r[rs1] \).

4. The PSR’s integer condition codes are updated according to the addition performed in step 2.

Traps:

none

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>100100</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ 31 \ 30 \ 29 \ 25 \ 24 \ 19 \ 18 \ 14 \ 13 \ 12 \ 0 \]

| 1  | 0  | rd | 100100 | rs1 | i=1 | simm13 | 0 |

\[ 31 \ 30 \ 29 \ 25 \ 24 \ 19 \ 18 \ 14 \ 13 \ 12 \ 0 \]
**OR**

**Inclusive-Or**

**Operation:**

\[ r[rd] \leftarrow r[rs1] \text{ OR } (r[rs2] \text{ or sign extnd(simm13})) \]

**Assembler Syntax:**

\[ \text{or } \text{reg}_rs1, \text{ reg_or_imm, reg}_rd \]

**Description:**

This instruction does a bitwise logical OR of the contents of register \( r[rs1] \) with either the contents of \( r[rs2] \) (if bit field \( i=0 \)) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field \( i=1 \)). The result is stored in register \( r[rd] \).

**Traps:**

none

**Format:**

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 1 & 0 & rs1 & i=0 & ignored & rs2 \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 1 & 0 & rs1 & i=1 & simm13 \\
\end{array}
\]
ORcc  

**Inclusive-Or and modify icc**

**Operation:**
\[
\begin{align*}
\text{r[rd]} &\leftarrow \text{r[rs1]} \text{ OR } (\text{r[rs2]} \text{ or sign extnd(simm13)}) \\
\text{n} &\leftarrow \text{r[rd]}<31> \\
\text{z} &\leftarrow \text{if } [\text{r[rd]}]=0 \text{ then 1, else 0} \\
\text{v} &\leftarrow 0 \\
\text{c} &\leftarrow 0
\end{align*}
\]

**Assembler Syntax:**
orcc regrs1, reg_or_imm, regrd

**Description:**
This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ORcc also modifies all the integer condition codes in the manner described above.

**Traps:**
none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>25</td>
<td>24</td>
<td>19</td>
<td>18</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inclusive-Or Not

Operation:

\[ r[rd] \leftarrow r[rs1] \text{ OR not}(\text{operand2}), \text{ where } \text{operand2} = (r[rs2] \text{ or sign extnd(simm13)}) \]

Assembler Syntax:

\texttt{orn regrs1, reg_or_imm, regrd}

Description:

This instruction does a bitwise logical OR of the contents of register \( r[rs1] \) with the one’s complement of either the contents of \( r[rs2] \) (if bit field \( i=0 \)) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field \( i=1 \)). The result is stored in register \( r[rd] \).

Traps:

none

Format:

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 0 & 0 & 0 & 1 & 1 & 0 & \text{rs1} & i=0 & \text{ignored} & \text{rs2} \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & \text{rd} & 0 & 0 & 0 & 1 & 1 & 0 & \text{rs1} & i=1 & \text{simm13} \\
\end{array}
\]
ORNcc

Inclusive-Or Not and modify icc

Operation:

\[
\begin{align*}
\text{r[rd]} &\leftarrow \text{r[rs1] OR not(operand2)}, \text{ where operand2} = (\text{r[rs2]} \text{ or sign extnd(simm13)}) \\
n &\leftarrow \text{r[rd]<31>} \\
z &\leftarrow \text{if } [\text{r[rd]}]=0 \text{ then 1, else 0} \\
v &\leftarrow 0 \\
c &\leftarrow 0
\end{align*}
\]

Assembler Syntax:

orncc regrs1, reg_or_imm, regrd

Description:

This instruction does a bitwise logical OR of the contents of register r[rs1] with the one’s complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ORNcc also modifies all the integer condition codes in the manner described above.

Traps:

none

Format:

\[
\begin{array}{cccccccccccc}
\text{31} & \text{30} & \text{29} & \text{25} & \text{24} & \text{19} & \text{18} & \text{14} & \text{13} & \text{12} & \text{5} & \text{4} & \text{0} \\
1 & 0 & \text{rd} & 0 & 1 & 0 & 1 & 1 & 0 & \text{rs1} & i=0 & \text{ignored} & \text{rs2} \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & \text{rd} & 0 & 1 & 0 & 1 & 1 & 0 & \text{rs1} & i=1 & \text{simm13}
\end{array}
\]
RDPSR

Read Processor State Register
(Privileged Instruction)

Operation:
\[ r[rd] \leftarrow \text{PSR} \]

Assembler Syntax:
\[ \text{rd} \ %\text{psr, regrd} \]

Description:
RDPSR copies the contents of the PSR into the register specified by the \( rd \) field.

Traps:
privileged-instruction (if \( S=0 \))

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>ignored</td>
</tr>
</tbody>
</table>
RDTBR  Read Trap Base Register
(Privileged Instruction)

Operation:  \[ r[rd] \rightarrow \text{TBR} \]

Assembler Syntax:  \[ rd \ %tbr, \ regrd \]

Description:  RDTBR copies the contents of the TBR into the register specified by the \( rd \) field.

Traps:  privileged_instruction (if \( S=0 \))

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>101011</td>
<td></td>
<td>ignored</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RDWIM

Read Window Invalid Mask register
(Privileged Instruction)

Operation:
\[ r[rd] \leftarrow WIM \]

Assembler Syntax:
\[ rd \ %\text{wim}, \ regrd \]

Description:
RDWIM copies the contents of the WIM register into the register specified by the \textit{rd} field.

Traps:
privileged\_instruction (if S=0)

Format:

\begin{array}{cccc}
31 & 30 & 29 & 25 \\
& & & 24 \\
& & & 19 \\
& & & 18 \\
1 & 0 & \text{rd} & 1 0 1 0 1 0 \\
& & & \text{ignored} \\
\end{array}
RDY  

Read Y register

Operation:  
r[rd] → Y

Assembler Syntax:  
rd %y, regrd

Description:  
RDY copies the contents of the Y register into the register specified by the rd field.

Traps:  
none

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
RESTORE

**Operation:**
- \( ncwp \leftarrow CWP + 1 \)
- \( result \leftarrow r[rs1] + (r[rs2] \text{ or sign extnd(simm13)}) \)
- \( CWP \leftarrow ncwp \)
- \( r[rd] \leftarrow result \)
- RESTORE does not affect condition codes

**Assembler Syntax:**
- `restore regrs1, reg_or_imm, regrd`

**Description:**
RESTORE adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND 2ncwp = 1), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of \( r[rs1] \) is added to either the contents of \( r[rs2] \) (field bit \( i = 1 \)) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit \( i = 0 \)). Because the CWP has not been updated yet, \( r[rs1] \) and \( r[rs2] \) are read from the currently addressed window (the called window).

