FIGURE B.10.2 MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10, and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for six op values: 0, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses “f” to mean “s” if rs = 16 and op = 17 or “d” if rs = 17 and op = 17. The second field (rs) uses “z” to mean “0”, “1”, “2”, or “3” if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if z = 0, the operations are specified in the fourth field (bits 4 to 0); if z = 1, then the operations are in the last field with f = s. If rs = 17 and z = 1, then the operations are in the last field with f = d.
Pseudoinstructions follow roughly the same conventions, but omit instruction encoding information. For example:

**Multiply (without overflow)**

```plaintext
mul rdest, rsrcl, src2  pseudoinstruction
```

In pseudoinstructions, \textit{rdest} and \textit{rsrcl} are registers and \textit{src2} is either a register or an immediate value. In general, the assembler and SPIM translate a more general form of an instruction (e.g., \texttt{add $v1, $a0, 0x55}) to a specialized form (e.g., \texttt{addi $v1, $a0, 0x55}).

### Arithmetic and Logical Instructions

**Absolute value**

```plaintext
abs rdest, rsrcl  pseudoinstruction
```

Put the absolute value of register \textit{rsrcl} in register \textit{rdest}.

**Addition (with overflow)**

```plaintext
add rd, rs, rt 0 rs rt rd 0 0x20
```

**Addition (without overflow)**

```plaintext
addu rd, rs, rt 0 rs rt rd 0 0x21
```

**Addition immediate (with overflow)**

```plaintext
addi rt, rs, imm 8 rs rt imm
```

**Addition immediate (without overflow)**

```plaintext
addiu rt, rs, imm 9 rs rt imm
```

Put the sum of register \textit{rs} and \textit{rt} into register \textit{rd}. Put the sum of register \textit{rs} and the sign-extended immediate into register \textit{rt}. 
AND

\[
\text{and } rd, rs, rt \quad 0 \quad rs \quad rt \quad rd \quad 0 \quad 0x24
\]

Put the logical AND of registers \( rs \) and \( rt \) into register \( rd \).

AND immediate

\[
\text{andi } rt, rs, \text{imm} \quad 0xc \quad rs \quad rt \quad \text{imm}
\]

Put the logical AND of register \( rs \) and the zero-extended immediate into register \( rt \).

Count leading ones

\[
\text{clo } rd, rs \quad 0x1c \quad rs \quad 0 \quad rd \quad 0 \quad 0x21
\]

Count leading zeros

\[
\text{clz } rd, rs \quad 0x1c \quad rs \quad 0 \quad rd \quad 0 \quad 0x20
\]

Count the number of leading ones (zeros) in the word in register \( rs \) and put the result into register \( rd \). If a word is all ones (zeros), the result is 32.

Divide (with overflow)

\[
\text{div } rs, rt \quad 0 \quad rs \quad rt \quad 0 \quad 0x1a
\]

Divide (without overflow)

\[
\text{divu } rs, rt \quad 0 \quad rs \quad rt \quad 0 \quad 0x1b
\]

Divide register \( rs \) by register \( rt \). Leave the quotient in register \( lo \) and the remainder in register \( hi \). Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.
Divide (with overflow)

\[
div \ rdest, \ rsrcl, \ src2 \quad \text{pseudoinstruction}
\]

Divide (without overflow)

\[
divu \ rdest, \ rsrcl, \ src2 \quad \text{pseudoinstruction}
\]

Put the quotient of register \( rsrcl \) and \( src2 \) into register \( rdest \).

Multiply

\[
\text{mult} \ rs, \ rt \quad \begin{array}{cccccc}
0 & rs & rt & 0 & \text{0x18} \\
6 & 5 & 5 & 10 & 6
\end{array}
\]

Unsigned multiply

\[
\text{multu} \ rs, \ rt \quad \begin{array}{cccccc}
0 & rs & rt & 0 & \text{0x19} \\
6 & 5 & 5 & 10 & 6
\end{array}
\]

Multiply registers \( rs \) and \( rt \). Leave the low-order word of the product in register \( lo \) and the high-order word in register \( hi \).

Multiply (without overflow)

\[
\text{mul} \ rd, \ rs, \ rt \quad \begin{array}{cccccc}
0x1c \quad rs & rt & rd & 0 & 2 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Put the low-order 32 bits of the product of \( rs \) and \( rt \) into register \( rd \).

Multiply (with overflow)

\[
mulo \ rdest, \ rsrcl, \ src2 \quad \text{pseudoinstruction}
\]

Unsigned multiply (with overflow)

\[
mulou \ rdest, \ rsrcl, \ src2 \quad \text{pseudoinstruction}
\]

Put the low-order 32 bits of the product of register \( rsrcl \) and \( src2 \) into register \( rdest \).
Multiply add
\[
\text{madd } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 0 \\
6 \quad 5 \quad 5 \quad 10 \quad 6
\]

Unsigned multiply add
\[
\text{maddu } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 1 \\
6 \quad 5 \quad 5 \quad 10 \quad 6
\]

Multiply registers \(rs\) and \(rt\) and add the resulting 64-bit product to the 64-bit value in the concatenated registers \(lo\) and \(hi\).

