SPIM Instructions

Instructions marked with a dagger (†) are pseudoinstructions.

Arithmetic Instructions

In all instructions below, $\text{Src2}$ can either be a register or an immediate value (a 16 bit integer). The immediate forms of the instructions are only included for reference. The assembler will translate the more general form of an instruction (e.g., $\text{add}$) into the immediate form (e.g., $\text{addi}$) if the second argument is a constant.

---

Absolute Value

Put the absolute value of the integer from register $\text{Rsrc}$ in register $\text{Rdest}$:

$$\text{abs } \text{Rdest}, \text{Rsrc}$$  

$\text{Absolute Value} \, †$

---

Add

Put the sum of the integers from registers $\text{Rs}$ and $\text{Rt}$ (or $\text{Imm}$) into register $\text{Rd}$:

$$\text{add } \text{Rd}, \text{Rs}, \text{Rt}$$  

$\text{Addition (with overflow)}$

$$\text{addu } \text{Rd}, \text{Rs}, \text{Rt}$$  

$\text{Addition (without overflow)}$

$$\text{addi } \text{Rt}, \text{Rs}, \text{Imm}$$  

$\text{Addition Immediate (with overflow)}$

$$\text{addiu } \text{Rt}, \text{Rs}, \text{Imm}$$  

$\text{Addition Immediate (without overflow)}$
**Subtract**

Put the difference of the integers from register Rs and Rt into register Rd:

\[
\text{sub Rd, Rs, Rt} \quad \text{Subtract (with overflow)}
\]

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Multiply**

Put the product of registers Rs_{c1} and Src_{c2} into register Rdest:

\[
\text{mul Rdest, Rs_{c1}, Src_{c2}} \quad \text{Multiply (without overflow)}
\]

\[
\text{mulo Rdest, Rs_{c1}, Src_{c2}} \quad \text{Multiply (with overflow)}
\]

\[
\text{mulou Rdest, Rs_{c1}, Src_{c2}} \quad \text{Unsigned Multiply (with overflow)}
\]

Multiply the contents of registers Rs and Rt. Leave the low-order word of the product in register lo and the high-word in register hi:

\[
\text{mult Rs, Rt} \quad \text{Multiply}
\]

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>5</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>5</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

**Divide**

Divide the integer in register Rs by the integer in register Rt. Leave the quotient in register lo and the remainder in register hi:

\[
\text{div Rs, Rt} \quad \text{Divide (with overflow)}
\]

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>5</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>
\textbf{divu Rs, Rt} \hspace{1cm} \textit{Divide (without overflow)}

<table>
<thead>
<tr>
<th>0</th>
<th>Rs</th>
<th>Rt</th>
<th>0</th>
<th>0x1b</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

Put the quotient of the integers from register \texttt{Rsrc1} and \texttt{Src2} into register \texttt{Rdest}:

\begin{align*}
div & \texttt{Rdest, Rsrc1, Src2} \quad \textit{Divide (with overflow)}^* \\
divu & \texttt{Rdest, Rsrc1, Src2} \quad \textit{Divide (without overflow)}^* \\
\end{align*}

\textbf{Negative}

Put the negative of the integer from register \texttt{Rsrc} into register \texttt{Rdest}:

\begin{align*}
\textit{Negate Value (with overflow)}^* \\
\textit{Negate Value (without overflow)}^* \\
\end{align*}

\textbf{Logical Operations}

Put the logical AND of the integers from register \texttt{Rs} and register \texttt{Rt} (or the zero-extended immediate value \texttt{Imm}) into register \texttt{Rd}:

\begin{align*}
\text{and} & \texttt{Rd, Rs, Rt} \quad \textit{AND} \\
\text{andi} & \texttt{Rd, Rs, Imm} \quad \textit{AND Immediate} \\
\end{align*}