The new CWP value is written into the PSR, causing the previous window (the caller’s window) to become the active window. The result of the addition is now written into the \( r[rd] \) register of the restored window.

Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the ERC 32).

**Traps:**
- window_underflow

**Format:**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{rd} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>( i = 0 )</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{rd} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( i = 1 )</td>
<td>simm13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RET

Return from Trap
(Privileged Instruction)

Operation:
- \( \text{ncwp} \leftarrow \text{CWP} + 1 \)
- \( \text{ET} \leftarrow 1 \)
- \( \text{PC} \leftarrow \text{nPC} \)
- \( \text{nPC} \leftarrow r[\text{rs1}] + (r[\text{rs2}] \text{ or sign extnd(simm13)}) \)
- \( \text{CWP} \leftarrow \text{ncwp} \)
- \( \text{S} \leftarrow \text{pS} \)

Assembler Syntax:
rett \( \text{address} \)

Description:
RET adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND 2\(\text{ncwp} = 1\)), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then RETT causes a delayed control transfer to the address derived by adding the contents of \(r[\text{rs1}]\) to either the contents of \(r[\text{rs2}]\) (field bit \(i = 1\)) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit \(i = 0\)).

Before the control transfer takes place, the new CWP value is written into the PSR, causing the previous window (the one in which the trap was taken) to become the active window. In addition, the PSR’s ET bit is set to one (traps enabled) and the previous Supervisor bit (pS) is restored to the S field.

Although in theory RETT is a delayed control transfer instruction, in practice, RETT must always be immediately preceded by a JMPL instruction, creating a delayed control transfer couple. This has the effect of annulling the delay instruction.

If traps were already enabled before encountering the RETT instruction, an illegal_instruction trap is generated. If traps are not enabled (ET=0) when the RETT is encountered, but (1) the processor is not in supervisor mode (S=0), or (2) the window underflow condition described above occurs, or (3) if either of the two low-order bits of the target address are nonzero, then a reset trap occurs. If a reset trap does occur, the \(tt\) field of the TBR encodes the trap condition: privileged_instruction, window_underflow, or memory_address_not_aligned.

Programming note: To re-execute the trapping instruction when returning from a trap handler, use the following sequence:

- \( \text{jmp}l \%17, \%0 \) ! old PC
- \( \text{rett} \%18 \) ! old nPC

Note: The ERC 32 saves the PC in \(r[17]\) (local 1) and the nPC in \(r[18]\) (local2) of the trap window upon entering a trap.

To return to the instruction after the trapping instruction (e.g., when the trapping instruction is emulated), use the sequence:

- \( \text{jmp}l \%18, \%0 \) ! old nPC
- \( \text{rett} \%18 + 4 \) ! old nPC + 4
RETT

Return from Trap
(Privileged Instruction)

Traps:

- illegal_instruction
- reset (privileged_instruction)
- reset (memory_address_not_aligned)
- reset (window_underflow)

Format:

```
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ignored</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ignored</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
SAVE

Save caller’s window

Operation:

\[
\begin{align*}
\text{ncwp} & \leftarrow \text{CWP - 1} \\
\text{result} & \leftarrow r[rs1] + (r[rs2] \text{ or sign extnd(simm13)}) \\
\text{CWP} & \leftarrow \text{ncwp} \\
r[rd] & \leftarrow \text{result}
\end{align*}
\]

SAVE does not affect condition codes

Assembler Syntax:

```
save regrs1, reg_or_imm, regrd
```

Description:

SAVE subtracts one from the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND 2ncwp = 1), a window_overflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of \(r[rs1]\) is added to either the contents of \(r[rs2]\) (field bit \(i = 1\)) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit \(i = 0\)). Because the CWP has not been updated yet, \(r[rs1]\) and \(r[rs2]\) are read from the currently addressed window (the calling window).

The new CWP value is written into the PSR, causing the active window to become the previous window, and the called window to become the active window. The result of the addition is now written into the \(r[rd]\) register of the new window.

Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the ERC 32).

Traps:

window_overflow

Format:

\[
\begin{array}{ccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
\hline
1 & 0 & \text{rd} & 1 & 1 & 1 & 0 & 0 & \text{rs1} & \text{i=0} & \text{ignored} & \text{rs2} \\
\hline
1 & 0 & \text{rd} & 1 & 1 & 1 & 0 & 0 & \text{rs1} & \text{i=1} & \text{simm13} & \text{0}
\end{array}
\]
SETHI

Set High 22 bits of r register

Operation:
\[ r[rd]<31:10> \leftarrow \text{imm22} \]
\[ r[rd]<9:0> \leftarrow 0 \]

Assembler Syntax:
\text{sethi } \text{const22, regrd}

sethi \ %hi \text{ value, regrd}

Description:
SETHI zeros the ten least significant bits of the contents of r[rd] and replaces its high-order 22 bits with \text{imm22}. The condition codes are not affected.

Programming note: SETHI 0, %0 is the preferred instruction to use as a NOP, because it will not increase execution time if it follows a load instruction.

Traps:
none

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>22</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>imm22</td>
</tr>
</tbody>
</table>
SLL

**Shift Left Logical**

**Operation:**
\[ r[rd] \leftarrow r[rs1] \text{ SLL by (r[rs2] or shcnt)} \]

**Assembler Syntax:**
\[ \text{sll regrs1, reg_or_imm, regrd} \]

**Description:**
SLL shifts the contents of \( r[rs1] \) left by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into \( r[rd] \). No shift occurs if the shift count is zero.

If the \( i \) bit field equals zero, the shift count for SLL is the least significant five bits of the contents of \( r[rs2] \). If the \( i \) bit field equals one, the shift count for SLL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called \textit{shcnt}.

This instruction does \textit{not} modify the condition codes.

**Traps:**
none

**Format:**

\[
\begin{array}{ccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 1 & 0 & 0 & 1 & 0 & 1 & \text{rs1} & \text{i=0} & \text{ignored} & \text{rs2} \\
\end{array}
\]

\[
\begin{array}{ccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 1 & 0 & 0 & 1 & 0 & 1 & \text{rs1} & \text{i=1} & \text{ignored} & \text{shcnt} \\
\end{array}
\]
SRA

Operation:

\[ r[rd] \leftarrow r[rs1] \text{ SRA by (r[rs2] or shcnt)} \]

Assembler Syntax:

\[ \text{sra } \text{regrs1, reg_or_imm, regrd} \]

Description:

SRA shifts the contents of \( r[rs1] \) right by the number of bits specified by the shift count, filling the vacated positions with the MSB of \( r[rs1] \). The shifted results are written into \( r[rd] \). No shift occurs if the shift count is zero.

If the \( i \) bit field equals zero, the shift count for SRA is the least significant five bits of the contents of \( r[rs2] \). If the \( i \) bit field equals one, the shift count for SRA is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called \( \text{shcnt} \).

This instruction does not modify the condition codes.

Programming note: A “Shift Left Arithmetic by 1 (and calculate overflow)” can be implemented with an ADDcc instruction.

Traps:

none

Format:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1 0 0 1 1 1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1 0 0 1 1 1</td>
<td>rs1</td>
<td>i=1</td>
<td>ignored</td>
<td>shcnt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SRL

**Shift Right Logical**

**Operation:**
\[ r[rd] \leftarrow r[rs1] \text{ SRL by } (r[rs2] \text{ or } shcnt) \]

**Assembler Syntax:**
\[ srl \text{ reg}rs1, \text{ reg_or_imm, reg}rd \]

**Description:**
SRL shifts the contents of \( r[rs1] \) right by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into \( r[rd] \). No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SRL is the least significant five bits of the contents of \( r[rs2] \). If the i bit field equals one, the shift count for SRL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called \( shcnt \).

This instruction does not modify the condition codes.

**Traps:**
none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1 0 0 1 1 0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1 0 0 1 1 0</td>
<td>rs1</td>
<td>i=1</td>
<td>ignored</td>
<td>shcnt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ST

Store Word

Operation:

\[[r[rs1] + (r[rs2] or sign extnd(simm13))] \rightarrow r[rd]\]

Assembler Syntax:

\texttt{st regrd, [address]}

Description:

The ST instruction moves a word from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one.

If ST takes a trap, the contents of the memory address remain unchanged.

*Programming note:* If \(rs1\) is set to 0 and \(i\) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:

\begin{itemize}
  \item memory_address_not_aligned
  \item data_access_exception
\end{itemize}

Format:

\begin{center}
\begin{tabular}{ccccccccccc}
  & 31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
  1 & 1 & \text{rd} & 0 & 0 & 0 & 0 & \text{rs1} & \text{i=0} & \text{ignored} & \text{rs2} \\
  \\
  1 & 1 & \text{rd} & 0 & 0 & 0 & 0 & \text{rs1} & \text{i=1} & \text{simm13} & 0 \\
\end{tabular}
\end{center}
STA

Store Word into Alternate space
(Privileged Instruction)

Operation:
address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

Assembler Syntax:
sta regd, [regaddr] asi

Description:
The STA instruction moves a word from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2].

If STA takes a trap, the contents of the memory address remain unchanged.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0 1 0 1 0 0</td>
<td>rs1</td>
<td>i=0</td>
<td>asi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STB

Operation:

\[[r[rs1] + (r[rs2] or sign extnd(simm13))] \rightarrow r[rd]\]

Assembler Syntax:

\texttt{stb \textit{reg}, \textit{rd}, [address]}

synonyms: stub, stsb

Description:

The STB instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s \textit{i} bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \textit{i} equals one.

If STB takes a trap, the contents of the memory address remain unchanged.

\textit{Programming note}: If \textit{rs1} is set to 0 and \textit{i} is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:

data access exception

Format:

\begin{array}{|c|c|c|c|c|c|c|c|c|c|}
\hline
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
\hline
1 & 1 & rd & 0 & 0 & 1 & 0 & 1 & rs1 & i=0 & ignored & rs2 \\
\hline
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
\hline
1 & 1 & rd & 0 & 0 & 1 & 0 & 1 & rs1 & i=1 & simm13 \\
\hline
\end{array}
STBA

Store Byte into Alternate space
(Privileged Instruction)

Operation:
address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

Assembler Syntax:
stba regrd, [regaddr] asi

Description:
The STBA instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the asi field and the address derived by summing the contents of r[rs1] and r[rs2].

If STBA takes a trap, the contents of the memory address remain unchanged.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>asi</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>
STC

**Store Coprocessor register**

**Operation:**

\[ [r\[rs1\] + (r\[rs2\] or sign extnd(simm13))] \rightarrow c[rd] \]

**Assembler Syntax:**

st cregrd, [address]

**Description:**

The STC instruction moves a word from a coprocessor register, c[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one.

If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If STC takes a trap, memory remains unchanged.

*Programming note:* If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

**Traps:**

- cp_disabled
- cp_exception
- memory_address_not_aligned
- data_access_exception

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>

104  SPARC
STCSR

Store Coprocessor State Register

Operation:

\[ [r_{rs1} + (r_{rs2} \text{ or sign extnd(simm13))} ] \rightarrow \text{CSR} \]

Assembler Syntax:

\( \text{st } %\text{csr, [address]} \)

Description:

The STCSR instruction moves the contents of the Coprocessor State Register into memory. The effective memory address is derived by summing the contents of \( r_{rs1} \) and either the contents of \( r_{rs2} \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one.

If the PSR’s EC bit is set to zero or if no coprocessor is present, a \( \text{cp\_disabled} \) trap will be generated. If STCSR takes a trap, the contents of the memory address remain unchanged.

*Programming note:* If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:

- \( \text{cp\_disabled} \)
- \( \text{cp\_exception} \)
- \( \text{memory\_address\_not\_aligned} \)
- \( \text{data\_access\_exception} \)

Format:

\[
\begin{array}{ccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 1 & rd & 1 & 1 & 0 & 1 & 0 & 1 & rs1 & i=0 & ignored & rs2 \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 1 & rd & 1 & 1 & 0 & 1 & 0 & 1 & rs1 & i=1 & simm13
\end{array}
\]
STD

Operation:

\[ [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))] \rightarrow r[rd] \]
\[ [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))} + 4] \rightarrow r[rd + 1] \]

Assembler Syntax:

\text{std } reg, rd, \text{ [address]}

Description:

The STD instruction moves a doubleword from the destination register pair, \( r[rd] \) and \( r[rd+1] \), into memory. The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction's \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:

memory_address_not_aligned
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>25</td>
<td>24</td>
<td>19</td>
<td>18</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STDA  

Store Doubleword into Alternate space  
(Privileged Instruction)

**Operation:**
address space \( \rightarrow \) asi  
\([r_{rs1}] + (r_{rs2} \text{ or } \text{sign extnd(simm13)})] \rightarrow r_{rd}\)  
\([r_{rs1}] + (r_{rs2} \text{ or } \text{sign extnd(simm13)}) + 4 \rightarrow r_{rd+1}\)

**Assembler Syntax:**
`stda regrd, [regaddr] asi`

**Description:**
The STDA instruction moves a doubleword from the destination register pair, \(r_{rd}\) and \(r_{rd+1}\), into memory. The effective memory address is a combination of the address space value given in the `asi` field and the address derived by summing the contents of \(r_{rs1}\) and \(r_{rs2}\). The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

**Traps:**
- illegal_instruction (if i=1)  
- privileged_instruction (if S=0)  
- memory_address_not_aligned  
- data_access_exception

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i=0</td>
<td>asi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**STDC**

**Store Doubleword Coprocessor**

**Operation:**

\[ [r[rs1] + (r[rs2] or sign extnd(simm13))] \rightarrow c[rd] \]
\[ [r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] \rightarrow c[rd + 1] \]

**Assembler Syntax:**

`std cregrd, [address]`

**Description:**

The STDC instruction moves a doubleword from the coprocessor register pair, c[rd] and c[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If the PSR’s EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

*Programming note:* If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

**Traps:**

- cp_disabled
- cp_exception
- memory_address_not_aligned
- data_access_exception

**Format:**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>rd</td>
</tr>
<tr>
<td>30</td>
<td>1100111</td>
</tr>
<tr>
<td>29</td>
<td>rs1</td>
</tr>
<tr>
<td>25</td>
<td>i=0</td>
</tr>
<tr>
<td>24</td>
<td>ignored</td>
</tr>
<tr>
<td>19</td>
<td>rs2</td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

