

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, 1, 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)

```
mul rdest, rsrc1, src2    pseudoinstruction
```

In pseudoinstructions, `rdest` and `rsrc1` 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

```
abs rdest, rsrc    pseudoinstruction
```

Put the absolute value of register `rsrc` in register `rdest`.

Addition (with overflow)

```
add rd, rs, rt
```

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

Addition (without overflow)

```
addu rd, rs, rt
```

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

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

Addition immediate (with overflow)

```
addi rt, rs, imm
```

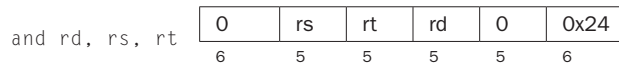
8	rs	rt	imm
6	5	5	16

Addition immediate (without overflow)

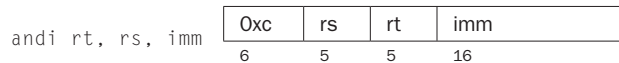
```
addiu rt, rs, imm
```

9	rs	rt	imm
6	5	5	16

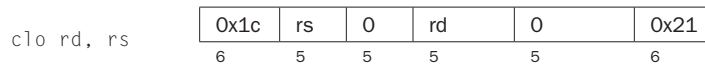
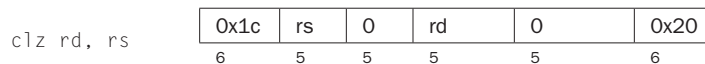
Put the sum of register `rs` and the sign-extended immediate into register `rt`.

AND

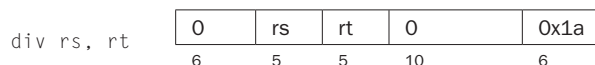
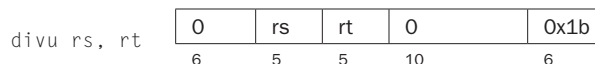
Put the logical AND of registers *rs* and *rt* into register *rd*.

AND immediate

Put the logical AND of register *rs* and the zero-extended immediate into register *rt*.

Count leading ones**Count leading zeros**

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)**Divide (without overflow)**

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    pseudoinstruction
```

Divide (without overflow)

```
divu rdest, rsrcl, src2  pseudoinstruction
```

Put the quotient of register `rsrcl` and `src2` into register `rdest`.

Multiply

```
mult rs, rt
```

0	rs	rt	0	0x18
6	5	5	10	6

Unsigned multiply

```
multu rs, rt
```

0	rs	rt	0	0x19
6	5	5	10	6

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)

```
mul rd, rs, rt
```

0x1c	rs	rt	rd	0	2
6	5	5	5	5	6

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

Multiply (with overflow)

```
mulo rdest, rsrcl, src2    pseudoinstruction
```

Unsigned multiply (with overflow)

```
mulou rdest, rsrcl, src2   pseudoinstruction
```

Put the low-order 32 bits of the product of register `rsrcl` and `src2` into register `rdest`.

Multiply add

madd rs, rt	0x1c	rs	rt	0	0
	6	5	5	10	6

Unsigned multiply add

maddu rs, rt	0x1c	rs	rt	0	1
	6	5	5	10	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

msub rs, rt	0x1c	rs	rt	0	4
	6	5	5	10	6

Unsigned multiply subtract

msubu rs, rt	0x1c	rs	rt	0	5
	6	5	5	10	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)

neg rdest, rsrc *pseudoinstruction*

Negate value (without overflow)

negu rdest, rsrc *pseudoinstruction*

Put the negative of register *rsrc* into register *rdest*.

NOR

nor rd, rs, rt	0	rs	rt	rd	0	0x27
	6	5	5	5	5	6

Put the logical NOR of registers *rs* and *rt* into register *rd*.

NOT

```
not rdest, rsrc           pseudoinstruction
```

Put the bitwise logical negation of register `rsrc` into register `rdest`.

OR

```
or rd, rs, rt
```

0	rs	rt	rd	0	0x25
6	5	5	5	5	6

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

OR immediate

```
ori rt, rs, imm
```

0xd	rs	rt	imm
6	5	5	16

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

Remainder

```
rem rdest, rsrc1, rsrc2   pseudoinstruction
```

Unsigned remainder

```
remu rdest, rsrc1, rsrc2 pseudoinstruction
```

Put the remainder of register `rsrc1` divided by register `rsrc2` 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