<table>
<thead>
<tr>
<th>0</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>0</th>
<th>0x24</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0xc</th>
<th>Rs</th>
<th>Rd</th>
<th>Imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Put the logical NOR of the integers from register \texttt{Rs} and \texttt{Rt} into register \texttt{Rd}:

\begin{align*}
nor & \texttt{Rd, Rs, Rt} \quad \textit{NOR} \\
\end{align*}

<table>
<thead>
<tr>
<th>0</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>0</th>
<th>0x27</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
Put the bitwise logical negation of the integer from register `Rsrc` into register `Rdest`:

\[ \text{not } Rdest, Rsrc \]  \quad \text{\textit{NOT}}^{\dagger}

Put the logical OR of the integers from register `Rs` and `Rt` (or `Imm`) into register `Rd`:

\[ \text{or } Rd, Rs, Rt \]  \quad \text{\textit{OR}}

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \text{ori } Rt, Rs, Imm \]  \quad \text{\textit{OR Immediate}}

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th>Imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Put the logical XOR of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`:

\[ \text{xor } Rd, Rs, Rt \]  \quad \text{\textit{XOR}}

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \text{xori } Rt, Rs, Imm \]  \quad \text{\textit{XOR Immediate}}

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th>Rt</th>
<th>Imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

**Remainder**

Put the remainder from dividing the integer in register `Rsrc1` by the integer in `Src2` into register `Rdest`:

\[ \text{rem } Rdest, Rsrc1, Src2 \]  \quad \text{\textit{Remainder}}^{\dagger}
\[ \text{remu } Rdest, Rsrc1, Src2 \]  \quad \text{\textit{Unsigned Remainder}}^{\dagger}

Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.
Rotate and Shift Instructions

Rotate the contents of register $R_{src1}$ left (right) by the distance indicated by $Src2$ and put the result in register $R_{dest}$:

\[
\text{rol } R_{dest}, R_{src1}, Src2 \quad \text{(Rotate Left)}
\]
\[
\text{ror } R_{dest}, R_{src1}, Src2 \quad \text{(Rotate Right)}
\]

Shift the contents of register $Rt$ left (right) by the distance indicated by $Sa$ ($Rs$) and put the result in register $Rd$:

\[
\text{sll } Rd, Rt, Sa \quad \text{(Shift Left Logical)}
\]
\[
\text{sllv } Rd, Rt, Rs \quad \text{(Shift Left Logical Variable)}
\]
\[
\text{sra } Rd, Rt, Sa \quad \text{(Shift Right Arithmetic)}
\]
\[
\text{srav } Rd, Rt, Rs \quad \text{(Shift Right Arithmetic Variable)}
\]
\[
\text{srl } Rd, Rt, Sa \quad \text{(Shift Right Logical)}
\]
\[
\text{srlv } Rd, Rt, Rs \quad \text{(Shift Right Logical Variable)}
\]
Constant-Manipulating Instructions

Move the immediate *imm* into register *Rdest*:

\[
\text{li } Rdest, \text{ imm} \quad \textit{Load Immediate} \dagger
\]

Load the lower halfword of the immediate *imm* into the upper halfword of register *Rdest*. The lower bits of the register are set to 0:

\[
\text{lui } Rt, \text{ imm} \quad \textit{Load Upper Immediate}
\]

<table>
<thead>
<tr>
<th>0xf</th>
<th>Rs</th>
<th>Rt</th>
<th>Imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Comparison Instructions

In all instructions below, *Src2* can either be a register or an immediate value (a 16 bit integer).

