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

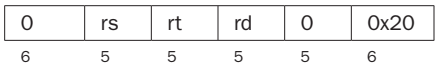
Absolute value

```
abs rdest, rsrc      pseudoinstruction
```

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

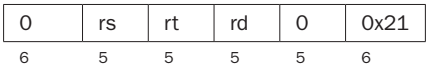
Addition (with overflow)

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



Addition (without overflow)

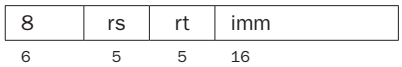
```
addu rd, rs, rt
```



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

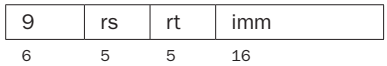
Addition immediate (with overflow)

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



Addition immediate (without overflow)

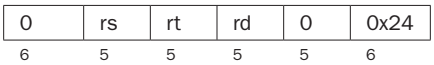
```
addiu rt, rs, imm
```



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

AND

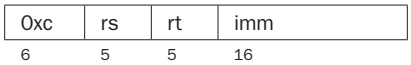
```
and rd, rs, rt
```



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

AND immediate

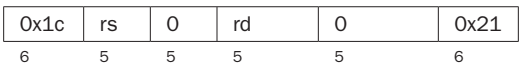
```
andi rt, rs, imm
```



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

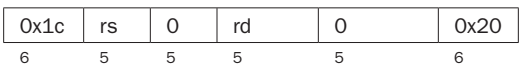
Count leading ones

```
clo rd, rs
```



Count leading zeros

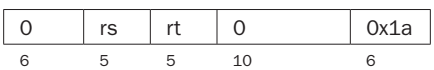
```
clz rd, rs
```



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.

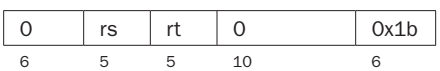
Divide (with overflow)

```
div rs, rt
```



Divide (without overflow)

```
divu rs, rt
```



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, rsrc1, src2      pseudoinstruction
```

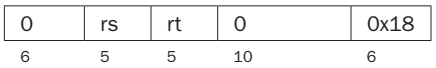
Divide (without overflow)

```
divu rdest, rsrc1, src2     pseudoinstruction
```

Put the quotient of register `rsrc1` and `src2` into register `rdest`.

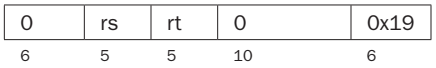
Multiply

```
mult rs, rt
```



Unsigned multiply

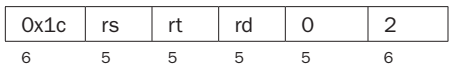
```
multu rs, rt
```



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
```



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

Multiply (with overflow)

```
mulo rdest, rsrc1, src2          pseudoinstruction
```

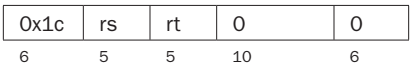
Unsigned multiply (with overflow)

```
mulou rdest, rsrc1, src2         pseudoinstruction
```

Put the low-order 32 bits of the product of register `rsrc1` and `src2` into register `rdest`.

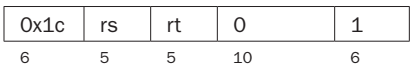
Multiply add

```
madd rs, rt
```



Unsigned multiply add

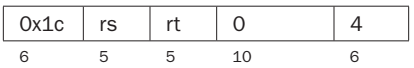
```
maddu rs, rt
```



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

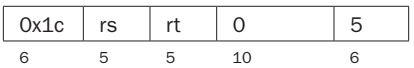
Multiply subtract

```
msub rs, rt
```



Unsigned multiply subtract

```
msub rs, rt
```



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
```

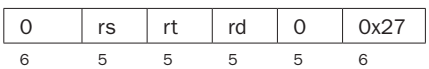
Negate value (without overflow)

```
negu rdest, rsrc                   pseudoinstruction
```

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

NOR

```
nor rd, rs, rt
```



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

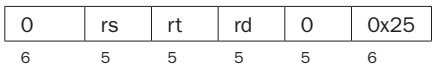
NOT

```
not rdest, rsrc                pseudoinstruction
```

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

OR

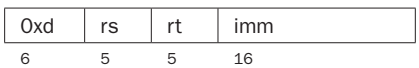
```
or rd, rs, rt
```



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

OR immediate

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



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

Remainder

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

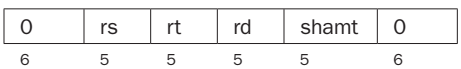
Unsigned remainder

```
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.

