A.10 MIPS R2000 Assembly Language

Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, subsequent items are stored starting at address addr.

.text <addr>

.word w1,..., wn

Store the n 32-bit quantities in successive memory words.

SPIM does not distinguish various parts of the data segment (.data, .rdata, and .sdata).

Encoding MIPS Instructions

Figure A.10.2 explains how a MIPS instruction is encoded in a binary number. Each column contains instruction encodings for a field (a contiguous group of bits) from an instruction. The numbers at the left margin are values for a field. For example, the j opcode has a value of 2 in the opcode field. The text at the top of a column names a field and specifies which bits it occupies in an instruction. For example, the op field is contained in bits 26–31 of an instruction. This field encodes most instructions. However, some groups of instructions use additional fields to distinguish related instructions. For example, the different floating-point instructions are specified by bits 0–5. The arrows from the first column show which opcodes use these additional fields.

Instruction Format

The rest of this appendix describes both the instructions implemented by actual MIPS hardware and the pseudoinstructions provided by the MIPS assembler. The two types of instructions are easily distinguished. Actual instructions depict the fields in their binary representation. For example, in

Addition (with overflow)

<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>0x20</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>

the add instruction consists of six fields. Each field’s size in bits is the small number below the field. This instruction begins with 6 bits of 0s. Register specifiers begin with an r, so the next field is a 5-bit register specifier called rs. This is the same register that is the second argument in the symbolic assembly at the left of this line. Another common field is imm16, which is a 16-bit immediate number.
Appendix A  Assemblers, Linkers, and the SPIM Simulator

A-50

A-50

FIGURE A.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 6 op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses “f” to mean “0”, “1”, “2”, “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 = 1.

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 6 op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses “f” to mean “0”, “1”, “2”, “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 = 1.
Pseudoinstructions follow roughly the same conventions, but omit instruction encoding information. For example:

**Multiply (without overflow)**

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

In pseudoinstructions, rdest and rsrcl are registers and 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., `add $v1, $a0, 0x55`) to a specialized form (e.g., `addi $v1, $a0, 0x55`).

**Arithmetic and Logical Instructions**

**Absolute value**

\[ \text{abs } rdest, src \quad \text{pseudoinstruction} \]

Put the absolute value of register src in register rdest.

**Addition (with overflow)**

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

\[
\begin{array}{cccccc}
0 & rs & rt & rd & 0 & 0x20 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

**Addition (without overflow)**

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

\[
\begin{array}{cccccc}
0 & rs & rt & rd & 0 & 0x21 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Put the sum of registers rs and rt into register rd.

**Addition immediate (with overflow)**

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

\[
\begin{array}{cccc}
8 & rs & rt & imm \\
6 & 5 & 5 & 16
\end{array}
\]

**Addition immediate (without overflow)**

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

\[
\begin{array}{cccc}
9 & rs & rt & imm \\
6 & 5 & 5 & 16
\end{array}
\]

Put the sum of register rs and the sign-extended immediate into register rt.
AND

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

\[
\begin{array}{cccccc}
0 & rs & rt & rd & 0 & 0x24 \\
6 & 5 & 5 & 5 & 6
\end{array}
\]

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

AND immediate

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

\[
\begin{array}{cccc}
0xc & rs & rt & imm \\
6 & 5 & 5 & 16
\end{array}
\]

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

Count leading ones

\[ \text{clo rd, rs} \]

\[
\begin{array}{cccccc}
0xlc & rs & 0 & rd & 0 & 0x21 \\
6 & 5 & 5 & 5 & 6
\end{array}
\]

Count leading zeros

\[ \text{clz rd, rs} \]

\[
\begin{array}{cccccc}
0xlc & rs & 0 & rd & 0 & 0x20 \\
6 & 5 & 5 & 5 & 6
\end{array}
\]

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)

\[ \text{div rs, rt} \]

\[
\begin{array}{cccc}
0 & rs & rt & 0 & 0x1a \\
6 & 5 & 5 & 10 & 6
\end{array}
\]

Divide (without overflow)

\[ \text{divu rs, rt} \]

\[
\begin{array}{cccc}
0 & rs & rt & 0 & 0x1b \\
6 & 5 & 5 & 10 & 6
\end{array}
\]

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)

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

Divide (without overflow)

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

Put the quotient of register \texttt{rsrlc} and \texttt{src2} into register \texttt{rdest}.

