Warning: Programs that use these syscalls to read from the terminal should not use memory-mapped I/O (see Section A.8).

sbrk returns a pointer to a block of memory containing *n* additional bytes. exit stops the program SPIM is running. exit2 terminates the SPIM program, and the argument to exit2 becomes the value returned when the SPIM simulator itself terminates.

print_char and read_char write and read a single character. open, read, write, and close are the standard UNIX library calls.

A.10 MIPS R2000 Assembly Language

A MIPS processor consists of an integer processing unit (the CPU) and a collection of coprocessors that perform ancillary tasks or operate on other types of data, such as floating-point numbers (see Figure A.10.1). SPIM simulates two coprocessors. Coprocessor 0 handles exceptions and interrupts. Coprocessor 1 is the floating-point unit. SPIM simulates most aspects of this unit.

Addressing Modes

MIPS is a load store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers. The bare machine provides only one memory-addressing mode: c(rx), which uses the sum of the immediate c and register rx as the address. The virtual machine provides the following addressing modes for load and store instructions:

Format	Address computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
label	address of label
label ± imm	address of label + or – immediate
label ± imm (register)	address of label + or – (immediate + contents of register)

Most load and store instructions operate only on aligned data. A quantity is *aligned* if its memory address is a multiple of its size in bytes. Therefore, a halfword

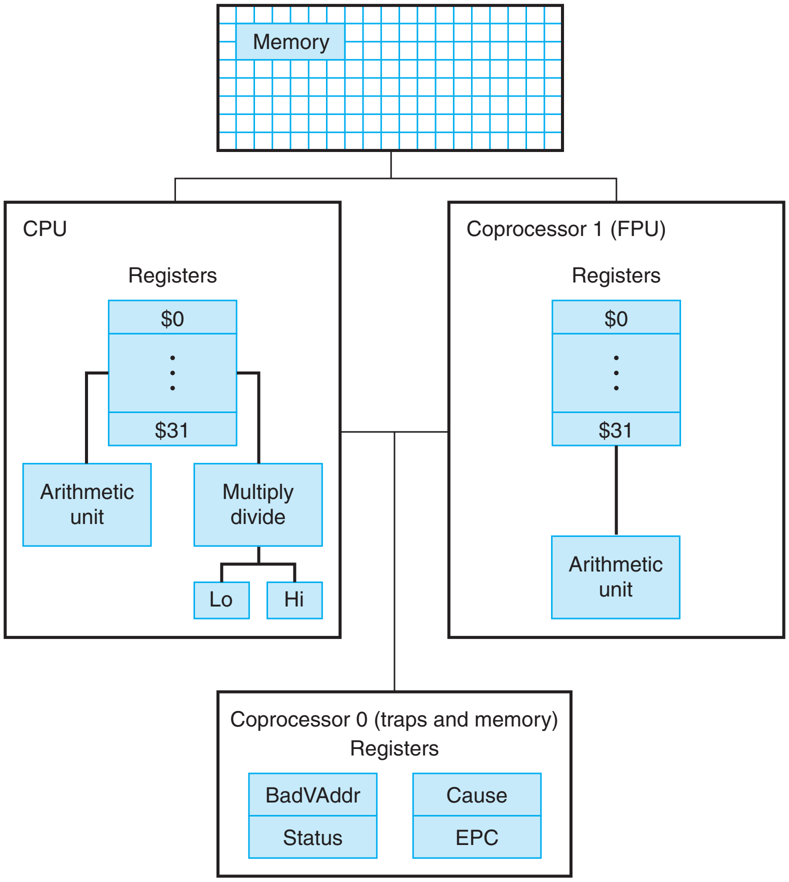


FIGURE A.10.1 MIPS R2000 CPU and FPU.

object must be stored at even addresses, and a full word object must be stored at addresses that are a multiple of four. However, MIPS provides some instructions to manipulate unaligned data (`lwl`, `lwr`, `swl`, and `swr`).

Elaboration: The MIPS assembler (and SPIM) synthesizes the more complex addressing modes by producing one or more instructions before the load or store to compute a complex address. For example, suppose that the label `table` referred to memory location 0x10000004 and a program contained the instruction

```
    ld $a0, table + 4($a1)
```

The assembler would translate this instruction into the instructions

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

The first instruction loads the upper bits of the label's address into register `$at`, which is the register that the assembler reserves for its own use. The second instruction adds the contents of register `$a1` to the label's partial address. Finally, the load instruction uses the hardware address mode to add the sum of the lower bits of the label's address and the offset from the original instruction to the value in register `$at`.