Set register *Rdest* to 1 if register *Rsrc1* equals *Src2* and to 0 otherwise:

\[
\text{seq } Rdest, Rsrc1, \text{ Src2} \quad \textit{Set Equal} \dagger
\]

Set register *Rdest* to 1 if register *Rsrc1* is greater than or equal to *Src2* and to 0 otherwise:

\[
\text{sge } Rdest, Rsrc1, \text{ Src2} \quad \textit{Set Greater Than Equal} \dagger
\]

\[
\text{sgeu } Rdest, Rsrc1, \text{ Src2} \quad \textit{Set Greater Than Equal Unsigned} \dagger
\]

Set register *Rdest* to 1 if register *Rsrc1* is greater than *Src2* and to 0 otherwise:

\[
\text{sgt } Rdest, Rsrc1, \text{ Src2} \quad \textit{Set Greater Than} \dagger
\]

\[
\text{sgtu } Rdest, Rsrc1, \text{ Src2} \quad \textit{Set Greater Than Unsigned} \dagger
\]

Set register *Rdest* to 1 if register *Rsrc1* is less than or equal to *Src2* and to 0 otherwise:

\[
\text{sle } Rdest, Rsrc1, \text{ Src2} \quad \textit{Set Less Than Equal} \dagger
\]

\[
\text{sleu } Rdest, Rsrc1, \text{ Src2} \quad \textit{Set Less Than Equal Unsigned} \dagger
\]
Set register $R_{dest}$ to 1 if register $R_{src1}$ is less than $Src2$ (or $Imm$) and to 0 otherwise:

$\text{slt } Rd, Rs, Rt$ \hspace{1cm} $\textbf{Set Less Than}$

<table>
<thead>
<tr>
<th>0</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>0</th>
<th>0x2a</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

$\text{sltu } Rd, Rs, Rt$ \hspace{1cm} $\textbf{Set Less Than Unsigned}$

<table>
<thead>
<tr>
<th>0</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>0</th>
<th>0x2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

$\text{slti } Rd, Rs, Imm$ \hspace{1cm} $\textbf{Set Less Than Immediate}$

<table>
<thead>
<tr>
<th>0xa</th>
<th>Rs</th>
<th>Rt</th>
<th>Imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

$\text{sltiu } Rd, Rs, Imm$ \hspace{1cm} $\textbf{Set Less Than Unsigned Immediate}$

<table>
<thead>
<tr>
<th>0xb</th>
<th>Rs</th>
<th>Rt</th>
<th>Imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Set register $R_{dest}$ to 1 if register $R_{src1}$ is not equal to $Src2$ and to 0 otherwise:

$sne R_{dest}, R_{src1}, Src2$ \hspace{1cm} $\textbf{Set Not Equal}$

---

**Branch and Jump Instructions**

In all instructions below, $Src2$ can either be a register or an immediate value (integer). Branch instructions use a signed 16-bit offset field; hence they can jump $2^{15} - 1$ instructions (not bytes) forward or $2^{15}$ instructions backwards. The jump instruction contains a 26 bit address field.

For branch instructions, the offset of the instruction at a label is computed by the assembler.

Unconditionally branch to the instruction at the label:

$b \text{ label}$ \hspace{1cm} $\textbf{Branch pseudoinstruction}$

---
Conditionally branch to the instruction at the label if coprocessor z’s condition flag is true (false):

\textbf{bczt label} \hspace{1cm} \textit{Branch Coprocessor z True}
\begin{tabular}{c|c|c|c}
0x1 & 8 & 1 & Offset \\
6 & 5 & 5 & 16
\end{tabular}

\textbf{bczf label} \hspace{1cm} \textit{Branch Coprocessor z False}
\begin{tabular}{c|c|c|c}
0x1 & 8 & 0 & Offset \\
6 & 5 & 5 & 16
\end{tabular}

Conditionally branch to the instruction at the label if the contents of register Rs equals the contents of register Rt:

\textbf{beq Rs, Rt, label} \hspace{1cm} \textit{Branch on Equal}
\begin{tabular}{c|c|c|c}
4 & Rs & Rt & Offset \\
6 & 5 & 5 & 16
\end{tabular}

Conditionally branch to the instruction at the label if the contents of Rs are greater than or equal to 0:

\textbf{bgez Rs, label} \hspace{1cm} \textit{Branch on Greater Than Equal Zero}
\begin{tabular}{c|c|c|c}
1 & Rs & 1 & Offset \\
6 & 5 & 5 & 16
\end{tabular}
Conditionally branch to the instruction at the label if the contents of Rs are greater than or equal to 0. Save the address of the next instruction in register 31:

\[
\text{bgezal Rs, label} \quad \text{Branch on Greater Than Equal Zero And Link}
\]

<table>
<thead>
<tr>
<th>1</th>
<th>Rs</th>
<th>0x11</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are greater than Src2:

\[
bgt \text{ Rsrc1, Src2, label} \quad \text{Branch on Greater Than} \quad \uparrow
\]

\[
bgtu \text{ Rsrc1, Src2, label} \quad \text{Branch on Greater Than Unsigned} \quad \uparrow
\]

Conditionally branch to the instruction at the label if the contents of Rs are greater than 0:

\[
bgtz \text{ Rs, label} \quad \text{Branch on Greater Than Zero}
\]

<table>
<thead>
<tr>
<th>7</th>
<th>Rs</th>
<th>0</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are less than or equal to Src2:

\[
\text{ble Rsrc1, Src2, label} \quad \text{Branch on Less Than Equal} \quad \uparrow
\]

\[
\text{bleu Rsrc1, Src2, label} \quad \text{Branch on LTE Unsigned} \quad \uparrow
\]

Conditionally branch to the instruction at the label if the contents of Rs are less than or equal to 0:

\[
\text{blez Rs, label} \quad \text{Branch on Less Than Equal Zero}
\]

<table>
<thead>
<tr>
<th>6</th>
<th>Rs</th>
<th>0</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Conditionally branch to the instruction at the label if the contents of Rs are less than 0. Save the address of the next instruction in register 31:

\[
\text{bltzal Rs, label} \quad \text{Branch on Less Than And Link}
\]

<table>
<thead>
<tr>
<th>1</th>
<th>Rs</th>
<th>0x10</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>
Conditionally branch to the instruction at the label if the contents of register Rs1 are less than Src2:

\[
\text{blt Rs1, Src2, label} \quad \text{Branch on Less Than} \quad ^\dagger \\
\text{bltu Rs1, Src2, label} \quad \text{Branch on Less Than Unsigned} \quad ^\dagger
\]

Conditionally branch to the instruction at the label if the contents of Rs are less than 0:

\[
\text{bltz Rs, label} \quad \text{Branch on Less Than Zero}
\]

Conditionally branch to the instruction at the label if the contents of register Rs1 are not equal to Src2:

\[
\text{bne Rs, Rt, label} \quad \text{Branch on Not Equal}
\]

Conditionally branch to the instruction at the label if the contents of Rsrc are not equal to 0:

\[
\text{bnez Rsrc, label} \quad \text{Branch on Not Equal Zero} \quad ^\dagger
\]

Unconditionally jump to the instruction at Target:

\[
\text{j label} \quad \text{Jump}
\]

Unconditionally jump to the instruction at Target. Save the address of the next instruction in register 31:

\[
\text{jal label} \quad \text{Jump and Link}
\]
Unconditionally jump to the instruction whose address is in register Rs. Save the address of the next instruction in register Rd (or in register 31, if Rd is omitted):

```
jalr [Rd,] Rs
```

**Jump and Link Register**

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th></th>
<th>Rd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unconditionally jump to the instruction whose address is in register Rs:

```
jr Rs
```

**Jump Register**

<table>
<thead>
<tr>
<th></th>
<th>Rs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**Load Instructions**

Load computed address, not the contents of the location, into register Rdest:

```
la Rdest, address
```

**Load Address**

Load the byte at address (or at Offset + contents of register Base) into register Rt. The byte is sign-extended by the lb, but not the lbu, instruction:

```
lb Rt, address|Offset(Base)
```

**Load Byte**

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
lbu Rt, address|Offset(Base)
```

**Load Unsigned Byte**

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x24</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load the 64-bit quantity at address into registers Rdest and Rdest + 1:

```
ld Rdest, address
```

**Load Double-Word**
Load the 16-bit quantity (halfword) at address (or at Offset + contents of register Base) into register Rt. The halfword is sign-extended by the lh, but not the lhu, instruction:

\[
\text{lh Rt, address|Offset(Base)} \quad \text{Load Halfword}
\]

\[
\begin{array}{cccc}
0x21 & \text{Base} & \text{Rt} & \text{Offset} \\
6 & 5 & 5 & 16 \\
\end{array}
\]

\[
\text{lhu Rt, address|Offset(Base)} \quad \text{Load Unsigned Halfword}
\]

\[
\begin{array}{cccc}
0x25 & \text{Base} & \text{Rt} & \text{Offset} \\
6 & 5 & 5 & 16 \\
\end{array}
\]

Load the 16-bit immediate into the most significant 16 bits of register Rt:

\[
\text{lui Rt, Imm} \quad \text{Load Upper Immediate}
\]

\[
\begin{array}{cccc}
15 & 0 & \text{Rt} & \text{Imm} \\
6 & 5 & 5 & 16 \\
\end{array}
\]

Load the 32-bit quantity (word) at address (or at Offset + contents of register Base) into register Rt:

\[
\text{lw Rt, address|Offset(Base)} \quad \text{Load Word}
\]

\[
\begin{array}{cccc}
0x23 & \text{Base} & \text{Rt} & \text{Offset} \\
6 & 5 & 5 & 16 \\
\end{array}
\]

Load the word at address (or at Offset + contents of register Base) into register Rt of coprocessor z (0–3):

\[
\text{lwc}z \text{ Rt, address|Offset(Base)} \quad \text{Load Word Coprocessor}
\]

\[
\begin{array}{cccc}
0x3z & \text{Base} & \text{Rt} & \text{Offset} \\
6 & 5 & 5 & 16 \\
\end{array}
\]

Load the left (right) bytes from the word at the possibly-unaligned address into register Rdest:

\[
\text{lw}l \text{ Rdest, address} \quad \text{Load Word Left}
\]

\[
\begin{array}{cccc}
0x22 & \text{Rs} & \text{Rt} & \text{Offset} \\
6 & 5 & 5 & 16 \\
\end{array}
\]
lwr Rdest, address

<table>
<thead>
<tr>
<th>0x23</th>
<th>Rs</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Load Word Right

Load the 16-bit quantity (halfword) at the possibly-unaligned address into register Rdest. The halfword is sign-extended by the ulh, but not the ulhu, instruction:

ulh Rdest, address

Unaligned Load Halfword†

ulhu Rdest, address

Unaligned Load Halfword Unsigned†

Load the 32-bit quantity (word) at the possibly-unaligned address into register Rdest:

ulw Rdest, address

Unaligned Load Word†

Store Instructions

Store the low byte from register Rt at address:

sb Rt, address

<table>
<thead>
<tr>
<th>0x28</th>
<th>Rs</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Store Byte

Store the 64-bit quantity in registers Rsrc and Rsrc + 1 at address:

sd Rsrc, address

Store Double-Word†

Store the low halfword from register Rt at address:

sh Rt, address

<table>
<thead>
<tr>
<th>0x29</th>
<th>Rs</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Store Halfword
Store the word from register $R_t$ at $address$:

$$\text{sw } R_t, \text{ address}$$  \hspace{1cm} \text{Store Word}

<table>
<thead>
<tr>
<th>0x2b</th>
<th>Rs</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Store the word from register $R_t$ of coprocessor $z$ at $address$:

$$\text{swc } z, R_t, \text{ address}$$  \hspace{1cm} \text{Store Word Coprocessor}

<table>
<thead>
<tr>
<th>0x3(1-z)</th>
<th>Rs</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Store the left (right) bytes from register $R_t$ at the possibly-unaligned $address$:

$$\text{swl } R_t, \text{ address}$$  \hspace{1cm} \text{Store Word Left}

<table>
<thead>
<tr>
<th>0x2a</th>
<th>Rs</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

$$\text{swr } R_t, \text{ address}$$  \hspace{1cm} \text{Store Word Right}

<table>
<thead>
<tr>
<th>0x2e</th>
<th>Rs</th>
<th>Rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Store the low halfword from register $R_{src}$ at the possibly-unaligned $address$:

$$\text{ush } R_{src}, \text{ address}$$  \hspace{1cm} \text{Unaligned Store Halfword} \dagger

Store the word from register $R_{src}$ at the possibly-unaligned $address$:

$$\text{usw } R_{src}, \text{ address}$$  \hspace{1cm} \text{Unaligned Store Word} \dagger

Data Movement Instructions

Move the contents of $R_{src}$ to $R_{dest}$:

$$\text{move } R_{dest}, R_{src}$$  \hspace{1cm} \text{Move} \dagger
The multiply and divide unit produces its result in two additional registers, hi and lo. The following instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudoinstructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.

Move the contents of the hi (lo) register to register Rd:

`mfhi Rd`  

\[ \begin{array}{cccc}
  0 & 0 & \text{Rd} & 0 \\
  6 & 10 & 5 & 5 & 6 \\
\end{array} \]

`mflo Rd`  

\[ \begin{array}{cccc}
  0 & 0 & \text{Rd} & 0 \\
  6 & 10 & 5 & 5 & 6 \\
\end{array} \]

Move the contents of register Rs to the hi (lo) register:

`mthi Rs`  

\[ \begin{array}{cccc}
  0 & \text{Rs} & 0 & 0 \\
  6 & 5 & 15 & 6 \\
\end{array} \]

`mtlo Rs`  

\[ \begin{array}{cccc}
  0 & \text{Rs} & 0 & 0 \\
  6 & 5 & 15 & 6 \\
\end{array} \]

Coprocessors have their own register sets. The following instructions move values between these registers and the CPU’s registers.

Move the contents of coprocessor z’s register Rd to CPU register Rt:

`mfcz Rt, Rd`  

\[ \begin{array}{cccc}
  0x1z & 0 & \text{Rt} & \text{Rd} & 0 \\
  6 & 5 & 5 & 5 & 11 \\
\end{array} \]
Move the contents of floating point registers \( FR_{src1} \) and \( FR_{src1} + 1 \) to CPU registers \( R_{dest} \) and \( R_{dest} + 1 \):

\[
mfc1.d \ R_{dest}, \ FR_{src1}
\]

*Move Double From Coprocessor 1†*

Move the contents of CPU register \( Rt \) to coprocessor \( z \)'s register \( Rd \):

\[
mtc: \ Rt, \ Rd
\]

*Move To Coprocessor \( z \)*

<table>
<thead>
<tr>
<th>0x11</th>
<th>4</th>
<th>( Rt )</th>
<th>( Rd )</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

**Floating Point Instructions**

The MIPS has a floating point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating point numbers. This coprocessor has its own registers, which are numbered \( \$f0 \)–\( \$f31 \). Because these registers are only 32-bits wide, two of them are required to hold doubles. To simplify matters, floating point operations only use even-numbered registers—including instructions that operate on single floats.

Values are moved in or out of these registers one word (32-bits) at a time by the \( lwc1, \ swc1, \ mtc1, \) and \( mfc1 \) instructions described above or by the \( l.s, \ l.d, \ s.s, \) and \( s.d \) pseudoinstructions described below. The flag set by floating point comparison operations is read by the CPU with its \( bc1t \) and \( bc1f \) instructions.

In the real instructions below, \( Fs \) and \( Fd \) are floating-point registers. In the pseudoinstructions, \( FR_{dest}, \ FR_{src1}, \ FR_{src2}, \) and \( FR_{src} \) are floating point registers (e.g., \( \$f2 \)).

Compute the absolute value of the floating float double (single) in register \( Fs \) and put it in register \( Fd \):

\[
\text{abs.d} \ Fd, \ Fs
\]

*Floating Point Absolute Value Double*

\[
\begin{array}{|c|c|c|c|c|}
\hline
0x11 & 1 & 0 & \( Fs \) & \( Fd \) & 5 \\
\hline
6 & 5 & 5 & 5 & 6 & \\
\hline
\end{array}
\]

\[
\text{abs.s} \ Fd, \ Fs
\]

*Floating Point Absolute Value Single*

\[
\begin{array}{|c|c|c|c|c|}
\hline
0x11 & 0 & 0 & \( Fs \) & \( Fd \) & 5 \\
\hline
6 & 5 & 5 & 5 & 6 & \\
\hline
\end{array}
\]
Compute the sum of the floating float doubles (singles) in registers $F_s$ and $F_t$ and put it in register $F_d$:

```
add.d F_d, F_s, F_t
```

Floating Point Addition Double

```
<table>
<thead>
<tr>
<th>0x11</th>
<th>1</th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
add.s F_d, F_s, F_t
```

Floating Point Addition Single

```
<table>
<thead>
<tr>
<th>0x11</th>
<th>0</th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

Compare the floating point double in register $F_s$ against the one in $F_t$ and set the floating point condition flag FC true if they are equal:

```
c.eq.d F_s, F_t
```

Compare Equal Double

```
<table>
<thead>
<tr>
<th>0x11</th>
<th>1</th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>FC</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
c.eq.s F_s, F_t
```

Compare Equal Single

```
<table>
<thead>
<tr>
<th>0x11</th>
<th>0</th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>FC</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Compare the floating point double in register $F_s$ against the one in $F_t$ and set the floating point condition flag true if the first is less than or equal to the second:

```
c.le.d F_s, F_t
```

Compare Less Than Equal Double

```
<table>
<thead>
<tr>
<th>0x11</th>
<th>1</th>
<th>Ft</th>
<th>Fs</th>
<th>0</th>
<th>FC</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
```

```
c.le.s F_s, F_t
```

Compare Less Than Equal Single

```
<table>
<thead>
<tr>
<th>0x11</th>
<th>0</th>
<th>Ft</th>
<th>Fs</th>
<th>0</th>
<th>FC</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
```

17
Compare the floating point double in register $F_s$ against the one in $F_t$ and set the condition flag true if the first is less than the second:

$$c.l.t.d \ Fs, \ Ft$$

**Compare Less Than Double**

```
0x11  1    Ft  Fs  0    FC  0xc  6  5  5  5  5  2  4
```

$$c.l.t.s \ Fs, \ Ft$$

**Compare Less Than Single**

```
0x11  0    Ft  Fs  0    FC  0xc  6  5  5  5  5  2  4
```

Convert the single precision floating point number or integer in register $F_s$ to a double precision number and put it in register $F_d$:

$$cvt.d.s \ Fd, \ Fs$$

**Convert Single to Double**

```
0x11  1  0    Fs  Fd  0x21  6  5  5  5  5  6
```

$$cvt.d.w \ Fd, \ Fs$$

**Convert Integer to Double**

```
0x11  0  0    Fs  Fd  0x21  6  5  5  5  5  6
```

Convert the double precision floating point number or integer in register $F_s$ to a single precision number and put it in register $F_d$:

$$cvt.s.d \ Fd, \ Fs$$

**Convert Double to Single**

```
0x11  1  0    Fs  Fd  0x20  6  5  5  5  5  6
```

$$cvt.s.w \ Fd, \ Fs$$

**Convert Integer to Single**