```
sll rd, rt, shamt
```

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

Shift left logical variable

```
sllv rd, rt, rs
```

0	rs	rt	rd	0	4
6	5	5	5	5	6

Shift right arithmetic

sra rd, rt, shamt

0	rs	rt	rd	shamt	3
6	5	5	5	5	6

Shift right arithmetic variable

srav rd, rt, rs

0	rs	rt	rd	0	7
6	5	5	5	5	6

Shift right logical

srl rd, rt, shamt

0	rs	rt	rd	shamt	2
6	5	5	5	5	6

Shift right logical variable

srlv rd, rt, rs

0	rs	rt	rd	0	6
6	5	5	5	5	6

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

Rotate left

rol rdest, rsrc1, rsrc2 *pseudoinstruction*

Rotate right

ror rdest, rsrc1, rsrc2 *pseudoinstruction*

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

Subtract (with overflow)

sub rd, rs, rt

0	rs	rt	rd	0	0x22
6	5	5	5	5	6

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
```

Oxe	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
```

Oxf	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

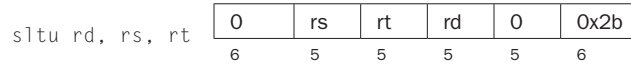
```
li rdest, imm pseudoinstruction
```

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

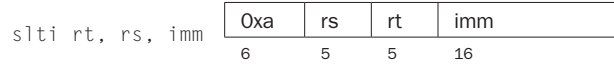
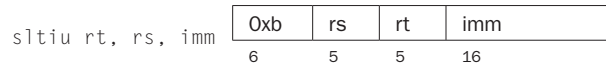
Comparison Instructions**Set less than**

```
slt rd, rs, rt
```

0	rs	rt	rd	0	0x2a
6	5	5	5	5	6

Set less than unsigned

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

Set less than immediate**Set less than unsigned immediate**

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

Set equal

seq rdest, rsrc1, rsrc2 *pseudoinstruction*

Set register rdest to 1 if register rsrc1 equals rsrc2, and to 0 otherwise.

Set greater than equal

sge rdest, rsrc1, rsrc2 *pseudoinstruction*

Set greater than equal unsigned

sgeu rdest, rsrc1, rsrc2 *pseudoinstruction*

Set register rdest to 1 if register rsrc1 is greater than or equal to rsrc2, and to 0 otherwise.

Set greater than

sgt rdest, rsrc1, rsrc2 *pseudoinstruction*

Set greater than unsigned

```
sgtu rdest, rsrc1, rsrc2      pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is greater than `rsrc2`, and to 0 otherwise.

Set less than equal

```
sle rdest, rsrc1, rsrc2      pseudoinstruction
```

Set less than equal unsigned

```
sleu rdest, rsrc1, rsrc2     pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is less than or equal to `rsrc2`, and to 0 otherwise.

Set not equal

```
sne rdest, rsrc1, rsrc2      pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is not equal to `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 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

b label *pseudoinstruction*

Unconditionally branch to the instruction at the label.

Branch coprocessor false

bclf cc label

0x11	8	cc	0	Offset
6	5	3	2	16

Branch coprocessor true

bclt cc label

0x11	8	cc	1	Offset
6	5	3	2	16

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

4	rs	rt	Offset
6	5	5	16

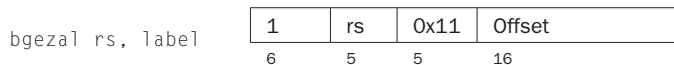
Conditionally branch the number of instructions specified by the offset if register *rs* equals *rt*.

Branch on greater than equal zero

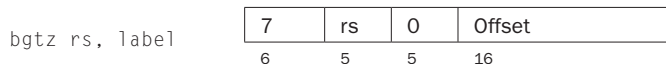
bgez rs, label

1	rs	1	Offset
6	5	5	16

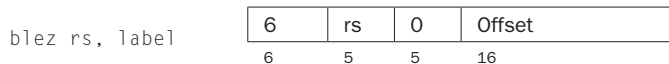
Conditionally branch the number of instructions specified by the offset if register *rs* is greater than or equal to 0.