```markdown
31 30 29 25 24 19 18 14 13 12 5 4 0
1 1 1 1 0 1 1 1 rs1 i=0 ignored rs2

31 30 29 25 24 19 18 14 13 12 0
1 1 1 1 0 1 1 1 rs1 i=1 simm13
```
STDCQ

Store Doubleword Coprocessor Queue
(Privileged Instruction)

Operation:

\[
\begin{align*}
[r[rs1] + (r[rs2] \text{ or sign extnd(simm13))] & \rightarrow \text{CQ.ADDR} \\
[r[rs1] + (r[rs2] \text{ or sign extnd(simm13))] + 4 & \rightarrow \text{CQ.INSTR}
\end{align*}
\]

Assembler Syntax:

```
std %cq, [address]
```

Description:

The STDCQ instruction moves the front entry of the Coprocessor Queue into memory. The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4.

If the PSR’s EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

**Programming note:** If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:

- cp_disabled
- cp_exception
- privileged_instruction (if S=0)
- memory_address_not_aligned
- data_access_exception

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td>rs2</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>
```
STDF

Store Doubleword Floating-Point

Operation:

\[ [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \rightarrow f[rd] \]
\[ [r[rs1] + (r[rs2] \text{ or sign extnd(simm13))} + 4] \rightarrow f[rd + 1] \]

Assembler Syntax:

std fregrd, [address]

Description:

The STDF instruction moves a doubleword from the floating-point register pair, f[rd] and f[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

Programming note: If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:

fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:

\[
\begin{array}{cccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 1 & rd & 1 & 0 & 0 & 1 & 1 & 1 & i=0 & ignored & rs2 \\
1 & 1 & rd & 1 & 0 & 0 & 1 & 1 & 1 & i=1 & simm13 & \\
\end{array}
\]

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
STDFQ

Store Doubleword Floating-Point Queue
(Privileged Instruction)

Operation:

\[
[r[rs1] + (r[rs2] or sign extnd(simm13))] \rightarrow \text{FQ.ADDR}
\]
\[
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] \rightarrow \text{FQ.INSTR}
\]

Assembler Syntax:

\text{std} \, \%fq, [\text{address}]

Description:

The STDFQ instruction moves the front entry of the floating-point queue into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4. If the FPU is in exception mode, the queue is then advanced to the next entry, or it becomes empty (as indicated by the qne bit in the FSR).

If the PSR’s EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

*Programming note:* If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:

\begin{itemize}
  \item fp_disabled
  \item privileged_instruction (if S=0)
  \item memory_address_not_aligned
  \item data_access_exception
\end{itemize}

Format:

\begin{verbatim}
  31  30  29  25  24  19  18  14  13  12  5  4  0
  1  1  rd  1 0 0 1 1 0  rs1  i=0  ignored  rs2

  31  30  29  25  24  19  18  14  13  12  0
  1  1  rd  1 0 0 1 1 0  rs1  i=1  simm13
\end{verbatim}

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
STF

**Store Floating-Point register**

**Operation:**

\[ [r[rs1] + (r[rs2] or sign extnd(simm13))] \rightarrow f[rd] \]

**Assembler Syntax:**

```
st fregrd, [address]
```

**Description:**

The STF instruction moves a word from a floating-point register, f[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one.

If the PSR’s EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STF takes a trap, memory remains unchanged.

*Programming note:* If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

**Traps:**

- fp_disabled
- fp_exception*
- memory_address_not_aligned
- data_access_exception

**Format:**

```
<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13 12 5 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 rd 1 0 0 1 0 0 rs1 i=0 ignored rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 30 29 25 24 19 18 14 13 12 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 rd 1 0 0 1 0 0 rs1 i=1 simm13</td>
</tr>
</tbody>
</table>
```

*NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
**STFSR**

**Store Floating-Point State Register**

**Operation:**

\[ [r[rs1] + (r[rs2] or sign extnd(simm13))] \rightarrow \text{FSR} \]

**Assembler Syntax:**

\text{st \%fsr, [address]} \]

**Description:**

The STFSR instruction moves the contents of the Floating-Point State Register into memory. The effective memory address is derived by summing the contents of \( r[rs1] \) and either the contents of \( r[rs2] \) if the instruction’s \( i \) bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if \( i \) equals one. This instruction will wait for all pending FPops to complete execution before it writes the FSR into memory.

If the PSR’s EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STFSR takes a trap, the contents of the memory address remain unchanged.

*Programming note:* If \( rs1 \) is set to 0 and \( i \) is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

**Traps:**

- fp_disabled
- fp_exception*
- memory_address_not_aligned
- data_access_exception

**Format:**

\[
\begin{array}{ccccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
& 1 & 1 & \text{rd} & 1 & 0 & 0 & 1 & 0 & 1 & \text{rs1} & \text{i=0} & \text{ignored} & \text{rs2} \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 & \text{simm13} \\
& 1 & 1 & \text{rd} & 1 & 0 & 0 & 1 & 0 & 1 & \text{rs1} & \text{i=1} & \text{simm13} \\
\end{array}
\]

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.
**STH**

**Store Halfword**

**Operation:**
\[ \text{[r[rs1] + (r[rs2] or sign extnd(simm13))] } \rightarrow \text{r[rd]} \]

**Assembler Syntax:**
sth reg rd, [address]  synonyms: stuh, stsh

**Description:**
The STH instruction moves the least significant halfword from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one.