```
0x11  0  0    Fs  Fd  0x20  6  5  5  5  5  6
```
Convert the double or single precision floating point number in register Fs to an integer and put it in register Fd:

\[
\text{cvt.w.d Fd, Fs} \quad \text{Convert Double to Integer}
\]

\[
\begin{array}{c|c|c|c|c|c}
0x11 & 1 & 0 & Fs & Fd & 0x24 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

\[
\text{cvt.w.s Fd, Fs} \quad \text{Convert Single to Integer}
\]

\[
\begin{array}{c|c|c|c|c|c}
0x11 & 0 & 0 & Fs & Fd & 0x24 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Compute the quotient of the floating float doubles (singles) in registers Fs and Ft and put it in register Fd:

\[
\text{div.d Fd, Fs, Ft} \quad \text{Floating Point Divide Double}
\]

\[
\begin{array}{c|c|c|c|c|c}
0x11 & 1 & Ft & Fs & Fd & 3 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

\[
\text{div.s Fd, Fs, Ft} \quad \text{Floating Point Divide Single}
\]

\[
\begin{array}{c|c|c|c|c|c}
0x11 & 0 & Ft & Fs & Fd & 3 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Load the floating float double (single) at address into register FRdest:

\[
\text{l.d FRdest, address} \quad \text{Load Floating Point Double}^†
\]

\[
\text{l.s FRdest, address} \quad \text{Load Floating Point Single}^†
\]

Move the floating float double (single) from register Fs to register Fd:

\[
\text{mov.d Fd, Fs} \quad \text{Move Floating Point Double}
\]

\[
\begin{array}{c|c|c|c|c|c}
0x11 & 1 & 0 & Fs & Fd & 6 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

\[
\text{mov.s Fd, Fs} \quad \text{Move Floating Point Single}
\]

\[
\begin{array}{c|c|c|c|c|c}
0x11 & 0 & 0 & Fs & Fd & 6 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]
Compute the product of the floating float doubles (singles) in registers Fs and Ft and put it in register Fd:

\[
\text{mul.d Fd, Fs, Ft} \quad \text{Floating Point Multiply Double}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

\[
\text{mul.s Fd, Fs, Ft} \quad \text{Floating Point Multiply Single}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Negate the floating point double (single) in register Fs and put it in register Fd:

\[
\text{neg.d Fd, Fs} \quad \text{Negate Double}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Fs</th>
<th>Fd</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

\[
\text{neg.s Fd, Fs} \quad \text{Negate Single}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Fs</th>
<th>Fd</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Store the floating float double (single) in register FR\text{dest} at address: Store the floating float double (single) in register FR\text{dest} at address:

\[
\text{s.d FR\text{dest}, address} \quad \text{Store Floating Point Double}^†
\]

\[
\text{s.s FR\text{dest}, address} \quad \text{Store Floating Point Single}^†
\]

Compute the difference of the floating float doubles (singles) in registers Fs and Ft and put it in register Fd:

\[
\text{sub.d Fd, Fs, Ft} \quad \text{Floating Point Subtract Double}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

\[
\text{sub.s Fd, Fs, Ft} \quad \text{Floating Point Subtract Single}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Ft</th>
<th>Fs</th>
<th>Fd</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Exception and Trap Instructions

Restore the Status register:

\[ \text{rfe} \quad \text{Return From Exception} \]

<table>
<thead>
<tr>
<th>0x11</th>
<th>1</th>
<th>0</th>
<th>0x20</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>19</td>
<td>6</td>
</tr>
</tbody>
</table>

Register $v0 contains the number of the system call (see Table ??) provided by SPIM:

\[ \text{syscall} \quad \text{System Call} \]

<table>
<thead>
<tr>
<th>0x11</th>
<th>0</th>
<th>0xc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

Cause exception \( n \). Exception 1 is reserved for the debugger:

\[ \text{break } n \quad \text{Break} \]

<table>
<thead>
<tr>
<th>0x11</th>
<th>code</th>
<th>0xd</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

Do nothing:

\[ \text{nop} \quad \text{No operation} \]

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>