Multiply subtract
\[
\text{msub } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 4 \\
6 \quad 5 \quad 5 \quad 10 \quad 6
\]

Unsigned multiply subtract
\[
\text{msub } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 5 \\
6 \quad 5 \quad 5 \quad 10 \quad 6
\]

Multiply registers \(rs\) and \(rt\) and subtract the resulting 64-bit product from the 64-bit value in the concatenated registers \(lo\) and \(hi\).

Negate value (with overflow)
\[
\text{neg } rdest, src \quad \text{pseudoinstruction}
\]

Negate value (without overflow)
\[
\text{negu } rdest, src \quad \text{pseudoinstruction}
\]

Put the negative of register \(src\) into register \(rdest\).

NOR
\[
\text{nor } rd, rs, rt \quad 0 \quad rs \quad rt \quad rd \quad 0 \quad 0x27 \\
6 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6
\]

Put the logical NOR of registers \(rs\) and \(rt\) into register \(rd\).
NOT

\[ \text{not rdest, rs} \]

\textit{pseudoinstruction}

Put the bitwise logical negation of register \texttt{rs} into register \texttt{rdest}.

OR

\[ \text{or rd, rs, rt} \]

\begin{array}{c|cccc|c}
0 & rs & rt & rd & 0 & 0x25 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}

Put the logical OR of registers \texttt{rs} and \texttt{rt} into register \texttt{rd}.

\textbf{OR immediate}

\[ \text{ori rt, rs, imm} \]

\begin{array}{c|cccc}
Oxd & rs & rt & imm \\
6 & 5 & 5 & 16 \\
\end{array}

Put the logical OR of register \texttt{rs} and the zero-extended immediate into register \texttt{rt}.

\textbf{Remainder}

\[ \text{rem rdest, rsrcl, rsrcc} \]

\textit{pseudoinstruction}

\textbf{Unsigned remainder}

\[ \text{remu rdest, rsrcl, rsrcc} \]

\textit{pseudoinstruction}

Put the remainder of register \texttt{rsrcl} divided by register \texttt{rsrcc} into register \texttt{rdest}. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.

\textbf{Shift left logical}

\[ \text{sll rd, rt, shamt} \]

\begin{array}{c|cccc|c}
0 & rs & rt & rd & shamt & 0 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}

\textbf{Shift left logical variable}

\[ \text{sllv rd, rt, rs} \]

\begin{array}{c|cccc|c}
0 & rs & rt & rd & 0 & 4 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
Shift right arithmetic

\[ \text{sra rd, rt, shamt} \]

\[
\begin{array}{cccccc}
\text{0} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{3} \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Shift right arithmetic variable

\[ \text{srav rd, rt, rs} \]

\[
\begin{array}{cccccc}
\text{0} & \text{rs} & \text{rt} & \text{rd} & 0 & 7 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Shift right logical

\[ \text{srl rd, rt, shamt} \]

\[
\begin{array}{cccccc}
\text{0} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{2} \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Shift right logical variable

\[ \text{srlv rd, rt, rs} \]

\[
\begin{array}{cccccc}
\text{0} & \text{rs} & \text{rt} & \text{rd} & 0 & 6 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Shift register \( rt \) left (right) by the distance indicated by immediate \( \text{shamt} \) or the register \( rs \) and put the result in register \( rd \). Note that argument \( rs \) is ignored for \( \text{sll}, \text{sra}, \text{and srl} \).

Rotate left

\[ \text{rol rdest, rsrcl, rsrcl} \]

\[ \text{pseudoinstruction} \]

Rotate right

\[ \text{ror rdest, rsrcl, rsrcl} \]

\[ \text{pseudoinstruction} \]

Rotate register \( rsrcl \) left (right) by the distance indicated by \( rsrcl \) and put the result in register \( rdest \).

Subtract (with overflow)

\[ \text{sub rd, rs, rt} \]

\[
\begin{array}{cccccc}
\text{0} & \text{rs} & \text{rt} & \text{rd} & 0 & 0x22 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]
**Subtract (without overflow)**

```
subu rd, rs, rt 0 rs rt rd 0 0x23
6 5 5 5 5 6
```

Put the difference of registers `rs` and `rt` into register `rd`.

**Exclusive OR**

```
xor rd, rs, rt 0 rs rt rd 0 0x26
6 5 5 5 5 6
```

Put the logical XOR of registers `rs` and `rt` into register `rd`.

**XOR immediate**

```
xori rt, rs, imm 0xe rs rt imm 6 5 5 16
```

Put the logical XOR of register `rs` and the zero-extended immediate into register `rt`.

**Constant-Manipulating Instructions**

**Load upper immediate**

```
lui rt, imm 0xf 0 rt imm 6 5 5 16
```

Load the lower halfword of the immediate `imm` into the upper halfword of register `rt`. The lower bits of the register are set to 0.

**Load immediate**

```
li rdest, imm
```

Move the immediate `imm` into register `rdest`.