Multiply

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

Unsigned multiply

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

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

Multiply (without overflow)

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

Put the low-order 32 bits of the product of \texttt{rs} and \texttt{rt} into register \texttt{rd}.

Multiply (with overflow)

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

Unsigned multiply (with overflow)

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

Put the low-order 32 bits of the product of register \texttt{rsrlc} and \texttt{src2} into register \texttt{rdest}. 
Multiply add

\[
\text{madd } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 0
\]

\[
\begin{array}{cccccc}
6 & 5 & 5 & 10 & 6 \\
\end{array}
\]

Unsigned multiply add

\[
\text{maddu } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 1
\]

\[
\begin{array}{cccccc}
6 & 5 & 5 & 10 & 6 \\
\end{array}
\]

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

\[
\text{msub } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 4
\]

\[
\begin{array}{cccccc}
6 & 5 & 5 & 10 & 6 \\
\end{array}
\]

Unsigned multiply subtract

\[
\text{msub } rs, rt \quad 0x1c \quad rs \quad rt \quad 0 \quad 5
\]

\[
\begin{array}{cccccc}
6 & 5 & 5 & 10 & 6 \\
\end{array}
\]

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)

\[
\text{neg } rdest, rsrc \quad \text{pseudoinstruction}
\]

Negate value (without overflow)

\[
\text{negu } rdest, rsrc \quad \text{pseudoinstruction}
\]

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

NOR

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

\[
\begin{array}{cccccc}
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

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

\[
\text{not } rdest, rsr \quad \text{pseudoinstruction}
\]

Put the bitwise logical negation of register \( rsr \) into register \( rdest \).

OR

\[
\text{or } rd, rs, rt \quad \begin{array}{cccccc}
0 & rs & rt & rd & 0 & 0x25 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

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

OR immediate

\[
\text{ori } rt, rs, imm \quad \begin{array}{cccc}
0xd & rs & rt & imm \\
6 & 5 & 5 & 16
\end{array}
\]

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

Remainder

\[
\text{rem } rdest, rsrcl, rsrcl \quad \text{pseudoinstruction}
\]

Unsigned remainder

\[
\text{remu } rdest, rsrcl, rsrcl \quad \text{pseudoinstruction}
\]

Put the remainder of register \( rsrcl \) divided by register \( rsrcl \) into register \( rdest \). 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.

Shift left logical

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

Shift left logical variable

\[
\text{sllv } rd, rt, rs \quad \begin{array}{cccccc}
0 & rs & rt & rd & 0 & 4 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]
Shift right arithmetic

\[
\text{sra rd, rt, shamt} \quad 0 \quad rs \quad rt \quad rd \quad \text{shamt} \quad 3
\]

Shift right arithmetic variable

\[
\text{srav rd, rt, rs} \quad 0 \quad rs \quad rt \quad rd \quad 0 \quad 7
\]

Shift right logical

\[
\text{srl rd, rt, shamt} \quad 0 \quad rs \quad rt \quad rd \quad \text{shamt} \quad 2
\]

Shift right logical variable

\[
\text{srlv rd, rt, rs} \quad 0 \quad rs \quad rt \quad rd \quad 0 \quad 6
\]

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, sra, and srl}.

Rotate left

\[
\text{rol rdest, rsrc1, rsrc2} \quad \text{pseudoinstruction}
\]

Rotate right

\[
\text{ror rdest, rsrc1, rsrc2} \quad \text{pseudoinstruction}
\]

Rotate register \(rsrc1\) left (right) by the distance indicated by \(rsrc2\) and put the result in register \(rdest\).

