

# SPIM S20: A MIPS R2000 Simulator

“ $\frac{1}{25}$ <sup>th</sup> the performance at none of the cost”

James R. Larus

Copyright ©1990–2004 by James R. Larus  
(This document may be copied without royalties,  
so long as this copyright notice remains on it.)

## 1 SPIM

### 1.1 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. Opcodes for instructions are reserved words that are **not** valid 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
```

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 assembler directives provided by the MIPS assembler:

**.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 in memory, but do not null-terminate it.

**.asciiz str**

Store the string in memory and null-terminate it.

**.byte b1, ..., bn**

Store the  $n$  values in successive bytes of memory.

**.comm sym size**  
 Allocate *size* bytes of data segment for symbol *sym*.

**.data <addr>**  
 The following data items should be stored in the data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

**.double d1, ..., dn**  
 Store the *n* floating point double precision numbers in successive memory locations.

**.extern sym size**  
 Declare that the datum stored at **sym** is **size** bytes large and is a global symbol. 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 symbol **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>**  
 The following data items should be stored in the kernel data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

**.ktext <addr>**  
 The next 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, the items are stored beginning at address *addr*.

**.label sym**  
 Declare that symbol **sym** is a label.

**.lcomm sym size**  
 Allocate *size* bytes for symbol *sym* in the portion of the data segment that can be accessed via register **\$gp**.

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

**.set noat**  
 Permit the program to refer to the **\$at** register explicitly, and forbid SPIM from generating pseudoinstructions that modify **\$at**.

**.set at**  
 Forbid the program from referring to the **\$at** register explicitly, and permit SPIM to generate pseudoinstructions that modify **\$at** (the default).

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	integer (in \$v0)
print_float	2	\$f12 = float	float (in \$f0)
print_double	3	\$f12 = double	double (in \$f0)
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_character	11	\$a0 = character	character (in \$v0)
read_character	12		file descriptor (in \$v0)
open	13	\$a0 = filename, \$a1 = flags, \$a2 = mode	
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = count	bytes read (in \$v0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = count	bytes written (in \$v0)
close	16	\$a0 = file descriptor	0 (in \$v0)
exit2	17	\$a0 = value	

Table 1: System services.

`.text <addr>`

The next 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, the items are stored beginning 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`).

## 1.2 System Calls

SPIM provides a small set of operating-system-like services through the system call (`syscall`) instruction. To request a service, a program loads the system call code (see Table 1) into register `$v0` and the arguments into registers `$a0..$a3` (or `$f12` for floating point values). System calls that return values put their result in register `$v0` (or `$f0` for floating point results). For example, to print “the answer = 5”, use the commands:

```

.data
str: .asciiz "the answer = "
.text
li $v0, 4      # system call code for print_str
la $a0, str    # address of string to print
syscall        # print the string

li $v0, 1      # system call code for print_int
li $a0, 5      # integer to print

```

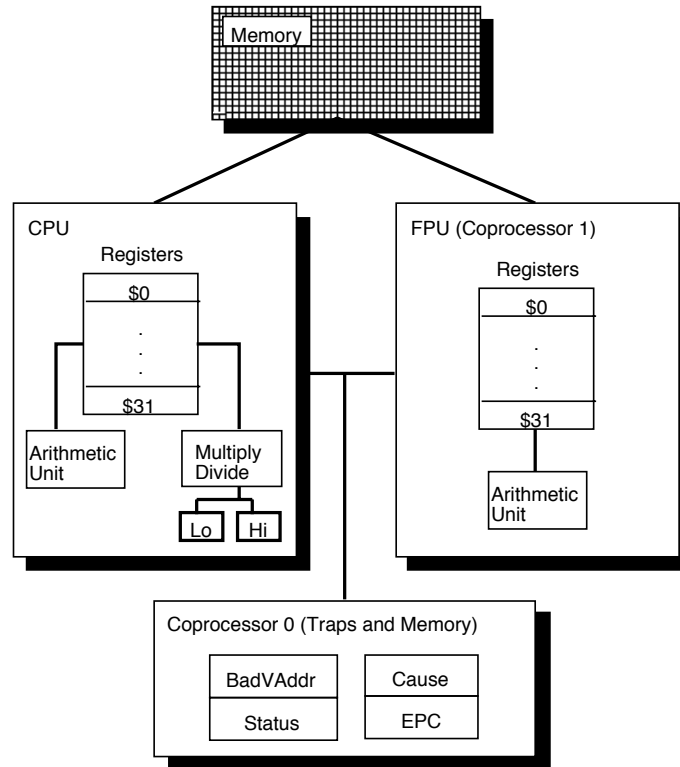


Figure 1: MIPS R2000 CPU and FPU