**Comparison Instructions**

**Set less than**

```
slt rd, rs, rt 0 rs rt rd 0 0x2a
6 5 5 5 5 6
```

B.10  MIPS R2000 Assembly Language  B-57
Set less than unsigned

\[
\text{sltu } rd, rs, rt
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>0x2b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Set register \( rd \) to 1 if register \( rs \) is less than \( rt \), and to 0 otherwise.

Set less than immediate

\[
\text{slti } rt, rs, \text{imm}
\]

<table>
<thead>
<tr>
<th></th>
<th>0xa</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Set less than unsigned immediate

\[
\text{sltiu } rt, rs, \text{imm}
\]

<table>
<thead>
<tr>
<th></th>
<th>0xb</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Set register \( rt \) to 1 if register \( rs \) is less than the sign-extended immediate, and to 0 otherwise.

Set equal

\[
\text{seq } \text{rdest}, \text{rsrcl}, \text{rsrcl}
\]

Set register \( \text{rdest} \) to 1 if register \( \text{rsrcl} \) equals \( \text{rsrcl} \), and to 0 otherwise.

Set greater than equal

\[
\text{sge } \text{rdest}, \text{rsrcl}, \text{rsrcl}
\]

Set greater than equal unsigned

\[
\text{sgeu } \text{rdest}, \text{rsrcl}, \text{rsrcl}
\]

Set register \( \text{rdest} \) to 1 if register \( \text{rsrcl} \) is greater than or equal to \( \text{rsrcl} \), and to 0 otherwise.

Set greater than

\[
\text{sgt } \text{rdest}, \text{rsrcl}, \text{rsrcl}
\]

Set equal
**Set greater than unsigned**

```
sgtu rdest, rsrcl, rsrcc
```

Set register `rdest` to 1 if register `rsrcl` is greater than `rsrcc`, and to 0 otherwise.

**Set less than equal**

```
sle rdest, rsrcl, rsrcc
```

**Set less than equal unsigned**

```
sleu rdest, rsrcl, rsrcc
```

Set register `rdest` to 1 if register `rsrcl` is less than or equal to `rsrcc`, and to 0 otherwise.

**Set not equal**

```
sne rdest, rsrcl, rsrcc
```

Set register `rdest` to 1 if register `rsrcl` is not equal to `rsrcc`, and to 0 otherwise.

**Branch Instructions**

Branch instructions use a signed 16-bit instruction offset field; hence, they can jump $2^{15} - 1$ instructions (not bytes) forward or $2^{15}$ instructions backward. The `jump` instruction contains a 26-bit address field. In actual MIPS processors, branch instructions are delayed branches, which do not transfer control until the instruction following the branch (its “delay slot”) has executed (see Chapter 4). Delayed branches affect the offset calculation, since it must be computed relative to the address of the delay slot instruction (PC + 4), which is when the branch occurs. SPIM does not simulate this delay slot, unless the `-bare` or `-delayed_branch` flags are specified.

In assembly code, offsets are not usually specified as numbers. Instead, an instructions branch to a label, and the assembler computes the distance between the branch and the target instructions.

In MIPS-32, all actual (not pseudo) conditional branch instructions have a “likely” variant (for example, `beq`’s likely variant is `beql`), which does not execute
the instruction in the branch’s delay slot if the branch is not taken. Do not use these instructions; they may be removed in subsequent versions of the architecture. SPIM implements these instructions, but they are not described further.

**Branch instruction**

\[
\text{b label}
\]

*pseudoinstruction*

Unconditionally branch to the instruction at the label.

**Branch coprocessor false**

\[
\text{bclf cc label}
\]

\[
\begin{array}{cccc}
0x11 & 8 & \text{cc} & 0 \\
6 & 5 & 3 & 2 & 16
\end{array}
\]

**Branch coprocessor true**

\[
\text{bclt cc label}
\]

\[
\begin{array}{cccc}
0x11 & 8 & \text{cc} & 1 \\
6 & 5 & 3 & 2 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if the floating-point coprocessor’s condition flag numbered \( \text{cc} \) is false (true). If \( \text{cc} \) is omitted from the instruction, condition code flag 0 is assumed.

**Branch on equal**

\[
\text{beq rs, rt, label}
\]

\[
\begin{array}{ccc}
4 & \text{rs} & \text{rt} \\
6 & 5 & 5 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if register \( \text{rs} \) equals \( \text{rt} \).

**Branch on greater than equal zero**

\[
\text{bgez rs, label}
\]

\[
\begin{array}{ccc}
1 & \text{rs} & 1 \\
6 & 5 & 5 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if register \( \text{rs} \) is greater than or equal to 0.
Branch on greater than equal zero and link

\[
\text{bgezal rs, label} \quad \begin{array}{ccc}
1 & rs & 0x11 \\
6 & 5 & 5 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if register \(rs\) is greater than or equal to 0. Save the address of the next instruction in register 31.

Branch on greater than zero

\[
\text{bgtz rs, label} \quad \begin{array}{ccc}
7 & rs & 0 \\
6 & 5 & 5 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if register \(rs\) is greater than 0.