Subtract (with overflow)

\[
\text{sub rd, rs, rt} \quad 0 \quad rs \quad rt \quad rd \quad 0 \quad 0x22
\]
Subtract (without overflow)

\[
\text{subu rd, rs, rt} \quad \begin{array}{cccccc}
0 & rs & rt & rd & 0 & 0x23 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

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

Exclusive OR

\[
\text{xor rd, rs, rt} \quad \begin{array}{cccccc}
0 & rs & rt & rd & 0 & 0x26 \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

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

XOR immediate

\[
\text{xori rt, rs, imm} \quad \begin{array}{cccc}
0xe & rs & rt & imm \\
6 & 5 & 5 & 16
\end{array}
\]

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

Constant-Manipulating Instructions

Load upper immediate

\[
\text{lui rt, imm} \quad \begin{array}{cccc}
0xf & 0 & rt & imm \\
6 & 5 & 5 & 16
\end{array}
\]

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

\[
\text{li rdest, imm} \quad \text{pseudoinstruction}
\]

Move the immediate \(imm\) into register \(rdest\).

Comparison Instructions

Set less than

\[
\text{slt rd, rs, rt} \quad \begin{array}{cccccc}
0 & rs & rt & rd & 0 & 0x2a \\
6 & 5 & 5 & 5 & 5 & 6
\end{array}
\]
Set less than unsigned

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

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

Set less than immediate

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

Set less than unsigned immediate

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

Set equal

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

Set greater than equal

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

Set greater than equal unsigned

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

Set greater than

\[
\text{sgt } rdest, rsrcl, rsrcl
\]
Set greater than unsigned

\texttt{sgtu rdest, rsrc1, rsrc2} \textit{pseudoinstruction}

Set register \texttt{rdest} to 1 if register \texttt{rsrc1} is greater than \texttt{rsrc2}, and to 0 otherwise.

Set less than equal

\texttt{sle rdest, rsrc1, rsrc2} \textit{pseudoinstruction}

Set less than equal unsigned

\texttt{sleu rdest, rsrc1, rsrc2} \textit{pseudoinstruction}

Set register \texttt{rdest} to 1 if register \texttt{rsrc1} is less than or equal to \texttt{rsrc2}, and to 0 otherwise.

Set not equal

\texttt{sne rdest, rsrc1, rsrc2} \textit{pseudoinstruction}

Set register \texttt{rdest} to 1 if register \texttt{rsrc1} is not equal to \texttt{rsrc2}, 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 backwards. The \textit{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 6). 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 \texttt{-bare} or \texttt{-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 MIPS32, all actual (not pseudo) conditional branch instructions have a "likely" variant (for example, \texttt{beq}’s likely variant is \texttt{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**

```
b label               pseudoinstruction
```

Unconditionally branch to the instruction at the label.

**Branch coprocessor false**

```
bclf cc label
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>8</th>
<th>cc</th>
<th>0</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

**Branch coprocessor true**

```
bclt cc label
```

<table>
<thead>
<tr>
<th>0x11</th>
<th>8</th>
<th>cc</th>
<th>1</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

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

**Branch on equal**

```
beq rs, rt, label
```

<table>
<thead>
<tr>
<th>4</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>

Conditionally branch the number of instructions specified by the offset if
register `rs` equals `rt`.

**Branch on greater than equal zero**