If STH takes a trap, the contents of the memory address remain unchanged.

*Programming note:* If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

**Traps:**
- memory_address_not_aligned
- data_access_exception

**Format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0 0 0 1 1 0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0 0 0 1 1 0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STHA  Store Halfword into Alternate space  
(Privileged Instruction)

Operation:  
address space  \(\leftarrow\)  asi  
\([r[rs1] + (r[rs2] \text{ or sign extnd(simm13))}] \leftarrow r[rd]\)

Assembler Syntax:  
\text{stha } regrd, [address]  
synonyms: stuha, stsha

Description:  
The STHA instruction moves the least significant halfword from the destination register,  
\(r[rd]\), into memory. The effective memory address is a combination of the address  
space value given in the asi field and the address derived by summing the contents of  
\(r[rs1]\) and \(r[rs2]\).

If STHA takes a trap, the contents of the memory address remain unchanged.

Traps:  
illegal_instruction (if i=1)  
privileged_instruction (if S=0)  
memory_address_not_aligned  
data_access_exception

Format:  
\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 1 & rd & 0 & 1 & 0 & 1 & 1 & 0 & rs1 & i=0 & asi & rs2 \\
\end{array}
\]
**SUB**

Subtract

**Operation:**

\[ r[rd] \rightarrow r[rs1] - (r[rs2] \text{ or sign extnd(simm13)}) \]

**Assembler Syntax:**

```
sub regrs1, reg_or_imm, regrd
```

**Description:**

The SUB instruction subtracts either the contents of the register named in the `rs2` field, \( r[rs2] \), if the instruction’s `i` bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if `i` equals one, from register \( r[rs1] \). The result is placed in the register specified in the `rd` field.

**Traps:**

none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>simm13</td>
</tr>
</tbody>
</table>
**SUBcc**

**Subtract and modify icc**

**Operation:**

\[
\text{r[rd]} \rightarrow \text{r[rs1]} - \text{operand2}, \text{where operand2} = (\text{r[rs2]} \text{ or sign extnd(simm13)})
\]

- **n** \(\rightarrow \text{r[rd]}<31>
- **z** \(\rightarrow \text{if r[rd]} = 0 \text{ then } 1, \text{ else } 0 \)
- **v** \(\rightarrow (\text{r[rs1]}<31> \text{ AND not operand2}<31> \text{ AND not r[rd]}<31>)\)
- **c** \(\rightarrow (\text{not r[rs1]}<31> \text{ AND operand2}<31>)\)

**Assembler Syntax:**

`subcc  regrs1, reg_or_imm, regrd`

**Description:**

The SUBcc instruction subtracts either the contents of register r[rs2] (if the instruction’s i bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if i equals one) from register r[rs1]. The result is placed in register r[rd]. In addition, SUBcc modifies all the integer condition codes in the manner described above.

**Programming note:** A SUBcc instruction with \(rd = 0\) can be used for signed and unsigned integer comparison.

**Traps:**

none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td>rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>
SUBX

Subtract with Carry

Operation:
\[ r[rd] \leftarrow r[rs1] - (r[rs2] \text{ or sign extnd(simm13)}) - c \]

Assembler Syntax:
\text{subx regrs1, reg_or_imm, regrd}

Description:
SUBX subtracts either the contents of register r[rs2] (if the instruction's i bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if i equals one) from register r[rs1]. It then subtracts the PSR's carry bit (c) from that result. The final result is placed in the register specified in the rd field.

Traps:
none

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i=1</td>
</tr>
</tbody>
</table>
**SUBXcc**  

**Subtract with Carry and modify icc**

**Operation:**

\[
\begin{align*}
    r[rd] &\leftarrow r[rs1] - \text{operand2} - c, \text{ where } \text{operand2} = (r[rs2] \text{ or sign extnd(simm13)}) \\
n &\leftarrow r[rd]<31> \\
z &\leftarrow \text{if } r[rd] = 0 \text{ then } 1, \text{ else } 0 \\
v &\leftarrow (r[rs1]<31> \text{ AND not operand2}<31> \text{ AND not } r[rd]<31>) \\
    &\quad \text{OR} (\text{not } r[rs1]<31> \text{ AND operand2}<31> \text{ AND } r[rd]<31>) \\
c &\leftarrow (\text{not } r[rs1]<31> \text{ AND operand2}<31>) \\
    &\quad \text{OR} (r[rd]<31> \text{ AND (not } r[rs1]<31> \text{ OR operand2}<31>))
\end{align*}
\]

**Assembler Syntax:**

```
subxcc  regrs1, reg_or_imm, regrd
```

**Description:**

SUBXcc subtracts either the contents of register r[rs2] (if the instruction’s i bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if i equals one) from register r[rs1]. It then subtracts the PSR’s carry bit (c) from that result. The final result is placed in the register specified in the rd field. In addition, SUBXcc modifies all the integer condition codes in the manner described above.

**Traps:**

none

**Format:**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SWAP

Swap r register with memory

Operation:

\[ \text{word} \leftarrow [r[rs1] + (r[rs2] \text{ or sign extnd(simm13)])] \]
\[ \text{temp} \leftarrow r[rd] \]
\[ r[rd] \leftarrow \text{word} \]
\[ r[rs1] + (r[rs2] \text{ or sign extnd(simm13)]) \leftarrow \text{temp} \]

Assembler Syntax:

\text{swap} \ [\text{source}, \text{regrd}]

Description:

SWAP atomically exchanges the contents of r[rd] with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAP instructions simultaneously are guaranteed to execute them serially, in some order.

The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction’s i bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if i equals one.

If SWAP takes a trap, the contents of the memory address and the destination register remain unchanged.

Programming note: If rs1 is set to 0 and i is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:

memory_address_not_aligned
data_access_exception

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td>rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>
SWAPA

**Swap r register with memory in Alternate space**
*(Privileged Instruction)*

**Operation:**
- `address space ← asi`
- `word ← [r[rs1] + r[rs2]]`
- `temp ← r[rd]`
- `r[rd] ← word`
- `[r[rs1] + r[rs2]] ← temp`

**Assembler Syntax:**

`swapa [regsource] asi, regrd`

**Description:**

SWAPA atomically exchanges the contents of `r[rd]` with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAPA instructions simultaneously are guaranteed to execute them serially, in some order.

The effective memory address is a combination of the address space value given in the `asi` field and the address derived by summing the contents of `r[rs1]` and `r[rs2].`

If SWAPA takes a trap, the contents of the memory address and the destination register remain unchanged.