Branch on less than equal zero

\[
\text{blez rs, label} \quad \begin{array}{ccc}
6 & rs & 0 \\
6 & 5 & 5 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if register \(rs\) is less than or equal to 0.

Branch on less than and link

\[
\text{bltzal rs, label} \quad \begin{array}{ccc}
1 & rs & 0x10 \\
6 & 5 & 5 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if register \(rs\) is less than 0. Save the address of the next instruction in register 31.

Branch on less than zero

\[
\text{bltz rs, label} \quad \begin{array}{ccc}
1 & rs & 0 \\
6 & 5 & 5 & 16
\end{array}
\]

Conditionally branch the number of instructions specified by the offset if register \(rs\) is less than 0.
Branch on not equal

\[
\text{bne rs, rt, label}
\]

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>rt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Conditionally branch the number of instructions specified by the offset if register \(rs\) is not equal to \(rt\).

Branch on equal zero

\[
\text{beqz rsr\(c\), label}
\]

Conditionally branch to the instruction at the label if \(rsr\(c\) equals 0.

Branch on greater than equal

\[
\text{bge rsr\(c\)l, rsr\(c\)2, label}
\]

Branch on greater than equal unsigned

\[
\text{bgeu rsr\(c\)l, rsr\(c\)2, label}
\]

Conditionally branch to the instruction at the label if register \(rsr\(c\)l is greater than or equal to \(rsr\(c\)2.

Branch on greater than

\[
\text{bgt rsr\(c\)l, src2, label}
\]

Branch on greater than unsigned

\[
\text{bgtu rsr\(c\)l, src2, label}
\]

Conditionally branch to the instruction at the label if register \(rsr\(c\)l is greater than src2.

Branch on less than equal

\[
\text{ble rsr\(c\)l, src2, label}
\]
Branch on less than equal unsigned

bleu rsrc1, src2, label 

Conditionally branch to the instruction at the label if register rsrc1 is less than or equal to src2.

Branch on less than

blt rsrcl, rsrc2, label 

Branch on less than unsigned

bltu rsrc1, rsrc2, label 

Conditionally branch to the instruction at the label if register rsrc1 is less than rsrc2.

Branch on not equal zero

bnez rsrcl, label 

Conditionally branch to the instruction at the label if register rsrcl is not equal to 0.

Jump Instructions

Jump

j target

Unconditionally jump to the instruction at target.

Jump and link

jal target

Unconditionally jump to the instruction at target. Save the address of the next instruction in register $ra.
Jump and link register

\[
\text{jalr } rs, rd \\
0 \quad rs \quad 0 \quad rd \quad 0 \quad 9
\]

Unconditionally jump to the instruction whose address is in register \(rs\). Save the address of the next instruction in register \(rd\) (which defaults to 31).

Jump register

\[
\text{jr } rs \\
0 \quad rs \quad 0 \quad 8
\]

Unconditionally jump to the instruction whose address is in register \(rs\).

**Trap Instructions**

**Trap if equal**

\[
\text{teq } rs, rt \\
0 \quad rs \quad rt \quad 0 \quad 0x34
\]

If register \(rs\) is equal to register \(rt\), raise a Trap exception.

**Trap if equal immediate**

\[
\text{teqi } rs, \text{ imm} \\
1 \quad rs \quad \text{Oxc} \quad \text{imm}
\]

If register \(rs\) is equal to the sign-extended value \(\text{imm}\), raise a Trap exception.

**Trap if not equal**

\[
\text{teq } rs, rt \\
0 \quad rs \quad rt \quad 0 \quad 0x36
\]

If register \(rs\) is not equal to register \(rt\), raise a Trap exception.

**Trap if not equal immediate**

\[
\text{teqi } rs, \text{ imm} \\
1 \quad rs \quad \text{Oxe} \quad \text{imm}
\]

If register \(rs\) is not equal to the sign-extended value \(\text{imm}\), raise a Trap exception.
Trap if greater equal

\[
tge \ rs, \ rt \\
0 \ rs \ rt \ 0 \ 0x30
\]

Unsigned trap if greater equal

\[
tgeu \ rs, \ rt \\
0 \ rs \ rt \ 0 \ 0x31
\]

If register \( rs \) is greater than or equal to register \( rt \), raise a Trap exception.

Trap if greater equal immediate

\[
tgei \ rs, \ imm \\
1 \ rs \ 8 \ imm
\]

Unsigned trap if greater equal immediate

\[
tgeiu \ rs, \ imm \\
1 \ rs \ 9 \ imm
\]

If register \( rs \) is greater than or equal to the sign-extended value \( imm \), raise a Trap exception.

Trap if less than

\[
tlt \ rs, \ rt \\
0 \ rs \ rt \ 0 \ 0x32
\]

Unsigned trap if less than

\[
tltu \ rs, \ rt \\
0 \ rs \ rt \ 0 \ 0x33
\]

If register \( rs \) is less than register \( rt \), raise a Trap exception.