```
bgez rs, label
```

<table>
<thead>
<tr>
<th>1</th>
<th>rs</th>
<th>1</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 the number of instructions specified by the offset if
register `rs` is greater than or equal to 0.
Branch on greater than equal zero and link

\[
\text{bgezal rs, label} \\
1 \quad rs \quad 0x11 \quad \text{Offset}
\]

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

\[
\text{bgtz rs, label} \\
7 \quad rs \quad 0 \quad \text{Offset}
\]

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

Branch on less than equal zero

\[
\text{blez rs, label} \\
6 \quad rs \quad 0 \quad \text{Offset}
\]

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

\[
\text{bltzal rs, label} \\
1 \quad rs \quad 0x10 \quad \text{Offset}
\]

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

\[
\text{bltz rs, label} \\
1 \quad rs \quad 0 \quad \text{Offset}
\]

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

\[ \text{bne } rs, rt, \text{ label} \quad 5 \quad rs \quad rt \quad \text{Offset} \]

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

Branch on equal zero

\[ \text{beqz } rsrc, \text{ label} \quad \text{pseudoinstruction} \]

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

Branch on greater than equal

\[ \text{bge } rsrc1, rsrc2, \text{ label} \quad \text{pseudoinstruction} \]

Branch on greater than equal unsigned

\[ \text{bgeu } rsrc1, rsrc2, \text{ label} \quad \text{pseudoinstruction} \]

Conditionally branch to the instruction at the label if register \( rsrc1 \) is greater than or equal to \( rsrc2 \).

Branch on greater than

\[ \text{bgt } rsrc1, src2, \text{ label} \quad \text{pseudoinstruction} \]

Branch on greater than unsigned

\[ \text{bgtu } rsrc1, src2, \text{ label} \quad \text{pseudoinstruction} \]

Conditionally branch to the instruction at the label if register \( rsrc1 \) is greater than \( src2 \).

Branch on less than equal

\[ \text{ble } rsrc1, src2, \text{ label} \quad \text{pseudoinstruction} \]
Branch on less than equal unsigned

bleu rsrc1, src2, label  
\textit{pseudoinstruction}

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

Branch on less than

blt rsrc1, rsrc2, label  \textit{pseudoinstruction}

Branch on less than unsigned

bltu rsrc1, rsrc2, label  \textit{pseudoinstruction}

Conditionally branch to the instruction at the label if register \textit{rsrc1} is less than \textit{rsrc2}.

Branch on not equal zero

bnez rsrc, label  \textit{pseudoinstruction}

Conditionally branch to the instruction at the label if register \textit{rsrc} is not equal to 0.

Jump Instructions

Jump

\textit{j} target

\begin{array}{c|c}
2 & \text{target} \\
6 & 26
\end{array}

Unconditionally jump to the instruction at target.

Jump and link

\textit{jal} target

\begin{array}{c|c}
3 & \text{target} \\
6 & 26
\end{array}

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

Jump and link register

\[
\text{jalr rs, rd} \\
0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 9 \\
6 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6
\]

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

\[
\text{jr rs} \\
0 \quad 0 \quad 0 \quad 8 \\
6 \quad 5 \quad 15 \quad 6
\]

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

Trap Instructions

Trap if equal

\[
\text{teq rs, rt} \\
0 \quad rs \quad rt \quad 0 \quad 0x34 \\
6 \quad 5 \quad 5 \quad 10 \quad 6
\]

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

Trap if equal immediate

\[
\text{teqi rs, imm} \\
1 \quad rs \quad 0xc \quad imm \\
6 \quad 5 \quad 5 \quad 16
\]

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

Trap if not equal

\[
\text{teq rs, rt} \\
0 \quad rs \quad rt \quad 0 \quad 0x36 \\
6 \quad 5 \quad 5 \quad 10 \quad 6
\]

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

Trap if not equal immediate

\[
\text{teqi rs, imm} \\
1 \quad rs \quad 0xe \quad imm \\
6 \quad 5 \quad 5 \quad 16
\]

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

\[
\text{tge rs, rt} \quad 0 \quad rs \quad rt \quad 0 \quad 0x30
\]

Unsigned trap if greater equal

\[
\text{tgeu rs, rt} \quad 0 \quad rs \quad rt \quad 0 \quad 0x31
\]

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