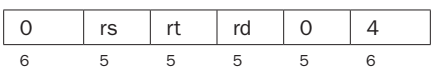
Shift left logical

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



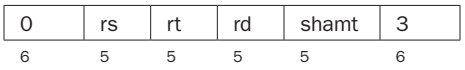
Shift left logical variable

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



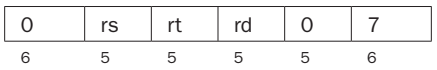
Shift right arithmetic

```
sra rd, rt, shamt
```



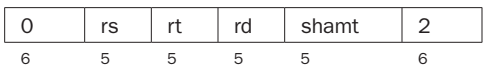
Shift right arithmetic variable

```
srav rd, rt, rs
```



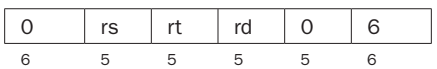
Shift right logical

```
srl rd, rt, shamt
```



Shift right logical variable

```
srlv rd, rt, rs
```



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
```

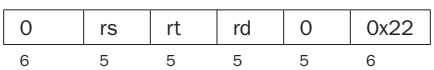
Rotate right

```
ror rdest, rsrc1, rsrc2          pseudoinstruction
```

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

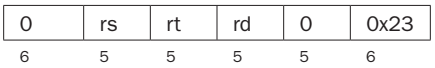
Subtract (with overflow)

```
sub rd, rs, rt
```



Subtract (without overflow)

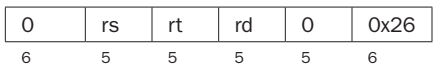
```
subu rd, rs, rt
```



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

Exclusive OR

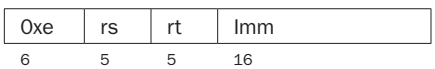
```
xor rd, rs, rt
```



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

XOR immediate

```
xori rt, rs, imm
```

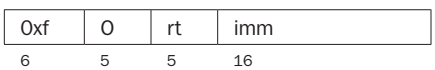


Put the logical XOR of register `rs` and the zero-extended immediate into register `rt`.

Constant-Manipulating Instructions

Load upper immediate

```
lui rt, imm
```



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

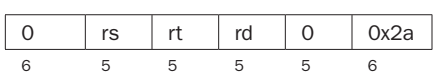
```
li rdest, imm                    pseudoinstruction
```

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

Comparison Instructions

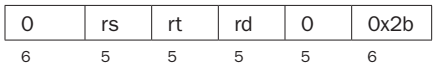
Set less than

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



Set less than unsigned

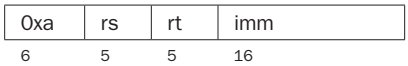
```
sltu rd, rs, rt
```



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

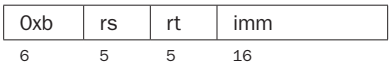
Set less than immediate

```
slti rt, rs, imm
```



Set less than unsigned immediate

```
sltiu rt, rs, imm
```



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.

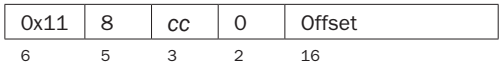
Branch instruction

```
b label                                    pseudoinstruction
```

Unconditionally branch to the instruction at the label.

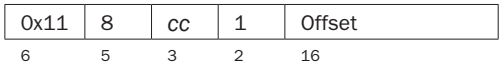
Branch coprocessor false

```
bclf cc label
```



Branch coprocessor true

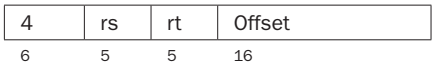
```
bclt cc label
```



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

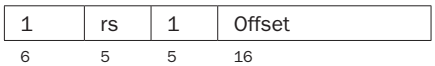
```
beq rs, rt, label
```



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

Branch on greater than equal zero

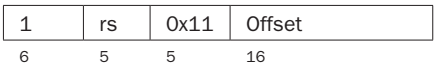
```
bgez rs, label
```



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

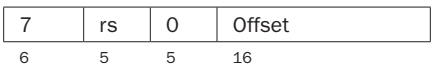
```
bgezal rs, label
```



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

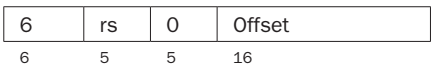
```
bgtz rs, label
```



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

Branch on less than equal zero

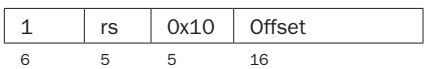
```
blez rs, label
```



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

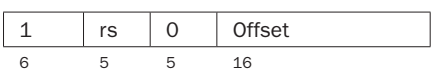
```
bltzal rs, label
```



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

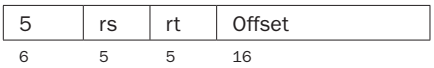
```
bltz rs, label
```



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
```