Trap if less than immediate

\[
tlti \ rs, \ imm \\
1 \ rs \ a \ imm
\]
Unsigned trap if less than immediate

\[ \text{tltiu} \text{ rs, imm} \]

<table>
<thead>
<tr>
<th></th>
<th>rs</th>
<th>b</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

If register `rs` is less than the sign-extended value `imm`, raise a Trap exception.

**Load Instructions**

**Load address**

\[ \text{la rdest, address} \]

Load computed `address`—not the contents of the location—into register `rdest`.

**Load byte**

\[ \text{lb rt, address} \]

<table>
<thead>
<tr>
<th></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 unsigned byte**

\[ \text{lbu rt, address} \]

<table>
<thead>
<tr>
<th></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 the byte at `address` into register `rt`. The byte is sign-extended by `lb`, but not by `lbu`.

**Load halfword**

\[ \text{lh rt, address} \]

<table>
<thead>
<tr>
<th></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 unsigned halfword**

\[ \text{lhu rt, address} \]

<table>
<thead>
<tr>
<th></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 the 16-bit quantity (halfword) at `address` into register `rt`. The halfword is sign-extended by `lh`, but not by `lhu`. 
Load word

\[ \text{lw rt. address} \quad 0x23 \quad rs \quad rt \quad \text{Offset} \]

Load the 32-bit quantity (word) at \textit{address} into register \textit{rt}.

Load word coprocessor 1

\[ \text{lwc1 ft. address} \quad 0x31 \quad rs \quad rt \quad \text{Offset} \]

Load the word at \textit{address} into register \textit{ft} in the floating-point unit.

Load word left

\[ \text{lwl rt. address} \quad 0x22 \quad rs \quad rt \quad \text{Offset} \]

Load word right

\[ \text{lwr rt. address} \quad 0x26 \quad rs \quad rt \quad \text{Offset} \]

Load doubleword

\[ \text{ld rdest, address} \quad \text{pseudoinstruction} \]

Load the 64-bit quantity at \textit{address} into registers \textit{rdest} and \textit{rdest+1}.

Unaligned load halfword

\[ \text{ulh rdest, address} \quad \text{pseudoinstruction} \]
Unaligned load halfword unsigned

\texttt{ulhu \ rdest, \ address} \hspace{1cm} \textit{pseudoinstruction}

Load the 16-bit quantity (halfword) at the possibly unaligned \textit{address} into register \textit{rdest}. The halfword is sign-extended by \texttt{ulh}, but not \texttt{ulhu}.

Unaligned load word

\texttt{ulw \ rdest, \ address} \hspace{1cm} \textit{pseudoinstruction}

Load the 32-bit quantity (word) at the possibly unaligned \textit{address} into register \textit{rdest}.

Load linked

\texttt{ll \ rt, \ address} \hspace{1cm} \begin{tabular}{|c|c|c|c|}
\hline
\texttt{0x30} & rs & rt & Offset \\
\hline
6 & 5 & 5 & 16 \\
\hline
\end{tabular}

Load the 32-bit quantity (word) at \textit{address} into register \textit{rt} and start an atomic read-modify-write operation. This operation is completed by a store conditional (sc) instruction, which will fail if another processor writes into the block containing the loaded word. Since SPIM does not simulate multiple processors, the store conditional operation always succeeds.

Store Instructions

Store byte

\texttt{sb \ rt, \ address} \hspace{1cm} \begin{tabular}{|c|c|c|c|}
\hline
\texttt{0x28} & rs & rt & Offset \\
\hline
6 & 5 & 5 & 16 \\
\hline
\end{tabular}

Store the low byte from register \textit{rt} at \textit{address}.

Store halfword

\texttt{sh \ rt, \ address} \hspace{1cm} \begin{tabular}{|c|c|c|c|}
\hline
\texttt{0x29} & rs & rt & Offset \\
\hline
6 & 5 & 5 & 16 \\
\hline
\end{tabular}

Store the low halfword from register \textit{rt} at \textit{address}.
**Store word**

\[
\text{sw rt, address}
\]

<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 \( rt \) at \( address \).

**Store word coprocessor 1**

\[
\text{swcl ft, address}
\]

<table>
<thead>
<tr>
<th>0x31</th>
<th>rs</th>
<th>ft</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 floating-point value in register \( ft \) of floating-point coprocessor at \( address \).

**Store double coprocessor 1**

\[
\text{sdcl ft, address}
\]

<table>
<thead>
<tr>
<th>0x3d</th>
<th>rs</th>
<th>ft</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 doubleword floating-point value in registers \( ft \) and \( ft + 1 \) of floating-point coprocessor at \( address \). Register \( ft \) must be even numbered.

**Store word left**

\[
\text{swl rt, address}
\]

<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>

**Store word right**

\[
\text{swr rt, address}
\]

<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 left (right) bytes from register \( rt \) at the possibly unaligned \( address \).

**Store doubleword**

\[
\text{sd rsr, address}
\]

\( \text{pseudoinstruction} \)

Store the 64-bit quantity in registers \( rsr \) and \( rsr + 1 \) at \( address \).
Unaligned store halfword

\texttt{ush rsr}, \texttt{address}
pseudoinstruction

Store the low halfword from register \texttt{rsr} at the possibly unaligned \texttt{address}.