Branch on greater than equal zero and link

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

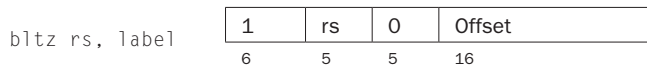
Conditionally branch the number of instructions specified by the offset if register `rs` is greater than 0.

Branch on less than equal zero

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

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

Conditionally branch the number of instructions specified by the offset if register `rs` is less than 0.

Branch on not equal

bne rs, rt, label	5	rs	rt	Offset
	6	5	5	16

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

Branch on equal zero

beqz rsrc, label *pseudoinstruction*

Conditionally branch to the instruction at the label if `rsrc` equals 0.

Branch on greater than equal

bge rsrc1, rsrc2, label *pseudoinstruction*

Branch on greater than equal unsigned

bgeu rsrc1, rsrc2, label *pseudoinstruction*

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

Branch on greater than

bgt rsrc1, src2, label *pseudoinstruction*

Branch on greater than unsigned

bgtu rsrc1, src2, label *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc1` is greater than `src2`.

Branch on less than equal

ble rsrc1, src2, label *pseudoinstruction*

Branch on less than equal unsigned

```
bleu rsrc1, src2, label      pseudoinstruction
```

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

Branch on less than

```
blt rsrc1, rsrc2, label     pseudoinstruction
```

Branch on less than unsigned

```
bltu rsrc1, rsrc2, label   pseudoinstruction
```

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

Branch on not equal zero

```
bnez rsrc, label          pseudoinstruction
```

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

Jump Instructions**Jump**

```
j target
```

2	target
6	26

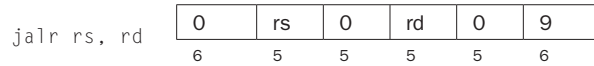
Unconditionally jump to the instruction at target.

Jump and link

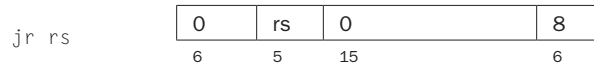
```
jal target
```

3	target
6	26

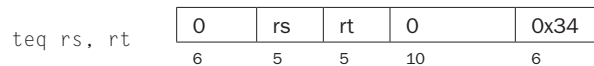
Unconditionally jump to the instruction at target. Save the address of the next instruction in register `$ra`.

Jump and link register

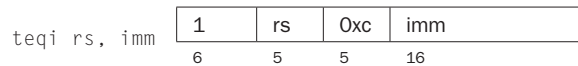
Unconditionally jump to the instruction whose address is in register `rs`. Save the address of the next instruction in register `rd` (which defaults to 31).

Jump register

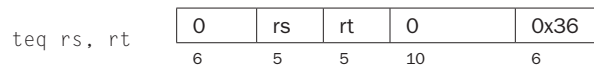
Unconditionally jump to the instruction whose address is in register `rs`.

Trap Instructions**Trap if equal**

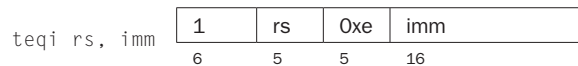
If register `rs` is equal to register `rt`, raise a Trap exception.

Trap if equal immediate

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

Trap if not equal

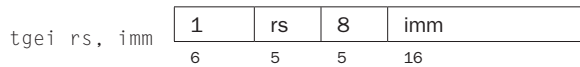
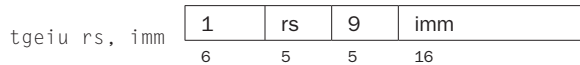
If register `rs` is not equal to register `rt`, raise a Trap exception.

Trap if not equal immediate

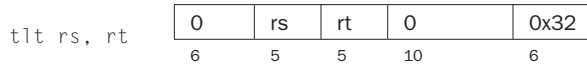
If register `rs` is not equal to the sign-extended value `imm`, raise a Trap exception.