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

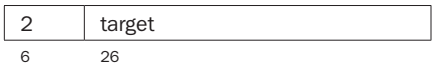
```
bnez rsrc, label                 pseudoinstruction
```

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

Jump Instructions

Jump

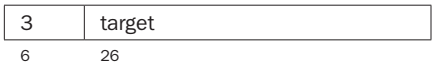
```
j target
```



Unconditionally jump to the instruction at target.

Jump and link

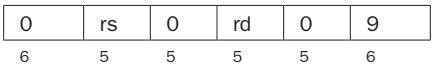
```
jal target
```



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

Jump and link register

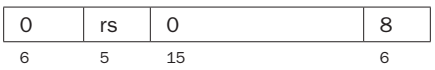
```
jalr rs, rd
```



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

```
jr rs
```

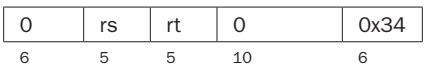


Unconditionally jump to the instruction whose address is in register `rs`.

Trap Instructions

Trap if equal

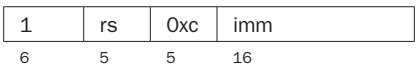
```
teq rs, rt
```



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

Trap if equal immediate

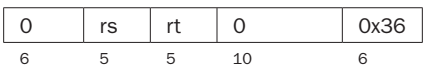
```
teqi rs, imm
```



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

Trap if not equal

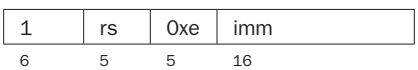
```
teq rs, rt
```



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

Trap if not equal immediate

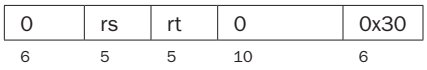
```
teqi rs, imm
```



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

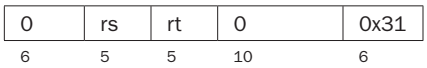
Trap if greater equal

`tge rs, rt`



Unsigned trap if greater equal

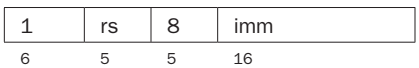
`tgeu rs, rt`



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

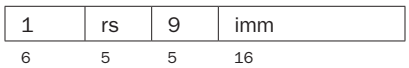
Trap if greater equal immediate

`tgei rs, imm`



Unsigned trap if greater equal immediate

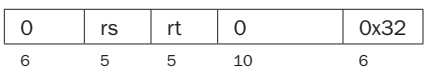
`tgeiu rs, imm`



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

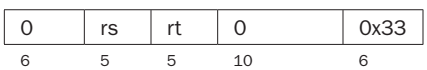
Trap if less than

`tlt rs, rt`



Unsigned trap if less than

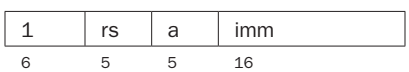
`tltu rs, rt`



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

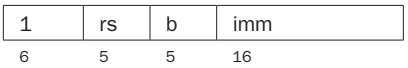
Trap if less than immediate

`tlti rs, imm`



Unsigned trap if less than immediate

```
tltiu rs, imm
```



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

Load Instructions

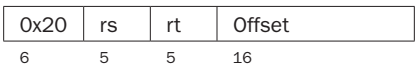
Load address

```
la rdest, address                    pseudoinstruction
```

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

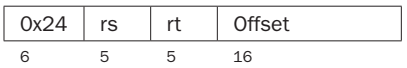
Load byte

```
lb rt, address
```



Load unsigned byte

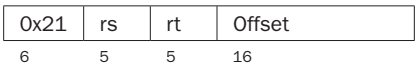
```
lbu rt, address
```



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

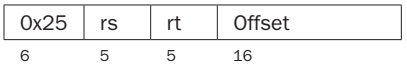
Load halfword

```
lh rt, address
```



Load unsigned halfword

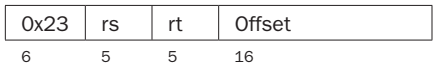
```
lhu rt, address
```



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

Load word

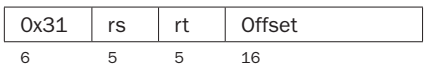
```
lw rt, address
```



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

Load word coprocessor 1

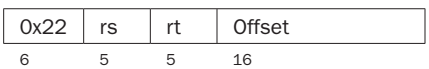
```
lwc1 ft, address
```



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

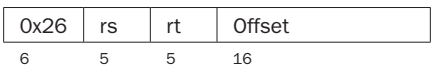
Load word left

```
lwl rt, address
```



Load word right

```
lwr rt, address
```



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`.