Trap if greater equal immediate

\[
\text{tgei rs, imm} \quad 1 \quad rs \quad 8 \quad \text{imm} \quad 0x32
\]

Unsigned trap if greater equal immediate

\[
\text{tgeiu rs, imm} \quad 1 \quad rs \quad 9 \quad \text{imm} \quad 0x33
\]

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

Trap if less than

\[
\text{tlt rs, rt} \quad 0 \quad rs \quad rt \quad 0 \quad 0x32
\]

Unsigned trap if less than

\[
\text{tltu rs, rt} \quad 0 \quad rs \quad rt \quad 0 \quad 0x33
\]

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

Trap if less than immediate

\[
\text{tlti rs, imm} \quad 1 \quad rs \quad a \quad \text{imm} \quad 0x34
\]
Unsigned trap if less than immediate

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

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

**Load Instructions**

**Load address**

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

Load computed \( \text{address} \)—not the contents of the location—into register \( rdest \).

**Load byte**

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

**Load unsigned byte**

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

Load the byte at \( \text{address} \) into register \( rt \). The byte is sign-extended by \( \text{lb} \), but not by \( \text{lbu} \).

**Load halfword**

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

**Load unsigned halfword**

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

Load the 16-bit quantity (halfword) at \( \text{address} \) into register \( rt \). The halfword is sign-extended by \( \text{lh} \), but not by \( \text{lhu} \).
Load word

\[
\text{lwan } rt, \text{ address } \quad \begin{array}{cccc}
0x23 & rs & rt & \text{Offset} \\
6 & 5 & 5 & 16
\end{array}
\]

Load the 32-bit quantity (word) at address into register rt.

Load word coprocessor 1

\[
\text{lwc1 } ft, \text{ address } \quad \begin{array}{cccc}
0x31 & rs & ft & \text{Offset} \\
6 & 5 & 5 & 16
\end{array}
\]

Load the word at address into register ft in the floating-point unit.

Load word left

\[
\text{lwl } rt, \text{ address } \quad \begin{array}{cccc}
0x22 & rs & rt & \text{Offset} \\
6 & 5 & 5 & 16
\end{array}
\]

Load word right

\[
\text{lwr } rt, \text{ address } \quad \begin{array}{cccc}
0x26 & rs & rt & \text{Offset} \\
6 & 5 & 5 & 16
\end{array}
\]

Load the left (right) bytes from the word at the possibly unaligned address into register rt.

Load doubleword

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

Load the 64-bit quantity at address into registers rdest and rdest + 1.

Unaligned load halfword

\[
\text{ulh } rdest, \text{ address } \quad \text{pseudoinstruction}
\]
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|c} 0x30 & rs & rt & Offset \hline 6 & 5 & 5 & 16 \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 (\texttt{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|c} 0x28 & rs & rt & Offset \hline 6 & 5 & 5 & 16 \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|c} 0x29 & rs & rt & Offset \hline 6 & 5 & 5 & 16 \end{tabular}

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

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

Store the word from register \( rt \) at \( \text{address} \).

Store word coprocessor 1

\[ \text{swc1 } ft, \text{ address} \]

Store the floating-point value in register \( ft \) of floating-point coprocessor at \( \text{address} \).

Store double coprocessor 1

\[ \text{sdc1 } ft, \text{ address} \]

Store the double word floating-point value in registers \( ft \) and \( ft + 1 \) of floating-point coprocessor at \( \text{address} \). Register \( ft \) must be even numbered.

Store word left

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

Store word right

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

Store the left (right) bytes from register \( rt \) at the possibly unaligned \( \text{address} \).

Store doubleword

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

pseudoinstruction

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

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

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