Trap if greater equal**Unsigned trap if greater equal**

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

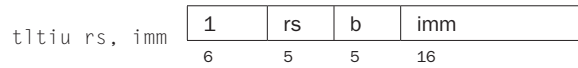
Trap if greater equal immediate**Unsigned trap if greater equal immediate**

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

Trap if less than**Unsigned trap if less than**

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

Trap if less than immediate

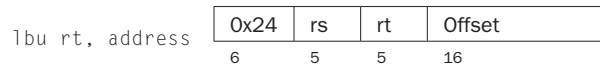
Unsigned trap if less than immediate

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

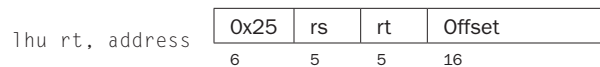
Load Instructions**Load address**

la rdest, address *pseudoinstruction*

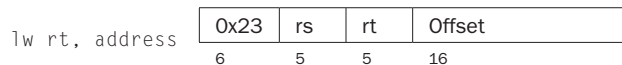
Load computed *address*—not the contents of the location—into register *rdest*.

Load byte**Load unsigned byte**

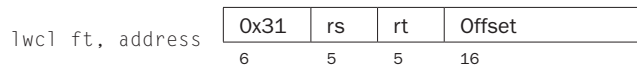
Load the byte at *address* into register *rt*. The byte is sign-extended by `lb`, but not by `lbu`.

Load halfword**Load unsigned halfword**

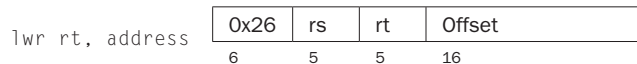
Load the 16-bit quantity (halfword) at *address* into register *rt*. The halfword is sign-extended by `lh`, but not by `lhu`.

Load word

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

Load word coprocessor 1

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

Load word left**Load word right**

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

Load doubleword

ld rdest, address *pseudoinstruction*

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

Unaligned load halfword

ulh rdest, address *pseudoinstruction*

Unaligned load halfword unsigned

```
ulhu rdest, address pseudoinstruction
```

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

Unaligned load word

```
ulw rdest, address pseudoinstruction
```

Load the 32-bit quantity (word) at the possibly unaligned *address* into register *rdest*.

Load linked

```
ll rt, address
```

0x30	rs	rt	Offset
6	5	5	16

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

```
sb rt, address
```

0x28	rs	rt	Offset
6	5	5	16

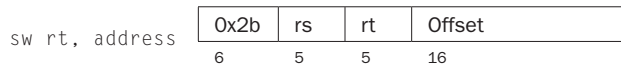
Store the low byte from register *rt* at *address*.

Store halfword

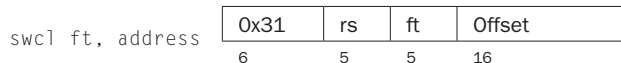
```
sh rt, address
```

0x29	rs	rt	Offset
6	5	5	16

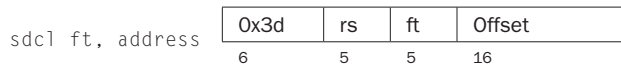
Store the low halfword from register *rt* at *address*.

Store word

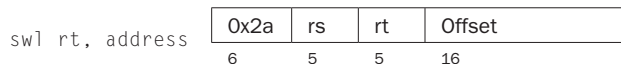
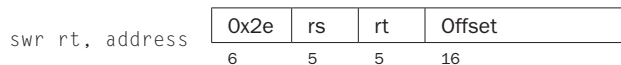
Store the word from register *rt* at *address*.

Store word coprocessor 1

Store the floating-point value in register *ft* of floating-point coprocessor at *address*.

Store double coprocessor 1

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**Store word right**

Store the left (right) bytes from register *rt* at the possibly unaligned *address*.

Store doubleword

sd *rsrc*, *address* *pseudoinstruction*

Store the 64-bit quantity in registers *rsrc* and *rsrc + 1* at *address*.

Unaligned store halfword

```
ush rsrc, address pseudoinstruction
```

Store the low halfword from register `rsrc` at the possibly unaligned *address*.

Unaligned store word

```
usw rsrc, address pseudoinstruction
```

Store the word from register `rsrc` at the possibly unaligned *address*.

Store conditional

```
sc rt, address
```

0x38	rs	rt	Offset
6	5	5	16

Store the 32-bit quantity (word) in register `rt` into memory at *address* and complete an atomic read-modify-write operation. If this atomic operation is successful, the memory word is modified and register `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 `rt`. Since SPIM does not simulate multiple processors, the instruction always succeeds.