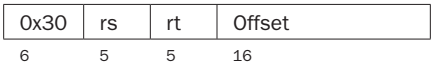
Unaligned load word

```
ulw rdest, address           pseudoinstruction
```

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

Load linked

```
ll rt, address
```

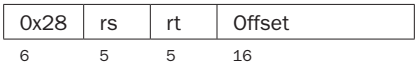


Load the 32-bit quantity (word) at *address* into register `rt` and start an atomic read-modify-write operation. This operation is completed by a store conditional (`sc`) instruction, which will fail if another processor writes into the block containing the loaded word. Since SPIM does not simulate multiple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

Store Instructions

Store byte

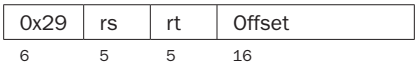
```
sb rt, address
```



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

Store halfword

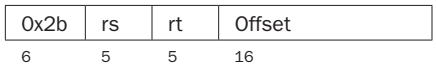
```
sh rt, address
```



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

Store word

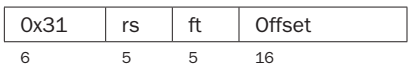
```
sw rt, address
```



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

Store word coprocessor 1

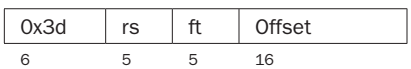
```
swc1 ft, address
```



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

Store double coprocessor 1

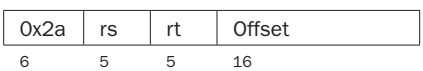
```
sdc1 ft, address
```



Store the doubleword floating-point value in registers `ft` and `ft + 1` of floating-point coprocessor at *address*. Register `ft` must be even numbered.

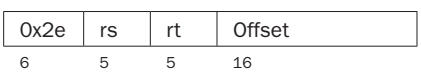
Store word left

```
swl rt, address
```



Store word right

```
swr rt, address
```



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*.

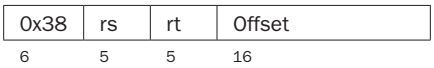
Unaligned store word

```
usw rsrc, address                pseudoinstruction
```

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

Store conditional

```
sc rt, address
```



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

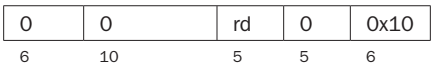
Move

```
move rdest, rsrc                 pseudoinstruction
```

Move register `rsrc` to `rdest`.

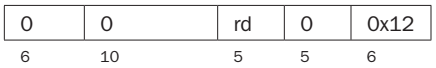
Move from hi

```
mfhi rd
```



Move from lo

`mflo rd`

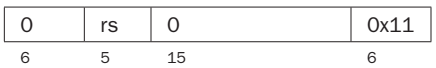


The multiply and divide unit produces its result in two additional registers, `hi` and `lo`. These instructions move values to and from these registers. The multiply, divide, and remainder pseudoinstructions that make this unit appear to operate on the general registers move the result after the computation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

Move the `hi` (`lo`) register to register `rd`.

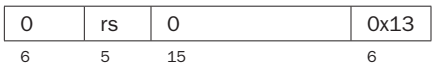
Move to hi

`mthi rs`



Move to lo

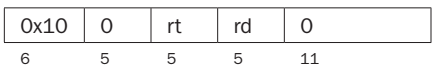
`mtlo rs`



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

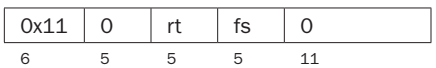
Move from coprocessor 0

`mfc0 rt, rd`



Move from coprocessor 1

`mfcl rt, fs`



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.

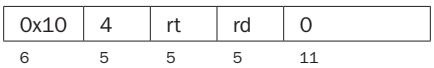
Move double from coprocessor 1

```
mfc1.d rdest, frsrc1          pseudoinstruction
```

Move floating-point registers `frsrc1` and `frsrc1 + 1` to CPU registers `rdest` and `rdest + 1`.