Unaligned store word

\texttt{usw rsr}, \texttt{address}
pseudoinstruction

Store the word from register \texttt{rsr} at the possibly unaligned \texttt{address}.

Store conditional

\begin{center}
\texttt{sc rt, address} \\
0x38 | rs | rt | Offset
\end{center}

6 5 5 16

Store the 32-bit quantity (word) in register \texttt{rt} into memory at \texttt{address} and complete an atomic read-modify-write operation. If this atomic operation is successful, the memory word is modified and register \texttt{rt} is set to 1. If the atomic operation fails because another processor wrote to a location in the block containing the addressed word, this instruction does not modify memory and writes 0 into register \texttt{rt}. Since SPIM does not simulate multiple processors, the instruction always succeeds.

Data Movement Instructions

Move

\texttt{move rdest, rsr}
pseudoinstruction

Move register \texttt{rsr} to \texttt{rdest}.

Move from hi

\begin{center}
\texttt{mfhi rd} \\
0 | 0 | rd | 0 | 0x10
\end{center}

6 10 5 5 6


The multiply and divide unit produces its result in two additional registers, \texttt{hi} and \texttt{lo}. These instructions move values to and from these registers. The multiply, divide, and remainder pseudoinstructions that make this unit appear to operate on the general registers move the result after the computation finishes.

Move the \texttt{hi} (\texttt{lo}) register to register \texttt{rd}.

Move to \texttt{hi}

\begin{verbatim}
 mthi rs
 \end{verbatim}

Move to \texttt{lo}

\begin{verbatim}
 mtlo rs
 \end{verbatim}

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

Move register \texttt{rd} in a coprocessor (register \texttt{fs} in the FPU) to CPU register \texttt{rt}. The floating-point unit is coprocessor 1.
Move double from coprocessor 1

\[
\text{mfcl.d rdest, frsrl} \quad \text{pseudoinstruction}
\]

Move floating-point registers \(\text{frsrl}\) and \(\text{frsrl} + 1\) to CPU registers \(\text{rdest}\) and \(\text{rdest} + 1\).

Move to coprocessor 0

\[
\text{mtc0 rd, rt} \quad 0 \times 10 | 4 | rt | rd | 0 \\
6 | 5 | 5 | 5 | 11
\]

Move to coprocessor 1

\[
\text{mtc1 rd, fs} \quad 0 \times 11 | 4 | rt | fs | 0 \\
6 | 5 | 5 | 5 | 11
\]

Move CPU register \(rt\) to register \(rd\) in a coprocessor (register \(fs\) in the FPU).

Move conditional not zero

\[
\text{movn rd, rs, rt} \quad 0 | rs | rt | rd | \text{0xb} \\
6 | 5 | 5 | 5 | 11
\]

Move register \(rs\) to register \(rd\) if register \(rt\) is not 0.

Move conditional zero

\[
\text{movz rd, rs, rt} \quad 0 | rs | rt | rd | \text{0xa} \\
6 | 5 | 5 | 5 | 11
\]

Move register \(rs\) to register \(rd\) if register \(rt\) is 0.

Move conditional on FP false

\[
\text{movf rd, rs, cc} \quad 0 | rs | cc | 0 | rd | 0 | 1 \\
6 | 5 | 3 | 2 | 5 | 5 | 6
\]

Move CPU register \(rs\) to register \(rd\) if FPU condition code flag number \(cc\) is 0. If \(cc\) is omitted from the instruction, condition code flag 0 is assumed.
Move conditional on FP true

```
movt rd, rs, cc
```

<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>cc</th>
<th>1</th>
<th>rd</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Move CPU register rs to register rd if FPU condition code flag number cc is 1. If cc is omitted from the instruction, condition code bit 0 is assumed.

**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, so only floating-point registers with even numbers can hold double precision values. The floating-point coprocessor also has eight condition code (cc) flags, numbered 0–7, which are set by compare instructions and tested by branch (bclf or bclt) and conditional move instructions.

Values are moved in or out of these registers one word (32 bits) at a time by `lwcl`, `swcl`, `mtcl`, and `mfc1` instructions or one double (64 bits) at a time by `ldcl` and `sdcl`, described above, or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudoinstructions described below.

In the actual instructions below, bits 21–26 are 0 for single precision and 1 for double precision. In the pseudoinstructions below, fdest is a floating-point register (e.g., $f2$).

**Floating-point absolute value double**

```
abs.d fd, fs
```

```
0x11 | 1 | 0 | fs | fd | 5
```

**Floating-point absolute value single**

```
abs.s fd, fs
```

```
0x11 | 0 | 0 | fs | fd | 5
```

Compute the absolute value of the floating-point double (single) in register fs and put it in register fd.

**Floating-point addition double**

```
add.d fd, fs, ft
```

```
0x11 | 0x11 | ft | fs | fd | 0
```

B.10  MIPS R2000 Assembly Language  B-73
Floating-point addition single

```
add.s fd, fs, ft 0x11 0x10 ft fs fd 0
6 5 5 5 5 6
```

Compute the sum of the floating-point doubles (singles) in registers fs and ft and put it in register fd.