```
syscall          # print it
```

`print_int` is passed an integer and prints it on the console. `print_float` prints a single floating point number. `print_double` prints a double precision number. `print_string` is passed a pointer to a null-terminated string, which it writes to the console. `print_character` prints a single ASCII character.

`read_int`, `read_float`, and `read_double` read an entire line of input up to and including the newline. Characters following the number are ignored. `read_string` has the same semantics as the Unix library routine `fgets`. It reads up to  $n - 1$  characters into a buffer and terminates the string with a null byte. If there are fewer characters on the current line, it reads through the newline and again null-terminates the string. `read_character` reads a single ASCII character.

`sbrk` returns a pointer to a block of memory containing  $n$  additional bytes. This pointer is word aligned. `exit` stops a program from running. `exit2` stops the program from running and takes an argument, which is the value that `spim` or `xspim` uses in its call on `exit`.

`open`, `read`, `write` and `close` behave the same as the Unix system calls of the same name. They all return  $-1$  on failure.

## 2 Description of the MIPS R2000

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 1). SPIM simulates two coprocessors. Coprocessor 0 handles traps, exceptions, and the virtual memory system. SPIM simulates most of the first two and entirely omits details of

Register Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and
v1	3	results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp or s8	30	Frame pointer
ra	31	Return address (used by function call)

Table 2: MIPS registers and the convention governing their use.

the memory system. Coprocessor 1 is the floating point unit. SPIM simulates most aspects of this unit.

## 2.1 CPU Registers

The MIPS (and SPIM) central processing unit contains 32 general purpose 32-bit registers that are numbered 0–31. Register  $n$  is designated by  $\$n$ . Register  $\$0$  always contains the hardwired value 0. MIPS has established a set of conventions as to how registers should be used. These suggestions are guidelines, which are not enforced by the hardware. However a program that violates them will not work properly with other software. Table 2 lists the registers and describes their intended use.

Registers  $\$at$  (1),  $\$k0$  (26), and  $\$k1$  (27) are reserved for use by the assembler and operating system.

Registers  $\$a0$ – $\$a3$  (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers  $\$v0$  and  $\$v1$  (2, 3) are used to return values

from functions. Registers `$t0–$t9` (8–15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls. Registers `$s0–$s7` (16–23) are callee-saved registers that hold long-lived values that should be preserved across calls.

Register `$sp` (29) is the stack pointer, which points to the last location in use on the stack.<sup>1</sup> Register `$fp` (30) is the frame pointer.<sup>2</sup> Register `$ra` (31) is written with the return address for a call by the `jal` instruction.

Register `$gp` (28) is a global pointer that points into the middle of a 64K block of memory in the heap that holds constants and global variables. The objects in this heap can be quickly accessed with a single load or store instruction.

## 2.2 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 (integer) `c` and the contents of 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
symbol	address of symbol
symbol ± imm	address of symbol + or – immediate
symbol (register)	address of symbol + contents of register
symbol ± imm (register)	(address of symbol + 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 object must be stored at even addresses and a full word object must be stored at addresses that are a multiple of 4. However, MIPS provides some instructions for manipulating unaligned data.