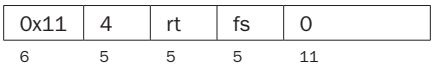
Move to coprocessor 0

```
mtc0 rd, rt
```



Move to coprocessor 1

```
mtc1 rd, fs
```



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

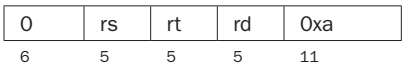
Move conditional not zero

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

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

Move conditional zero

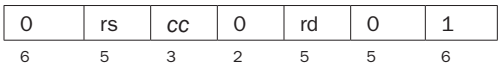
```
movz rd, rs, rt
```



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

Move conditional on FP false

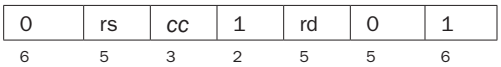
```
movf rd, rs, cc
```



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`



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

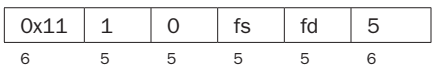
The MIPS has a floating-point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating-point numbers. This coprocessor has its own registers, which are numbered `$f0`–`$f31`. Because these registers are only 32 bits wide, two of them are required to hold 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 (`bc1f` or `bc1t`) 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`).

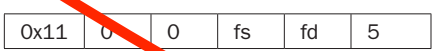
Floating-point absolute value double

`abs.d fd, fs`



Floating-point absolute value single

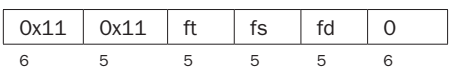
`abs.s fd, fs`



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

Floating-point addition double

`add.d fd, fs, ft`



1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

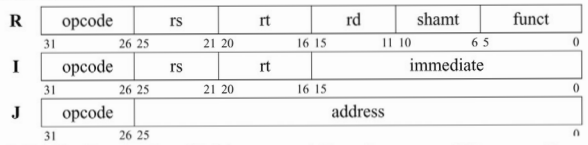
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

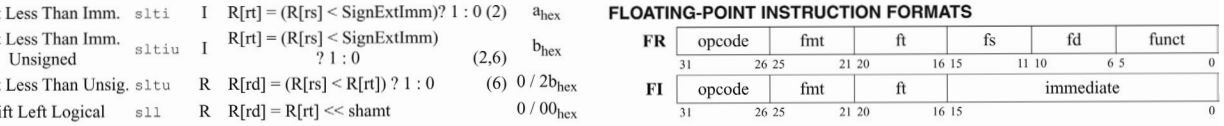
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

MIPS Reference Data Card ("Green Card")

OPCODES, BASE CONVERSION, ASCII SYMBOLS

③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal	Hexadecimal	ASCII Character	Decimal	Hexadecimal	ASCII Character
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	’	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0

(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}).*f* = s (single); if fmt(25:21)==17_{ten} (11_{hex}).*f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

where Single Precision Bias = 127, Double Precision Bias = 1023.

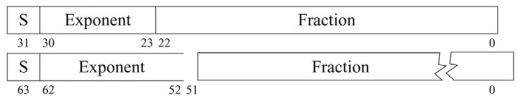
④

IEEE 754 Symbols

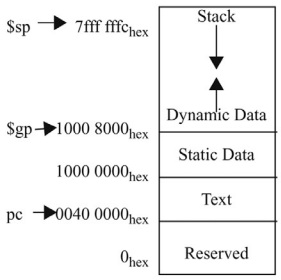
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

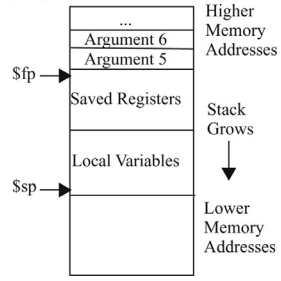
IEEE Single Precision and Double Precision Formats:



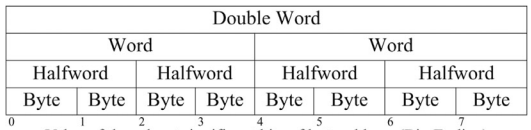
MEMORY ALLOCATION



STACK FRAME

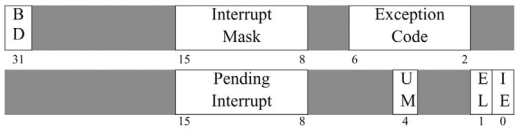


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES

	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki	10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^6	Mega-	M	2^{20}	Mebi-	Mi	10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^9	Giga-	G	2^{30}	Gibi-	Gi	10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti	10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together