Assembler Syntax

Comments in assembler files begin with a sharp sign (#). Everything from the sharp sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underbars (_), and dots (.) that do not begin with a number. Instruction opcodes are reserved words that *cannot* be used as identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```
      .data
item: .word 1
      .text
      .globl main      # Must be global
main: lw         $t0, item
```

Numbers are base 10 by default. If they are preceded by *0x,* they are interpreted as hexadecimal. Hence, 256 and 0x100 denote the same value.

Strings are enclosed in double quotes ("). Special characters in strings follow the C convention:

- newline `\n`
- tab `\t`
- quote `\"`

SPIM supports a subset of the MIPS assembler directives:

`.align n` Align the next datum on a 2^n byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align 0` turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive.

`.ascii str` Store the string *str* in memory, but do not null-terminate it.

`.asciiz str` Store the string *str* in memory and null-terminate it.

`.byte b1,..., bn` Store the *n* values in successive bytes of memory.

`.data <addr>` Subsequent items are stored in the data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.

`.double d1,..., dn` Store the *n* floating-point double precision num-bers in successive memory locations.

`.extern sym size` Declare that the datum stored at *sym* is *size* bytes large and is a global label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.

`.float f1,..., fn` Store the *n* floating-point single precision numbers in successive memory locations.

`.globl sym` Declare that label *sym* is global and can be referenced from other files.

`.half h1,..., hn` Store the *n* 16-bit quantities in successive memory halfwords.

`.kdata <addr>` Subsequent data items are stored in the kernel data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.

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

`.set noat` and `.set at` The first directive prevents SPIM from complaining about subsequent instructions that use register `$at`. The second directive re-enables the warning. Since pseudoinstructions expand into code that uses register `$at`, programmers must be very careful about leaving values in this register.

`.space n` Allocates *n* bytes of space in the current segment (which must be the data segment in SPIM).

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

`.word w1,..., wn` Store the *n* 32-bit quantities in successive memory words.

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

Encoding MIPS Instructions

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

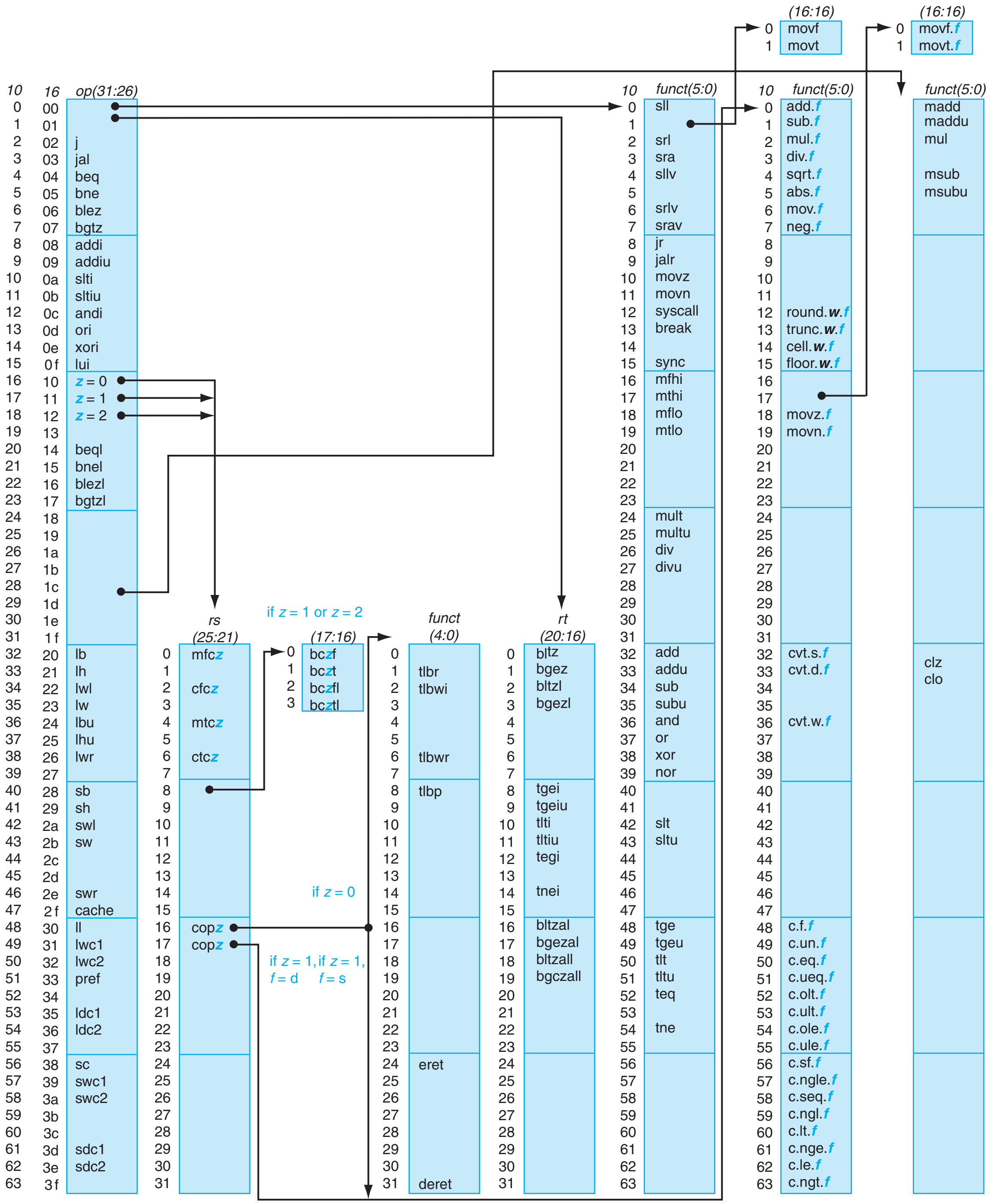
Instruction Format

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

Addition (with overflow)

	0	rs	rt	rd	0	0x20
`add rd, rs, rt`	6	5	5	5	5	6