**Traps:**

- `illegal_instruction` (if `i=1`)
- `privileged_instruction` (if `S=0`)
- `memory_address_notAligned`
- `data_access_exception`

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>asi</td>
<td>rs2</td>
</tr>
</tbody>
</table>
TADDcc

Tagged Add and modify icc

Operation:

\[ r[\text{rd}] \leftarrow r[\text{rs1}] + \text{operand2}, \text{ where } \text{operand2} = \begin{cases} r[\text{rs2}] & \text{if } i = 0 \\ \text{sign extnd}\text{(simm13)} & \text{if } i = 1 \end{cases} \]

- \( n \leftarrow r[\text{rd}]_{<31>} \)
- \( z \leftarrow \) if \( r[\text{rd}] = 0 \) then 1, else 0
- \( v \leftarrow \) \( (r[\text{rs1}]_{<31>} \text{ AND operand2}_{<31>} \text{ AND not } r[\text{rd}]_{<31>}) \text{ OR (not } r[\text{rs1}]_{<31>} \text{ AND not operand2}_{<31>} \text{ AND } r[\text{rd}]_{<31>}) \text{ OR (} r[\text{rs1}]_{<1:0>} \text{¼ 0 OR operand2}_{<1:0>} \text{¼ 0}) \)
- \( c \leftarrow \) \( (r[\text{rs1}]_{<31>} \text{ AND operand2}_{<31>}) \text{ OR (not } r[\text{rd}]_{<31>} \text{ AND (} r[\text{rs1}]_{<31>} \text{ OR operand2}_{<31>}) \)

Assembler Syntax:

taddcc \( \text{regrs1, reg}_or\_imm, \text{regrd} \)

Description:

TADDcc adds the contents of \( r[\text{rs1}] \) to either the contents of \( r[\text{rs2}] \) if the instruction's \( i \) bit equals zero, or to a 13-bit, sign-extended immediate operand if \( i \) equals one. The result is placed in the register specified in the \( \text{rd} \) field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TADDcc modifies all the integer condition codes in the manner described above.

Traps:

none

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i=0</td>
<td>ignored</td>
<td></td>
<td>rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>
TADDccTV

Tagged Add (modify icc) Trap on Overflow

Operation:
result ← r[rs1] + operand2, where operand 2 = (r[rs2] or sign extnd(simm13))
tv ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
   OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
   OR (r[rs1]<1:0> ¼ 0 OR operand2<1:0> ¼ 0)
if tv = 1, then tag overflow trap; else
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← tv
c ← (r[rs1]<31> AND operand2<31>)
   OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))
r[rd] ← result

Assembler Syntax:
taddccctv  regrs1, reg_or_imm, regrd

Description:
TADDccTV adds the contents of r[rs1] to either the contents of r[rs2] if the instruction’s ‘i’ bit equals zero, or to a 13-bit, sign-extended immediate operand if ‘i’ equals one. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

If TADDccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TADDccTV places the result in the register specified in the rd field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

Traps:
tag_overflow

Format:

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>5</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>18</th>
<th>19</th>
<th>24</th>
<th>25</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>rd</td>
<td>1 0 0 0 1 0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>5</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>18</th>
<th>19</th>
<th>24</th>
<th>25</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>rd</td>
<td>1 0 0 0 1 0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ticc

Trap on integer condition codes

Operation:
If condition true, then trap_instruction;
\[ tt \leftarrow 128 + [r[rs1] + (r[rs2] or sign extnd(simm13))]<6:0> \]
else PC → nPC
nPC → nPC + 4

Assembler Syntax:
\[ \text{ta}, \text{tn}, \text{tne}, \text{te}, \text{tg}, \text{tle}, \text{tge}, \text{tl}, \text{tgu}, \text{tleu}, \text{tc}, \text{tc}, \text{tpos}, \text{tneg}, \text{tv}, \text{tv} \]
\[ \text{label} \]
\[ \text{label} \]
\[ \text{synonym: tnz} \]
\[ \text{synonym: tz} \]
\[ \text{synonym: tge} \]
\[ \text{synonym: tz} \]
\[ \text{label} \]
\[ \text{label} \]
\[ \text{label} \]
\[ \text{label} \]
\[ \text{tt} \]
\[ \text{nPC} \]

Description:
A Ticc instruction evaluates specific integer condition code combinations (from the PSR's icc field) based on the trap type as specified by the value in the instruction's cond field. If the specified combination of condition codes evaluates as true, and there are no higher-priority traps pending, then a trap_instruction trap is generated. If the condition codes evaluate as false, the trap is not generated.

If a trap_instruction trap is generated, the tt field of the Trap Base Register (TBR) is written with 128 plus the least significant seven bits of r[rs1] plus either r[rs2] (bit field \( i =0 \)) or the 13-bit sign-extended immediate value contained in the instruction (bit field \( i =1 \)).

Traps:
trap_instruction
### Trap on integer condition codes

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Cond.</th>
<th>Operation</th>
<th>icc Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>TN</td>
<td>0000</td>
<td>Trap Never</td>
<td>No test</td>
</tr>
<tr>
<td>TE</td>
<td>0001</td>
<td>Trap on Equal</td>
<td>z</td>
</tr>
<tr>
<td>TLE</td>
<td>0010</td>
<td>Trap on Less or Equal</td>
<td>z OR (n XOR v)</td>
</tr>
<tr>
<td>TL</td>
<td>0011</td>
<td>Trap on Less</td>
<td>n XOR v</td>
</tr>
<tr>
<td>TLEU</td>
<td>0100</td>
<td>Trap on Less or Equal, Unsigned</td>
<td>c OR z</td>
</tr>
<tr>
<td>TCS</td>
<td>0101</td>
<td>Trap on Carry Set (Less then, Unsigned)</td>
<td>c</td>
</tr>
<tr>
<td>TNEG</td>
<td>0110</td>
<td>Trap on Negative</td>
<td>n</td>
</tr>
<tr>
<td>TVS</td>
<td>0111</td>
<td>Trap on oVerflow Set</td>
<td>v</td>
</tr>
<tr>
<td>TA</td>
<td>1000</td>
<td>Trap Always</td>
<td>No test</td>
</tr>
<tr>
<td>TNE</td>
<td>1001</td>
<td>Trap on Not Equal</td>
<td>not z</td>
</tr>
<tr>
<td>TG</td>
<td>1010</td>
<td>Trap on Greater</td>
<td>not(z OR (n XOR v))</td>
</tr>
<tr>
<td>TGE</td>
<td>1011</td>
<td>Trap on Greater or Equal</td>
<td>not(n XOR v)</td>
</tr>
<tr>
<td>TGU</td>
<td>1100</td>
<td>Trap on Greater, Unsigned</td>
<td>not(c OR z)</td>
</tr>
<tr>
<td>TCC</td>
<td>1101</td>
<td>Trap on Carry Clear (Greater than or Equal, Unsigned)</td>
<td>not c</td>
</tr>
<tr>
<td>TPOS</td>
<td>1110</td>
<td>Trap on Positive</td>
<td>not n</td>
</tr>
<tr>
<td>TVC</td>
<td>1111</td>
<td>Trap on oVerflow Clear</td>
<td>not v</td>
</tr>
</tbody>
</table>