## 2.3 Arithmetic and Logical Instructions

In all instructions below, `Src2` can either be a register or an immediate value (a 16 bit integer). The immediate forms of the instructions are only included for reference. The assembler will translate the more general form of an instruction (e.g., `add`) into the immediate form (e.g., `addi`) if the second argument is constant.

<code>abs Rdest, Rsrc</code>	<i>Absolute Value</i> <sup>†</sup>
Put the absolute value of the integer from register <code>Rsrc</code> in register <code>Rdest</code> .	
<code>addu Rdest, Rsrc1, Src2</code>	<i>Addition</i>
<code>addiu Rdest, Rsrc1, Imm</code>	<i>Addition Immediate</i>
Put the sum of the integers from register <code>Rsrc1</code> and <code>Src2</code> (or <code>Imm</code> ) into register <code>Rdest</code> .	

---

<sup>1</sup>In earlier version of SPIM, `$sp` was documented as pointing at the first free word on the stack (not the last word of the stack frame). Recent MIPS documents have made it clear that this was an error. Both conventions work equally well, but we choose to follow the real system.

<sup>2</sup>The MIPS compiler does not use a frame pointer, so this register is used as callee-saved register `$s8`.

`and Rdest, Rsrc1, Src2` *AND*  
`andi Rdest, Rsrc1, Imm` *AND Immediate*  
Put the logical AND of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`div Rsrc1, Rsrc2` *Divide (signed)*  
`divu Rsrc1, Rsrc2` *Divide (unsigned)*  
Divide the contents of the two registers. `divu` treats its operands as unsigned values. 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 conventions of the machine on which SPIM is run.

`div Rdest, Rsrc1, Src2` *Divide (signed)* <sup>†</sup>  
`divu Rdest, Rsrc1, Src2` *Divide (unsigned)* <sup>†</sup>  
Put the quotient of the integers from register `Rsrc1` and `Src2` into register `Rdest`. `divu` treats its operands as unsigned values.

`mul Rdest, Rsrc1, Src2` *Multiply* <sup>†</sup>  
`mulou Rdest, Rsrc1, Src2` *Unsigned Multiply* <sup>†</sup>  
Put the product of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`mult Rsrc1, Rsrc2` *Multiply*  
`multu Rsrc1, Rsrc2` *Unsigned Multiply*  
Multiply the contents of the two registers. Leave the low-order word of the product in register `lo` and the high-word in register `hi`.

`negu Rdest, Rsrc` *Negate Value* <sup>†</sup>  
Put the negative of the integer from register `Rsrc` into register `Rdest`.

`nor Rdest, Rsrc1, Src2` *NOR*  
Put the logical NOR of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`not Rdest, Rsrc` *NOT* <sup>†</sup>  
Put the bitwise logical negation of the integer from register `Rsrc` into register `Rdest`.

`or Rdest, Rsrc1, Src2` *OR*  
`ori Rdest, Rsrc1, Imm` *OR Immediate*  
Put the logical OR of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`rem Rdest, Rsrc1, Src2` *Remainder* <sup>†</sup>  
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* <sup>†</sup>  
Put the remainder from dividing the integer in register `Rsrc1` by the integer in `Src2` into register `Rdest`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

`subu Rdest, Rsrc1, Src2` *Subtract*  
Put the difference of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`xor Rdest, Rsrc1, Src2` *XOR*  
`xori Rdest, Rsrc1, Imm` *XOR Immediate*  
Put the logical XOR of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

## 2.4 Constant-Manipulating Instructions

`li Rdest, imm`

*Load Immediate* <sup>†</sup>

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

## 2.5 Comparison Instructions

In all instructions below, `Src2` can either be a register or an immediate value (a 16 bit integer).

`seq Rdest, Rsrc1, Src2`

*Set Equal* <sup>†</sup>

Set register `Rdest` to 1 if register `Rsrc1` equals `Src2` and to 0 otherwise.

`sge Rdest, Rsrc1, Src2`

*Set Greater Than Equal* <sup>†</sup>

`sgeu Rdest, Rsrc1, Src2`

*Set Greater Than Equal Unsigned* <sup>†</sup>

Set register `Rdest` to 1 if register `Rsrc1` is greater than or equal to `Src2` and to 0 otherwise.