### Unaligned store word

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

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

### Store conditional

\begin{verbatim}
sc rt, address
\end{verbatim}

![Table 1: sc rt, address](image1.png)

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

\begin{verbatim}
mov rdest, rsr
\end{verbatim}

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

#### Move from hi

\begin{verbatim}
{m}fh\texttt{i} rd
\end{verbatim}

![Table 2: Move from hi](image2.png)
Move from lo

\[
mflo \ rd \\
\text{0 0 \ rd 0 0x12} \\
6 10 5 5 6
\]

The multiply and divide unit produces its result in two additional registers, \text{hi} and \text{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 \text{hi (lo)} register to register \text{rd}.

Move to \text{hi}

\[
mthi \ rs \\
\text{0 rs 0 0x11} \\
6 5 15 6
\]

Move to \text{lo}

\[
mtlo \ rs \\
\text{0 rs 0 0x13} \\
6 5 15 6
\]

Move register \text{rs} to the \text{hi (lo) register}.

Move from coprocessor 0

\[
mfc0 \ rt, \ rd \\
\text{0x10 0 rt rd 0} \\
6 5 5 5 11
\]

Move from coprocessor 1

\[
mfc1 \ rt, \ fs \\
\text{0x11 0 rt fs 0} \\
6 5 5 5 11
\]

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

Move register \text{rd} in a coprocessor (register \text{fs} in the FPU) to CPU register \text{rt}. The floating-point unit is coprocessor 1.
Appendix A  Assemblers, Linkers, and the SPIM Simulator

Move double from coprocessor 1

\[ \text{mfc1.d rdest, frsrc1} \]

* pseudoinstruction*

Move floating-point registers \( \text{frsrc1} \) and \( \text{frsrc1} + 1 \) to CPU registers \( \text{rdest} \) and \( \text{rdest} + 1 \).

Move to coprocessor 0

\[ \text{mtc0 rd, rt} \]

\[
\begin{array}{cccc}
0 & 10 & 4 & r_t & r_d & 0 \\
6 & 5 & 5 & 5 & 11 \\
\end{array}
\]

Move to coprocessor 1

\[ \text{mtc1 rd, fs} \]

\[
\begin{array}{cccc}
0 & 11 & 4 & r_t & f_s & 0 \\
6 & 5 & 5 & 5 & 11 \\
\end{array}
\]

Move CPU register \( r_t \) to register \( r_d \) in a coprocessor (register \( f_s \) in the FPU).

Move conditional not zero

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

\[
\begin{array}{cccc}
0 & r_s & r_t & r_d & 0xb \\
6 & 5 & 5 & 5 & 11 \\
\end{array}
\]

Move register \( r_s \) to register \( r_d \) if register \( r_t \) is not 0.

Move conditional zero

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

\[
\begin{array}{cccc}
0 & r_s & r_t & r_d & 0xa \\
6 & 5 & 5 & 5 & 11 \\
\end{array}
\]

Move register \( r_s \) to register \( r_d \) if register \( r_t \) is 0.

Move conditional on FP false

\[ \text{movf rd, rs, cc} \]

\[
\begin{array}{cccccc}
0 & r_s & c_c & 0 & r_d & 0 & 1 \\
6 & 5 & 3 & 2 & 5 & 5 & 6 \\
\end{array}
\]

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

\[
\text{movt rd, rs, cc}
\]

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 8 condition code (\( cc \)) flags, numbered 0–7, which are set by compare instructions and tested by branch (\( bclf \) or \( bclf \)) and conditional move instructions.

Values are moved in or out of these registers one word (32 bits) at a time by \( \text{lwc1, swc1, mtc1, and mfcl instructions} \) or one double (64 bits) at a time by \( \text{ldcl and sdc1 described above, or by the l.s, l.d, s.s, and s.d pseudoinstructions described below.} \)

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

\[
\text{abs.d fd, fs}
\]

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

Floating-point addition double

\[
\text{add.d fd, fs, ft}
\]
Floating-point addition single

\[
\text{add.s fd, fs, ft} \quad 0x11 \quad 0x10 \quad 0 \quad 5 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6
\]

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

\[
\text{ceil.w.d fd, fs} \quad 0x11 \quad 0x11 \quad 0 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6 \quad 6
\]

\[
\text{ceil.w.s fd, fs} \quad 0x11 \quad 0x10 \quad 0 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6 \quad 6
\]

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

\[
\text{c.eq.d cc fs, ft} \quad 0x11 \quad 0x11 \quad 0 \quad 5 \quad 5 \quad 5 \quad 3 \quad 2 \quad 2 \quad 4 \quad 2
\]

Compare equal single

\[
\text{c.eq.s cc fs, ft} \quad 0x11 \quad 0x10 \quad 0 \quad 5 \quad 5 \quad 5 \quad 3 \quad 2 \quad 2 \quad 4 \quad 2