**Format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>2524</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ign.</td>
<td>cond.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>2524</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ign.</td>
<td>cond.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>rs1</td>
</tr>
</tbody>
</table>
```

ign. = ignored
cond. = condition
**TSUBcc**

**Tagged Subtract and modify icc**

**Operation:**

\[
\begin{align*}
\text{r}[\text{rd}] & \rightarrow \text{r}[\text{rs1}] - \text{operand2, where operand2} = (\text{r}[\text{rs2}] \text{ or sign extnd}(\text{simm13})) \\
\text{n} & \rightarrow \text{r}[\text{rd}]<31> \\
\text{z} & \rightarrow \text{if r}[\text{rd}]=0 \text{ then 1, else 0} \\
\text{v} & \rightarrow \left(\text{r}[\text{rs1}]<31> \text{ AND not operand2}<31> \text{ AND not r}[\text{rd}]<31> \right) \text{ OR } \left(\text{not r}[\text{rs1}]<31> \text{ AND operand2}<31> \text{ AND r}[\text{rd}]<31> \right) \text{ OR } \left(\text{r}[\text{rs1}]<1:0> = 0 \text{ OR operand2}<1:0> = 0 \right) \\
\text{c} & \rightarrow \left(\text{not r}[\text{rs1}]<31> \text{ AND operand2}<31> \right) \text{ OR } \left(\text{r}[\text{rd}]<31> \text{ AND (not r}[\text{rs1}]<31> \text{ OR operand2}<31>) \right)
\end{align*}
\]

**Assembler Syntax:**

\[\text{tsubcc \ regrs1, reg_or_imm, regrd}\]

**Description:**

TSUBcc subtracts either the contents of register r[rs2] (if the instruction’s i bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if i equals one) from register r[rs1]. The result is placed in the register specified in the rd field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TSUBcc modifies all the integer condition codes in the manner described above.

**Traps:**

none

**Format:**

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & rd & 1 & 0 & 0 & 0 & 0 & 1 & rs1 & i=0 & ignored & rs2 \\
1 & 0 & rd & 1 & 0 & 0 & 0 & 0 & 1 & rs1 & i=1 & simm13 & 0
\end{array}
\]
TSUBccTV  
Tagged Subtract (modify icc)  
Trap on Overflow

**Operation:**

\[
\text{result} \leftarrow r[r]s1 - \text{operand2}, \text{where } r[r]s2 = (r[r]s2 \text{ or sign extnd(simm13))}
\]
\[
tv \leftarrow (r[r]s1)<31> \text{ AND not operand2}<31> \text{ AND not } (r[rd]<31>) \text{ OR (not } r[r]s1)<31> \text{ AND operand2}<31> \text{ AND r[rd]<31>)}
\]
\[
\text{if } tv = 1, \text{ then tag overflow trap; else}
\]
\[
r[n]d \leftarrow r[rd]<31>
\]
\[
z \leftarrow \text{if } r[rd]=0 \text{ then } 1, \text{ else } 0
\]
\[
v \leftarrow tv
\]
\[
c \leftarrow (\text{not}(r[r]s1)<31>) \text{ AND operand2}<31> \text{ OR}
\]
\[
(r[rd]<31> \text{ AND (not(r[r]s1]<31>) OR operand2<31>))
\]
\[
[r[rd]] \leftarrow \text{result}
\]

**Assembler Syntax:**

tsubccv  reg[rs1], reg_or_imm, reg[rd]

**Description:**

TSUBccTV subtracts either the contents of register r[r]s2 (if the instruction’s i bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if i equals one) from register r[r]s1. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

If TSUBccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TSUBccTV places the result in the register specified in the rd field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

**Traps:**

tag_overflow

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
</tr>
</tbody>
</table>

127
Unimplemented instruction

Operation: illegal instruction trap

Assembler Syntax:
unimp const22

Description: Executing the UNIMP instruction causes an immediate illegal_instruction trap. The value in the const22 field is ignored.

*Programming note:* UNIMP can be used as part of the protocol for calling a function that is expected to return an aggregate value, such as a C-language structure.

1. An UNIMP instruction is placed after (not in) the delay slot after the CALL instruction in the calling function.
2. If the called function is expecting to return a structure, it will find the size of the structure that the caller expects to be returned as the const22 operand of the UNIMP instruction. The called function can check the opcode to make sure it is indeed UNIMP.
3. If the function is not going to return a structure, upon returning, it attempts to execute UNIMP rather than skipping over it as it should. This causes the program to terminate. The behavior adds some run-time checking to an interface that cannot be checked properly at compile time.

Traps: illegal_instruction

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>22</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>ignored</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>CONST 22</td>
</tr>
</tbody>
</table>
WRPSR  

Write Processor State Register  
(Privileged Instruction)

**Operation:**  
PSR ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

**Assembler Syntax:**  
wr regrs1, reg_or_imm, %psr

**Description:**  
WRPSR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the writable subfields of the PSR. However, if the result’s CWP field would point to an unimplemented window, an illegal_instruction trap is generated and the PSR remains unchanged.

WRPSR is a delayed-write instruction:

1. If any of the three instructions following a WRPSR uses any PSR field that WRPSR modified, the value of that field is unpredictable. Note that any instruction which references a non-global register makes use of the CWP, so following WRPSR with three NOPs would be the safest course.

2. If a WRPSR instruction is updating the PSR’s Processor Interrupt Level (PIL) to a new value and is simultaneously setting Enable Traps (ET) to one, this could result in an interrupt trap at a level equal to the old PIL value.

3. If any of the three instructions after a WRPSR instruction reads the modified PSR, the value read is unpredictable.

4. If any of the three instructions after a WRPSR is trapped, a subsequent RDPSR in the trap handler will get the register’s new value.

**Programming note:** Two WRPSR instructions should be used when enabling traps and changing the PIL value. The first WRPSR should specify ET=0 with the new PIL value, and the second should specify ET=1 with the new PIL value.

**Traps:**  
illegal_instruction  
privileged_instruction (if S=0)

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ignored</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs1</td>
<td>rs2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ignored</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>
WRTBR

Write Trap Base Register
(Privileged Instruction)

Operation:
TBR ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax:
wr regrs1, reg_or_imm, %tbr

Description:
WRTBR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the Trap Base Address field of the TBR.

WRTBR is a delayed-write instruction:
1. If any of the three instructions following a WRTBR causes a trap, the TBA used may be either the old or the new value.
2. If any of the three instructions after a WRTBR is trapped, a subsequent RDTBR in the trap handler will get the register’s new TBA value.

Traps:
privileged_instruction (if S=0)

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>ignored</td>
<td>1 1 0 0 1 1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>ignored</td>
<td>1 1 0 0 1 1</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
WRWIM  Write Window Invalid Mask register  
(Privileged Instruction)

Operation:  
\[
WIM \leftarrow r[rs1] \text{ XOR } (r[rs2] \text{ or sign extnd(simm13)})
\]

Assembler Syntax:  
\[
wr \ regrs1, reg_or_imm, \%wim
\]

Description:  
WRWIM does a bitwise logical XOR of the contents of register \(r[rs1]\) with either the contents of \(r[rs2]\) (if bit field \(i=0\)) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field \(i=1\)). The result is written into the writable bits of the WIM register.

WRWIM is a delayed-write instruction:
1. If any of the three instructions following a WRWIM is a SAVE, RESTORE, or RETT, the occurrence of window_overflow and window_underflow is unpredictable.
2. If any of the three instructions after a WRWIM instruction reads the modified WIM, the value read is unpredictable.
3. If any of the three instructions after a WRWIM is trapped, a subsequent RDWIM in the trap handler will get the register’s new value.

Traps:  
privileged_instruction (if S=0)

Format:  
\[
\begin{array}{cccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & \text{ignored} & 1 & 1 & 0 & 0 & 1 & 0 & \text{rs1} & i=0 & \text{ignored} & \text{rs2} \\
1 & 0 & \text{ignored} & 1 & 1 & 0 & 0 & 1 & 0 & \text{rs1} & i=1 & \text{simm13} & 0 \\
\end{array}
\]
WRY

Write Y register

Operation:

\[ Y \leftarrow \text{r[rs1]} \oplus \text{r[rs2] or sign extnd(simm13)} \]

Assembler

Syntax:

\[ \text{wr regrs1, reg_or_imm, %y} \]

Description:

WRY does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the Y register.

WRY is a delayed-write instruction:

1. If any of the three instructions following a WRY is a MULScc or a RDY, the value of Y used is unpredictable.
2. If any of the three instructions after a WRY instruction reads the modified Y register, the value read is unpredictable.
3. If any of the three instructions after a WRY is trapped, a subsequent RDY in the trap handler will get the register’s new value.

Traps:

none

Format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ignored</td>
<td>1 1 0 0 0 0</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ignored</td>
<td>1 1 0 0 0 0</td>
<td>rs1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
XNOR

**Operation:**

\[ r[rd] \leftarrow r[rs1] \text{ XOR not}(r[rs2] \text{ or sign extnd(simm13)}) \]

**Assembler Syntax:**

\[ \text{xnor } regrs1, \text{ reg_or_imm, regrd} \]

**Description:**

This instruction does a bitwise logical XOR of the contents of register \( r[rs1] \) with the one's complement of either the contents of \( r[rs2] \) (if bit field \( i=0 \)) or the 13-bit sign-extended immediate value contained in the instruction (if bit field \( i=1 \)). The result is stored in register \( r[rd] \).

**Traps:**

none

**Format:**

\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & \text{rd} & 0 & 0 & 0 & 1 & 1 & \text{rs1} & \text{i=0} & \text{ignored} & \text{rs2} \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & \text{rd} & 0 & 0 & 0 & 1 & 1 & \text{rs1} & \text{i=1} & \text{simm13} \\
\end{array}
\]
XNORcc  

**Operation:**

\[
\begin{align*}
\text{r}\{\text{rd}\} & \leftarrow \text{r}\{\text{rs1}\} \text{ XOR } \neg (\text{r}\{\text{rs2}\} \text{ or } \text{sign extnd(simm13)}) \\
n & \leftarrow \text{r}\{\text{rd}\}^{<31>} \\
z & \leftarrow \text{if r}\{\text{rd}\} = 0 \text{ then 1, else 0} \\
v & \leftarrow 0 \\
c & \leftarrow 0
\end{align*}
\]

**Assembler Syntax:**

\text{xnorcc regrs1, reg_or_imm, regrd}

**Description:**

This instruction does a bitwise logical XOR of the contents of register r\{rs1\} with the one’s complement of either the contents of r\{rs2\} (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r\{rd\}. XNORcc also modifies all the integer condition codes in the manner described above.

**Traps:**

none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i=1</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>
XOR

Operation:  
\[ r[rd] \leftarrow r[rs1] \text{ XOR } (r[rs2] \text{ or sign extnd(simm13)}) \]

Assembler Syntax:  
\text{xor regrs1, reg_or_imm, regrd}

Description:  
This instruction does a bitwise logical XOR of the contents of register \(r[rs1]\) with either the contents of \(r[rs2]\) (if bit field \(i=0\)) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field \(i=1\)). The result is stored in register \(r[rd]\).

Traps:  
none

Format:  
\[
\begin{array}{cccccccccc}
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 5 & 4 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 0 & 1 & 1 & rs1 & i=0 & ignored & rs2 \\
31 & 30 & 29 & 25 & 24 & 19 & 18 & 14 & 13 & 12 & 0 \\
1 & 0 & rd & 0 & 0 & 0 & 0 & 1 & 1 & rs1 & i=1 & simm13 \\
\end{array}
\]
**XORcc**

**Exclusive-Or and modify icc**

**Operation:**

\[ r[rd] \leftarrow r[rs1] \text{ XOR } (r[rs2] \text{ or sign extnd(simm13)}) \]

\[ n \leftarrow r[rd]<31> \]

\[ z \leftarrow \text{if } r[rd] =0 \text{ then } 1, \text{ else } 0 \]

\[ v \leftarrow 0 \]

\[ c \leftarrow 0 \]

**Assembler Syntax:**

$xorc \text{ rsrs1, reg_or_imm, regrd}$

**Description:**

This instruction does a bitwise logical XOR of the contents of register \( r[rs1] \) with either the contents of \( r[rs2] \) (if bit field \( i=0 \)) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field \( i=1 \)). The result is stored in register \( r[rd] \). XORcc also modifies all the integer condition codes in the manner described above.

**Traps:**

none

**Format:**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>rs1</td>
<td>i=0</td>
<td>ignored</td>
<td>rs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>25</th>
<th>24</th>
<th>19</th>
<th>18</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>rd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>rs1</td>
<td>i=1</td>
</tr>
</tbody>
</table>