Data Movement Instructions**Move**

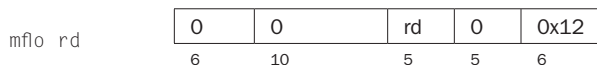
```
move rdest, rsrc pseudoinstruction
```

Move register `rsrc` to `rdest`.

Move from hi

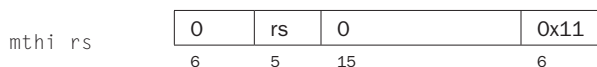
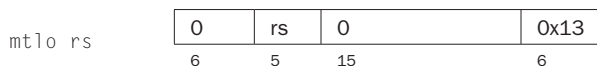
```
mfhi rd
```

0	0	rd	0	0x10
6	10	5	5	6

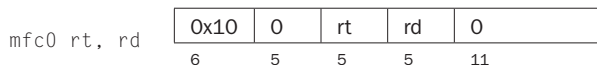
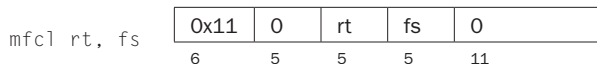
Move from lo

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

Move to hi**Move to lo**

Move register *rs* to the *hi* (*lo*) register.

Move from coprocessor 0**Move from coprocessor 1**

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

Move register *rd* in a coprocessor (register *fs* in the FPU) to CPU register *rt*. The floating-point unit is coprocessor 1.

Move double from coprocessor 1

```
mfcd.d rdest, frsrcl pseudoinstruction
```

Move floating-point registers `frsrcl` and `frsrcl + 1` to CPU registers `rdest` and `rdest + 1`.

Move to coprocessor 0

```
mtc0 rd, rt
```

0x10	4	rt	rd	0
6	5	5	5	11

Move to coprocessor 1

```
mtc1 rd, fs
```

0x11	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

```
movn rd, rs, rt
```

0	rs	rt	rd	0xb
6	5	5	5	11

Move register `rs` to register `rd` if register `rt` is not 0.

Move conditional zero

```
movz rd, rs, rt
```

0	rs	rt	rd	0xa
6	5	5	5	11

Move register `rs` to register `rd` if register `rt` is 0.

Move conditional on FP false

```
movf rd, rs, cc
```

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	0	rs	cc	1	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 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 *lwc1*, *swc1*, *mtc1*, and *mfc1* instructions or one double (64 bits) at a time by *ldc1* 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

abs.d fd, fs	0x11	1	0	fs	fd	5
	6	5	5	5	5	6

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
	6	5	5	5	5	6

Floating-point addition single

add.s fd, fs, ft	Ox11	Ox10	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	Ox11	Ox11	0	fs	fd	Oxe
	6	5	5	5	5	6

ceil.w.s fd, fs	Ox11	Ox10	0	fs	fd	Oxe
-----------------	------	------	---	----	----	-----

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	Ox11	Ox11	ft	fs	cc	0	FC	2
	6	5	5	5	3	2	2	4

Compare equal single

c.eq.s cc fs, ft	Ox11	Ox10	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	Ox11	Ox11	ft	fs	cc	0	FC	Oxe
	6	5	5	5	3	2	2	4

Compare less than equal single

c.le.s cc fs, ft	Ox11	Ox10	ft	fs	cc	0	FC	Oxe
	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

<code>c.lt.d cc fs, ft</code>	0x11	0x11	ft	fs	cc	0	FC	0xc
	6	5	5	5	3	2	2	4

Compare less than single

<code>c.lt.s cc fs, ft</code>	0x11	0x10	ft	fs	cc	0	FC	0xc
	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 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

<code>cvt.d.s fd, fs</code>	0x11	0x10	0	fs	fd	0x21
	6	5	5	5	5	6

Convert integer to double

<code>cvt.d.w fd, fs</code>	0x11	0x14	0	fs	fd	0x21
	6	5	5	5	5	6

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

<code>cvt.s.d fd, fs</code>	0x11	0x11	0	fs	fd	0x20
	6	5	5	5	5	6

Convert integer to single

<code>cvt.s.w fd, fs</code>	0x11	0x14	0	fs	fd	0x20
	6	5	5	5	5	6

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

Convert double to integer

cvt.w.d fd, fs	0x11	0x11	0	fs	fd	0x24
	6	5	5	5	5	6

Convert single to integer

cvt.w.s fd, fs	0x11	0x10	0	fs	fd	0x24
	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*.