\]

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

\[
\text{c.le.d cc fs, ft} \quad 0x11 \quad 0x11 \quad 0 \quad 5 \quad 5 \quad 5 \quad 0 \quad 2 \quad 2 \quad 4 \quad Oxe
\]

Compare less than equal single

\[
\text{c.le.s cc fs, ft} \quad 0x11 \quad 0x10 \quad 0 \quad 5 \quad 5 \quad 5 \quad 0 \quad 2 \quad 2 \quad 4 \quad Oxe
\]
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**

\[
c.\text{lt}.d \ cc \ fs, \ ft
\]

\[
0x11 \ 0x11 \ \text{ft} \ \text{fs} \ \cc \ 0 \ \text{FC} \ 0x0c
\]

**Compare less than single**

\[
c.\text{lt}.s \ cc \ fs, \ ft
\]

\[
0x11 \ 0x10 \ \text{ft} \ \text{fs} \ \cc \ 0 \ \text{FC} \ 0x0c
\]

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**

\[
c\text{vt.d.s fd, fs}
\]

\[
0x11 \ 0x10 \ 0 \ \text{fs} \ \text{fd} \ 0x21
\]

**Convert integer to double**

\[
c\text{vt.d.w fd, fs}
\]

\[
0x11 \ 0x14 \ 0 \ \text{fs} \ \text{fd} \ 0x21
\]

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**

\[
c\text{vt.s.d fd, fs}
\]

\[
0x11 \ 0x11 \ 0 \ \text{fs} \ \text{fd} \ 0x20
\]

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

Convert double to integer

\[ \text{cvt.w.d } fd, fs \ 0x11 \ 0x11 \ 0 \ fs \ fd \ 0x24 \]

Convert single to integer

\[ \text{cvt.w.s } fd, fs \ 0x11 \ 0x10 \ 0 \ fs \ fd \ 0x24 \]

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

\[ \text{div.d } fd, fs, ft \ 0x11 \ 0x11 \ ft \ fs \ fd \ 3 \]

Floating-point divide single

\[ \text{div.s } fd, fs, ft \ 0x11 \ 0x10 \ ft \ fs \ fd \ 3 \]

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

\[ \text{floor.w.d } fd, fs \ 0x11 \ 0x11 \ 0 \ fs \ fd \ 0xf \]

\[ \text{floor.w.s } fd, fs \ 0x11 \ 0x10 \ 0 \ fs \ fd \ 0xf \]

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

\[ \text{l.d } fdest, address \hspace{1cm} \text{pseudoinstruction} \]
**Load floating-point single**

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

Load the floating-point double (single) at `address` into register `fdest`.

**Move floating-point double**

\[ \text{mov.d fd, fs} \]

\[
\begin{array}{cccccc}
0 & 1 & 1 & 0 & 1 & 1 \\
5 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

**Move floating-point single**

\[ \text{mov.s fd, fs} \]

\[
\begin{array}{cccccc}
0 & 1 & 0 & 0 & 1 & 1 \\
5 & 5 & 5 & 5 & 5 & 6
\end{array}
\]

Move the floating-point double (single) from register `fs` to register `fd`.

**Move conditional floating-point double false**

\[ \text{movf.d fd, fs, cc} \]

\[
\begin{array}{ccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 \\
5 & 5 & 3 & 2 & 5 & 5 & 6
\end{array}
\]

**Move conditional floating-point single false**

\[ \text{movf.s fd, fs, cc} \]

\[
\begin{array}{ccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 0 \\
5 & 5 & 3 & 2 & 5 & 5 & 6
\end{array}
\]

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

**Move conditional floating-point double true**

\[ \text{movf.d fd, fs, cc} \]

\[
\begin{array}{ccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 1 \\
6 & 3 & 3 & 2 & 3 & 5 & 6
\end{array}
\]

**Move conditional floating-point single true**

\[ \text{movt.s fd, fs, cc} \]

\[
\begin{array}{ccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1 \\
6 & 3 & 3 & 2 & 3 & 5 & 6
\end{array}
\]
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

\[
\text{movn.d } fd, fs, rt \quad 0x11 \quad 0x11 \quad rt \quad fs \quad fd \quad 0x13
\]

Move conditional floating-point single not zero

\[
\text{movn.s } fd, fs, rt \quad 0x11 \quad 0x10 \quad rt \quad fs \quad fd \quad 0x13
\]

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

\[
\text{movz.d } fd, fs, rt \quad 0x11 \quad 0x11 \quad rt \quad fs \quad fd \quad 0x12
\]

Move conditional floating-point single zero

\[
\text{movz.s } fd, fs, rt \quad 0x11 \quad 0x10 \quad rt \quad fs \quad fd \quad 0x12
\]

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

Floating-point multiply double

\[
\text{mul.d } fd, fs, ft \quad 0x11 \quad 0x11 \quad ft \quad fs \quad fd \quad 2
\]

Floating-point multiply single

\[
\text{mul.s } fd, fs, ft \quad 0x11 \quad 0x10 \quad ft \quad fs \quad fd \quad 2
\]

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