Floating-point ceiling to word

```
ceil.w.d fd, fs 0x11 0x11 0 fs fd 0xe
6 5 5 5 5 6
ceil.w.s fd, fs 0x11 0x10 0 fs fd 0xe
```

Compute the ceiling of the floating-point double (single) in register fs, convert to a 32-bit fixed-point value, and put the resulting word in register fd.

Compare equal double

```
c.eq.d cc fs, ft 0x11 0x11 ft fs cc 0 FC 2
6 5 5 5 3 2 2 4
```

Compare equal single

```
c.eq.s cc fs, ft 0x11 0x10 ft fs cc 0 FC 2
6 5 5 5 3 2 2 4
```

Compare the floating-point double (single) in register fs against the one in ft and set the floating-point condition flag cc to 1 if they are equal. If cc is omitted, condition code flag 0 is assumed.

Compare less than equal double

```
c.le.d cc fs, ft 0x11 0x11 ft fs cc 0 FC 0xe
6 5 5 5 3 2 2 4
```

Compare less than equal single

```
c.le.s cc fs, ft 0x11 0x10 ft fs cc 0 FC 0xe
6 5 5 5 3 2 2 4
```
Compare the floating-point double (single) in register $fs$ against the one in $ft$ and set the floating-point condition flag $cc$ to 1 if the first is less than or equal to the second. If $cc$ is omitted, condition code flag 0 is assumed.

**Compare less than double**

```
c.lt.d cc fs, ft
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>0x11</th>
<th>ft</th>
<th>fs</th>
<th>cc</th>
<th>O</th>
<th>FC</th>
<th>Oxc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**Compare less than single**

```
c.lt.s cc fs, ft
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>0x10</th>
<th>ft</th>
<th>fs</th>
<th>cc</th>
<th>O</th>
<th>FC</th>
<th>Oxc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Compare the floating-point double (single) in register $fs$ against the one in $ft$ and set the condition flag $cc$ to 1 if the first is less than the second. If $cc$ is omitted, condition code flag 0 is assumed.

**Convert single to double**

```
cvt.d.s fd, fs
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>0x10</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>Ox21</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>

**Convert integer to double**

```
cvt.d.w fd, fs
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>0x14</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>Ox21</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>

Convert the single precision floating-point number or integer in register $fs$ to a double (single) precision number and put it in register $fd$.

**Convert double to single**

```
cvt.s.d fd, fs
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>0x11</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>Ox20</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>

**Convert integer to single**

```
cvt.s.w fd, fs
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>0x14</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>Ox20</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>

Convert the double precision floating-point number or integer in register $fs$ to a single precision number and put it in register $fd$. 
Appendix B  Assemblers, Linkers, and the SPIM Simulator

Convert double to integer
\[
\text{cvt.w.d fd, fs} \quad \begin{array}{cccccc}
0x11 & 0x11 & 0 & \text{fs} & \text{fd} & 0x24 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Convert single to integer
\[
\text{cvt.w.s fd, fs} \quad \begin{array}{cccccc}
0x11 & 0x10 & 0 & \text{fs} & \text{fd} & 0x24 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Convert the double or single precision floating-point number in register \textit{fs} to an integer and put it in register \textit{fd}.

Floating-point divide double
\[
\text{div.d fd, fs, ft} \quad \begin{array}{cccccc}
0x11 & 0x11 & \text{ft} & \text{fs} & \text{fd} & 3 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Floating-point divide single
\[
\text{div.s fd, fs, ft} \quad \begin{array}{cccccc}
0x11 & 0x10 & \text{ft} & \text{fs} & \text{fd} & 3 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Compute the quotient of the floating-point doubles (singles) in registers \textit{fs} and \textit{ft} and put it in register \textit{fd}.

Floating-point floor to word
\[
\text{floor.w.d fd, fs} \quad \begin{array}{cccccc}
0x11 & 0x11 & 0 & \text{fs} & \text{fd} & \text{0xf} \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

\[
\text{floor.w.s fd, fs} \quad \begin{array}{cccccc}
0x11 & 0x10 & 0 & \text{fs} & \text{fd} & \text{0xf} \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Compute the floor of the floating-point double (single) in register \textit{fs} and put the resulting word in register \textit{fd}.

Load floating-point double
\[
\text{l.d fdest, address} \quad \text{pseudoinstruction}
\]
**Load floating-point single**

\[
l.s \quad \text{fdest}, \text{address} \quad \text{pseudoinstruction}
\]

Load the floating-point double (single) at \text{address} into register \text{fdest}.

**Move floating-point double**

\[
\text{mov.d} \quad \text{fd}, \text{fs} \quad \begin{array}{ccccccc}
0x11 & 0x11 & 0 & \text{fs} & \text{fd} & 6 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

**Move floating-point single**

\[
\text{mov.s} \quad \text{fd}, \text{fs} \quad \begin{array}{ccccccc}
0x11 & 0x10 & 0 & \text{fs} & \text{fd} & 6 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Move the floating-point double (single) from register \text{fs} to register \text{fd}.