the `add` instruction consists of six fields. Each field's size in bits is the small number below the field. This instruction begins with six bits of 0s. Register specifiers begin with an *r*, so the next field is a 5-bit register specifier called `rs`. This is the same register that is the second argument in the symbolic assembly at the left of this line. Another common field is $\mathtt{imm}_{16}$, which is a 16-bit immediate number.



10	16	op(31:26)	funct(5:0)	funct(5:0)	funct(5:0)
0	00		sll	add.*f*	madd
1	01			sub.*f*	maddu
2	02	j	srl	mul.*f*	mul
3	03	jal	sra	div.*f*	
4	04	beq	sllv	sqrt.*f*	msub
5	05	bne		abs.*f*	msubu
6	06	blez	srlv	mov.*f*	
7	07	bgtz	srav	neg.*f*	
8	08	addi	jr		
9	09	addiu	jalr		
10	0a	slti	movz		
11	0b	sltiu	movn		
12	0c	andi	syscall	round.*w*.*f*	
13	0d	ori	break	trunc.*w*.*f*	
14	0e	xori		cell.*w*.*f*	
15	0f	lui	sync	floor.*w*.*f*	
16	10	*z* = 0	mfhi		
17	11	*z* = 1	mthi		
18	12	*z* = 2	mflo	movz.*f*	
19	13		mtlo	movn.*f*	
20	14	beql			
21	15	bnel			
22	16	blezl			
23	17	bgtzl			
24	18		mult		
25	19		multu		
26	1a		div		
27	1b		divu		
28	1c				
29	1d				
30	1e				
31	1f				
32	20	lb	add	cvt.s.*f*	clz
33	21	lh	addu	cvt.d.*f*	clo
34	22	lwl	sub		
35	23	lw	subu		
36	24	lbu	and	cvt.w.*f*	
37	25	lhu	or		
38	26	lwr	xor		
39	27		nor		
40	28	sb			
41	29	sh			
42	2a	swl	slt		
43	2b	sw	sltu		
44	2c				
45	2d				
46	2e	swr			
47	2f	cache			
48	30	ll	tge	c.f.*f*	
49	31	lwc1	tgeu	c.un.*f*	
50	32	lwc2	tlt	c.eq.*f*	
51	33	pref	tltu	c.ueq.*f*	
52	34		teq	c.olt.*f*	
53	35	ldc1		c.ult.*f*	
54	36	ldc2	tne	c.ole.*f*	
55	37			c.ule.*f*	
56	38	sc		c.sf.*f*	
57	39	swc1		c.ngle.*f*	
58	3a	swc2		c.seq.*f*	
59	3b			c.ngl.*f*	
60	3c			c.lt.*f*	
61	3d	sdc1		c.nge.*f*	
62	3e	sdc2		c.le.*f*	
63	3f			c.ngt.*f*	

	rs (25:21)	funct (4:0)	rt (20:16)
0	mfc*z*		bltz
1		tlbr	bgez
2	cfc*z*	tlbwi	bltzl
3			bgezl
4	mtc*z*		
5			
6	ctc*z*	tlbwr	
7			
8		tlbp	tgei
9			tgeiu
10			tlti
11			tltiu
12			tegi
13			
14			tnei
15			
16	cop*z*		bltzal
17	cop*z*		bgezal
18			bltzall
19			bgczall
20			
21			
22			
23			
24		eret	
25			
26			
27			
28			
29			
30			
31		deret	

	(17:16)
0	bc*z*f
1	bc*z*t
2	bc*z*fl
3	bc*z*tl

	(16:16)	(16:16)
0	movf	movf.*f*
1	movt	movt.*f*

FIGURE A.10.2 MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10, and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for 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

`and rd, rs, rt`

0	rs	rt	rd	0	0x24
6	5	5	5	5	6

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

AND immediate

`andi rt, rs, imm`

0xc	rs	rt	imm
6	5	5	16

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

Count leading ones

`clo rd, rs`

0x1c	rs	0	rd	0	0x21
6	5	5	5	5	6

Count leading zeros

`clz rd, rs`

0x1c	rs	0	rd	0	0x20
6	5	5	5	5	6

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`

0	rs	rt	0	0x1a
6	5	5	10	6

Divide (without overflow)

`divu rs, rt`

0	rs	rt	0	0x1b
6	5	5	10	6

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
```