```
fig probe 0x11 0x11 0f5 6
```

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

Floating-point round to word

```
fig probe 0x11 0x11 0f5 6 0xc
```

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

Square root double

```
fig probe 0x11 0x11 0f5 6
```

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

Store floating-point double

```
fig probe 0x11 0x10 0f5 6
```

\textit{pseudoinstruction}
Appendix A  Assemblers, Linkers, and the SPIM Simulator

Store floating-point single

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

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

Floating-point subtract double

\[
\begin{array}{ccccccc}
\text{sub.d } \text{fd, fs, ft} & 0x11 & 0x11 & \text{ft} & \text{fs} & \text{fd} & 1 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Floating-point subtract single

\[
\begin{array}{ccccccc}
\text{sub.s } \text{fd, fs, ft} & 0x11 & 0x10 & \text{ft} & \text{fs} & \text{fd} & 1 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

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

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

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

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

\[
\begin{array}{ccccccc}
\text{eret} & 0x10 & 0 & 0x18 \\
6 & 1 & 19 & 6 \\
\end{array}
\]

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.
A.11 Concluding Remarks

System call

syscall

\[
\begin{array}{c|c|c}
0 & 0 & 0xc \\
6 & 20 & 6 \\
\end{array}
\]

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

Break

\[
\begin{array}{c|c|c}
0 & \text{code} & 0xd \\
6 & 20 & 6 \\
\end{array}
\]

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

No operation

\[
\begin{array}{c|c|c|c|c|c}
0 & 0 & 0 & 0 & 0 & 0 \\
6 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

Do nothing.

A.11 Concluding Remarks

Programming in assembly language requires a programmer to trade off helpful features of high-level languages—such as data structures, type checking, and control constructs—for complete control over the instructions that a computer executes. External constraints on some applications, such as response time or program size, require a programmer to pay close attention to every instruction. However, the cost of this level of attention is assembly language programs that are longer, more time-consuming to write, and more difficult to maintain than high-level language programs.

Moreover, three trends are reducing the need to write programs in assembly language. The first trend is toward the improvement of compilers. Modern compilers produce code that is typically comparable to the best handwritten code—and is sometimes better. The second trend is the introduction of new processors that are not only faster, but in the case of processors that execute multiple instructions simultaneously, also more difficult to program by hand. In addition, the