**Move conditional floating-point double false**

\[
\text{movf.d} \quad \text{fd}, \text{fs}, \text{cc} \quad \begin{array}{ccccccc}
0x11 & 0x11 & \text{cc} & 0 & \text{fs} & \text{fd} & 0x11 \\
6 & 5 & 3 & 2 & 5 & 5 & 6
\end{array}
\]

**Move conditional floating-point single false**

\[
\text{movf.s} \quad \text{fd}, \text{fs}, \text{cc} \quad \begin{array}{ccccccc}
0x11 & 0x10 & \text{cc} & 0 & \text{fs} & \text{fd} & 0x11 \\
6 & 5 & 3 & 2 & 5 & 5 & 6
\end{array}
\]

Move the floating-point double (single) from register \text{fs} to register \text{fd} if condition code flag \text{cc} is 0. If \text{cc} is omitted, condition code flag 0 is assumed.

**Move conditional floating-point double true**

\[
\text{movt.d} \quad \text{fd}, \text{fs}, \text{cc} \quad \begin{array}{ccccccc}
0x11 & 0x11 & \text{cc} & 1 & \text{fs} & \text{fd} & 0x11 \\
6 & 5 & 3 & 2 & 5 & 5 & 6
\end{array}
\]

**Move conditional floating-point single true**

\[
\text{movt.s} \quad \text{fd}, \text{fs}, \text{cc} \quad \begin{array}{ccccccc}
0x11 & 0x10 & \text{cc} & 1 & \text{fs} & \text{fd} & 0x11 \\
6 & 5 & 3 & 2 & 5 & 5 & 6
\end{array}
\]
Move the floating-point double (single) from register \( fs \) to register \( fd \) if condition code flag \( cc \) is 1. If \( cc \) is omitted, condition code flag 0 is assumed.

**Move conditional floating-point double not zero**

\[
\text{movn.d fd, fs, rt} \quad 0\times11\quad 0\times11\quad rt\quad fs\quad fd\quad 0\times13
\]

**Move conditional floating-point single not zero**

\[
\text{movn.s fd, fs, rt} \quad 0\times11\quad 0\times10\quad rt\quad fs\quad fd\quad 0\times13
\]

Move the floating-point double (single) from register \( fs \) to register \( fd \) if processor register \( rt \) is not 0.

**Move conditional floating-point double zero**

\[
\text{movz.d fd, fs, rt} \quad 0\times11\quad 0\times11\quad rt\quad fs\quad fd\quad 0\times12
\]

**Move conditional floating-point single zero**

\[
\text{movz.s fd, fs, rt} \quad 0\times11\quad 0\times10\quad rt\quad fs\quad fd\quad 0\times12
\]

Move the floating-point double (single) from register \( fs \) to register \( fd \) if processor register \( rt \) is 0.

**Floating-point multiply double**

\[
\text{mul.d fd, fs, ft} \quad 0\times11\quad 0\times11\quad ft\quad fs\quad fd\quad 2
\]

**Floating-point multiply single**

\[
\text{mul.s fd, fs, ft} \quad 0\times11\quad 0\times10\quad ft\quad fs\quad fd\quad 2
\]

Compute the product of the floating-point doubles (singles) in registers \( fs \) and \( ft \) and put it in register \( fd \).

**Negate double**

\[
\text{neg.d fd, fs} \quad 0\times11\quad 0\times11\quad 0\quad fs\quad fd\quad 7
\]
Negate single

```
neg.s fd, fs
```

Negate the floating-point double (single) in register \( fs \) and put it in register \( fd \).

Floating-point round to word

```
round.w.d fd, fs
round.w.s fd, fs
```

Round the floating-point double (single) value in register \( fs \), convert to a 32-bit fixed-point value, and put the resulting word in register \( fd \).

Square root double

```
sqrt.d fd, fs
```

Compute the square root of the floating-point double (single) in register \( fs \) and put it in register \( fd \).

Square root single

```
sqrt.s fd, fs
```

Store floating-point double

```
s.d fdest, address
```

Store floating-point single

```
s.s fdest, address
```

Store the floating-point double (single) in register \( fdest \) at \( address \).

Floating-point subtract double

```
sub.d fd, fs, ft
```
Floating-point subtract single

```
sub.s fd, fs, ft
```

Compute the difference of the floating-point doubles (singles) in registers fs and ft and put it in register fd.

Floating-point truncate to word

```
trunc.w.d fd, fs
trunc.w.s fd, fs
```

Truncate the floating-point double (single) value in register fs, convert to a 32-bit fixed-point value, and put the resulting word in register fd.

**Exception and Interrupt Instructions**

**Exception return**

```
eret
```

Set the EXL bit in coprocessor 0’s Status register to 0 and return to the instruction pointed to by coprocessor 0’s EPC register.

**System call**

```
syscall
```

Register $v0 contains the number of the system call (see Figure B.9.1) provided by SPIM.

**Break**

```
break code
```

Cause exception code. Exception 1 is reserved for the debugger.

**No operation**

```
nop
```

Do nothing.