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, 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
```

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

```
msub 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
```

0xe	rs	rt	Imm
6	5	5	16

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

Constant-Manipulating Instructions

Load upper immediate

```
lui rt, imm
```

0xf	0	rt	imm
6	5	5	16

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

Load immediate

```
li rdest, imm
```

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

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

0	rs	rt	rd	0	0x2b
6	5	5	5	5	6

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

Set less than immediate

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

0xa	rs	rt	imm
6	5	5	16

Set less than unsigned immediate

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

0xb	rs	rt	imm
6	5	5	16

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

```
bgezal rs, label
```

1	rs	0x11	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. Save the address of the next instruction in register 31.

Branch on greater than zero

```
bgtz rs, label
```

7	rs	0	Offset
6	5	5	16

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
```

6	rs	0	Offset
6	5	5	16

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
```

1	rs	0x10	Offset
6	5	5	16

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
```

1	rs	0	Offset
6	5	5	16

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

`jalr rs, rd`

0	rs	0	rd	0	9
6	5	5	5	5	6

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

Jump register

`jr rs`

0	rs	0	8
6	5	15	6

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

Trap Instructions

Trap if equal

`teq rs, rt`

0	rs	rt	0	0x34
6	5	5	10	6

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

Trap if equal immediate

`teqi rs, imm`

1	rs	0xc	imm
6	5	5	16

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

Trap if not equal

`teq rs, rt`

0	rs	rt	0	0x36
6	5	5	10	6

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

Trap if not equal immediate

`teqi rs, imm`

1	rs	0xe	imm
6	5	5	16

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

Trap if greater equal

`tge rs, rt`

0	rs	rt	0	0x30
6	5	5	10	6

Unsigned trap if greater equal

`tgeu rs, rt`

0	rs	rt	0	0x31
6	5	5	10	6

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

Trap if greater equal immediate

`tgei rs, imm`

1	rs	8	imm
6	5	5	16

Unsigned trap if greater equal immediate

`tgeiu rs, imm`

1	rs	9	imm
6	5	5	16

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

Trap if less than

`tlt rs, rt`

0	rs	rt	0	0x32
6	5	5	10	6

Unsigned trap if less than

`tltu rs, rt`

0	rs	rt	0	0x33
6	5	5	10	6

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

Trap if less than immediate

`tlti rs, imm`

1	rs	a	imm
6	5	5	16

Unsigned trap if less than immediate

```
tltiu rs, imm
```

1	rs	b	imm
6	5	5	16

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
```

0x20	rs	rt	Offset
6	5	5	16

Load unsigned byte

```
lbu rt, address
```

0x24	rs	rt	Offset
6	5	5	16

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

Load halfword

```
lh rt, address
```

0x21	rs	rt	Offset
6	5	5	16

Load unsigned halfword

```
lhu rt, address
```

0x25	rs	rt	Offset
6	5	5	16

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
```