Floating-point divide double

div.d fd, fs, ft	0x11	0x11	ft	fs	fd	3
	6	5	5	5	5	6

Floating-point divide single

div.s fd, fs, ft	0x11	0x10	ft	fs	fd	3
	6	5	5	5	5	6

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

Floating-point floor to word

floor.w.d fd, fs	0x11	0x11	0	fs	fd	0xf
	6	5	5	5	5	6

floor.w.s fd, fs	0x11	0x10	0	fs	fd	0xf
------------------	------	------	---	----	----	-----

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

Load floating-point double

l.d fdest, address *pseudoinstruction*

Load floating-point single

`l.s fdest, address` *pseudoinstruction*

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

Move floating-point double

`mov.d fd, fs`

Ox11	Ox11	0	fs	fd	6
6	5	5	5	5	6

Move floating-point single

`mov.s fd, fs`

Ox11	Ox10	0	fs	fd	6
6	5	5	5	5	6

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

Move conditional floating-point double false

`movf.d fd, fs, cc`

Ox11	Ox11	cc	0	fs	fd	Ox11
6	5	3	2	5	5	6

Move conditional floating-point single false

`movf.s fd, fs, cc`

Ox11	Ox10	cc	0	fs	fd	Ox11
6	5	3	2	5	5	6

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

`movt.d fd, fs, cc`

Ox11	Ox11	cc	1	fs	fd	Ox11
6	5	3	2	5	5	6

Move conditional floating-point single true

`movt.s fd, fs, cc`

Ox11	Ox10	cc	1	fs	fd	Ox11
6	5	3	2	5	5	6

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

<code>movn.d fd, fs, rt</code>	0x11	0x11	rt	fs	fd	0x13
	6	5	5	5	5	6

Move conditional floating-point single not zero

<code>movn.s fd, fs, rt</code>	0x11	0x10	rt	fs	fd	0x13
	6	5	5	5	5	6

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

<code>movz.d fd, fs, rt</code>	0x11	0x11	rt	fs	fd	0x12
	6	5	5	5	5	6

Move conditional floating-point single zero

<code>movz.s fd, fs, rt</code>	0x11	0x10	rt	fs	fd	0x12
	6	5	5	5	5	6

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

Floating-point multiply double

<code>mul.d fd, fs, ft</code>	0x11	0x11	ft	fs	fd	2
	6	5	5	5	5	6

Floating-point multiply single

<code>mul.s fd, fs, ft</code>	0x11	0x10	ft	fs	fd	2
	6	5	5	5	5	6

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

Negate double

<code>neg.d fd, fs</code>	0x11	0x11	0	fs	fd	7
	6	5	5	5	5	6

Negate single

neg.s fd, fs	Ox11	Ox10	0	fs	fd	7
	6	5	5	5	5	6

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	Ox11	Ox11	0	fs	fd	Oxc
	6	5	5	5	5	6

round.w.s fd, fs	Ox11	Ox10	0	fs	fd	Oxc
------------------	------	------	---	----	----	-----

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	Ox11	Ox11	0	fs	fd	4
	6	5	5	5	5	6

Square root single

sqrt.s fd, fs	Ox11	Ox10	0	fs	fd	4
	6	5	5	5	5	6

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

Store floating-point double

s.d fdest, address *pseudoinstruction*

Store floating-point single

s.s fdest, address *pseudoinstruction*

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

Floating-point subtract double

sub.d fd, fs, ft	Ox11	Ox11	ft	fs	fd	1
	6	5	5	5	5	6

Floating-point subtract single

sub.s fd, fs, ft	0x11	0x10	ft	fs	fd	1
	6	5	5	5	5	6

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	0x11	0x11	0	fs	fd	0xd
	6	5	5	5	5	6

trunc.w.s fd, fs	0x11	0x10	0	fs	fd	0xd
------------------	------	------	---	----	----	-----

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	0x10	1	0	0x18
	6	1	19	6

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	0	0	0xc
	6	20	6

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

Break

break code	0	code	0xd
	6	20	6

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

No operation

nop	0	0	0	0	0	0
	6	5	5	5	5	6

Do nothing.