0x23	rs	rt	Offset
6	5	5	16

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

Load word coprocessor 1

```
lwc1 ft, address
```

0x31	rs	rt	Offset
6	5	5	16

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

Load word left

```
lwl rt, address
```

0x22	rs	rt	Offset
6	5	5	16

Load word right

```
lwr rt, address
```

0x26	rs	rt	Offset
6	5	5	16

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

`sw rt, address`

0x2b	rs	rt	Offset
6	5	5	16

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

Store word coprocessor 1

`swc1 ft, address`

0x31	rs	ft	Offset
6	5	5	16

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

Store double coprocessor 1

`sdc1 ft, address`

0x3d	rs	ft	Offset
6	5	5	16

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`

0x2a	rs	rt	Offset
6	5	5	16

Store word right

`swr rt, address`

0x2e	rs	rt	Offset
6	5	5	16

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

```
mflo rd
```

0	0	rd	0	0x12
6	10	5	5	6

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
```

0	rs	0	0x11
6	5	15	6

Move to lo

```
mtlo rs
```

0	rs	0	0x13
6	5	15	6

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

Move from coprocessor 0

```
mfc0 rt, rd
```

0x10	0	rt	rd	0
6	5	5	5	11

Move from coprocessor 1

```
mfcl rt, fs
```

0x11	0	rt	fs	0
6	5	5	5	11

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

Move register `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
```

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 (`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
```

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
```

0x11	0x10	ft	fs	fd	0
6	5	5	5	5	6

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

Floating-point ceiling to word

```
ceil.w.d fd, fs
```

0x11	0x11	0	fs	fd	0xe
6	5	5	5	5	6

```
ceil.w.s fd, fs
```

0x11	0x10	0	fs	fd	0xe

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

Compare equal double

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

0x11	0x11	ft	fs	*cc*	0	FC	2
6	5	5	5	3	2	2	4

Compare equal single

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

0x11	0x10	ft	fs	*cc*	0	FC	2
6	5	5	5	3	2	2	4

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

Compare less than equal double

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

0x11	0x11	ft	fs	*cc*	0	FC	0xe
6	5	5	5	3	2	2	4

Compare less than equal single

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

0x11	0x10	ft	fs	*cc*	0	FC	0xe
6	5	5	5	3	2	2	4

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

Compare less than double

`c.lt.d cc fs, ft`

0x11	0x11	ft	fs	*cc*	0	FC	0xc
6	5	5	5	3	2	2	4

Compare less than single

`c.lt.s cc fs, ft`

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

`cvt.d.s fd, fs`

0x11	0x10	0	fs	fd	0x21
6	5	5	5	5	6

Convert integer to double

`cvt.d.w fd, fs`

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

`cvt.s.d fd, fs`

0x11	0x11	0	fs	fd	0x20
6	5	5	5	5	6

Convert integer to single

`cvt.s.w fd, fs`

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
```

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

Move floating-point single

```
mov.s fd, fs
```

0x11	0x10	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
```

0x11	0x11	*cc*	0	fs	fd	0x11
6	5	3	2	5	5	6

Move conditional floating-point single false

```
movf.s fd, fs, cc
```

0x11	0x10	*cc*	0	fs	fd	0x11
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
```

0x11	0x11	*cc*	1	fs	fd	0x11
6	5	3	2	5	5	6

Move conditional floating-point single true

```
movt.s fd, fs, cc
```

0x11	0x10	*cc*	1	fs	fd	0x11
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

```
movn.d fd, fs, rt
```

0x11	0x11	rt	fs	fd	0x13
6	5	5	5	5	6

Move conditional floating-point single not zero

```
movn.s fd, fs, rt
```

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

```
movz.d fd, fs, rt
```

0x11	0x11	rt	fs	fd	0x12
6	5	5	5	5	6

Move conditional floating-point single zero

```
movz.s fd, fs, rt
```

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

```
mul.d fd, fs, ft
```

0x11	0x11	ft	fs	fd	2
6	5	5	5	5	6

Floating-point multiply single

```
mul.s fd, fs, ft
```

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

```
neg.d fd, fs
```

0x11	0x11	0	fs	fd	7
6	5	5	5	5	6

Negate single

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

0x11	0x10	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
```

0x11	0x11	0	fs	fd	0xc
6	5	5	5	5	6

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

0x11	0x10	0	fs	fd	0xc

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
```

0x11	0x11	0	fs	fd	4
6	5	5	5	5	6

Square root single

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

0x11	0x10	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
```

0x11	0x11	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 A.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.