`sgt Rdest, Rsrc1, Src2`

*Set Greater Than* <sup>†</sup>

`sgtu Rdest, Rsrc1, Src2`

*Set Greater Than Unsigned* <sup>†</sup>

Set register `Rdest` to 1 if register `Rsrc1` is greater than `Src2` and to 0 otherwise.

`sle Rdest, Rsrc1, Src2`

*Set Less Than Equal* <sup>†</sup>

`sleu Rdest, Rsrc1, Src2`

*Set Less Than Equal Unsigned* <sup>†</sup>

Set register `Rdest` to 1 if register `Rsrc1` is less than or equal to `Src2` and to 0 otherwise.

`slt Rdest, Rsrc1, Src2`

*Set Less Than*

`slti Rdest, Rsrc1, Imm`

*Set Less Than Immediate*

`sltu Rdest, Rsrc1, Src2`

*Set Less Than Unsigned*

`sltiu Rdest, Rsrc1, Imm`

*Set Less Than Unsigned Immediate*

Set register `Rdest` to 1 if register `Rsrc1` is less than `Src2` (or `Imm`) and to 0 otherwise.

`sne Rdest, Rsrc1, Src2`

*Set Not Equal* <sup>†</sup>

Set register `Rdest` to 1 if register `Rsrc1` is not equal to `Src2` and to 0 otherwise.

## 2.6 Branch and Jump Instructions

In all instructions below, `Src2` can either be a register or an immediate value (integer). Branch instructions use a signed 16-bit offset field; hence they can jump  $2^{15} - 1$  instructions (not bytes) forward or  $2^{15}$  instructions backwards. The *jump* instruction contains a 26 bit address field.

`b label`

*Branch instruction* <sup>†</sup>

Unconditionally branch to the instruction at the label.

`beq Rsrc1, Src2, label`

*Branch on Equal*

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` equals `Src2`.

`beqz Rsrc, label`

*Branch on Equal Zero* <sup>†</sup>

Conditionally branch to the instruction at the label if the contents of `Rsrc` equals 0.

`bge Rsrc1, Src2, label`

*Branch on Greater Than Equal* <sup>†</sup>

`bgeu Rsrc1, Src2, label`

*Branch on GTE Unsigned* <sup>†</sup>



Conditionally branch to the instruction at the label if the contents of register **Rsrc1** are greater than or equal to **Src2**.

**bgez Rsrc, label** *Branch on Greater Than Equal Zero*

Conditionally branch to the instruction at the label if the contents of **Rsrc** are greater than or equal to 0.

**bgezal Rsrc, label** *Branch on Greater Than Equal Zero And Link*

Conditionally branch to the instruction at the label if the contents of **Rsrc** are greater than or equal to 0. Save the address of the next instruction in register 31.

**bgt Rsrc1, Src2, label** *Branch on Greater Than <sup>†</sup>*

**bgtu Rsrc1, Src2, label** *Branch on Greater Than Unsigned <sup>†</sup>*

Conditionally branch to the instruction at the label if the contents of register **Rsrc1** are greater than **Src2**.

**bgtz Rsrc, label** *Branch on Greater Than Zero*

Conditionally branch to the instruction at the label if the contents of **Rsrc** are greater than 0.

**ble Rsrc1, Src2, label** *Branch on Less Than Equal <sup>†</sup>*

**bleu Rsrc1, Src2, label** *Branch on LTE Unsigned <sup>†</sup>*

Conditionally branch to the instruction at the label if the contents of register **Rsrc1** are less than or equal to **Src2**.

**blez Rsrc, label** *Branch on Less Than Equal Zero*

Conditionally branch to the instruction at the label if the contents of **Rsrc** are less than or equal to 0.

**bgezal Rsrc, label** *Branch on Greater Than Equal Zero And Link*

**bltzal Rsrc, label** *Branch on Less Than And Link*

Conditionally branch to the instruction at the label if the contents of **Rsrc** are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

**blt Rsrc1, Src2, label** *Branch on Less Than <sup>†</sup>*

**bltu Rsrc1, Src2, label** *Branch on Less Than Unsigned <sup>†</sup>*

Conditionally branch to the instruction at the label if the contents of register **Rsrc1** are less than **Src2**.

**bltz Rsrc, label** *Branch on Less Than Zero*

Conditionally branch to the instruction at the label if the contents of **Rsrc** are less than 0.

**bne Rsrc1, Src2, label** *Branch on Not Equal*

Conditionally branch to the instruction at the label if the contents of register **Rsrc1** are not equal to **Src2**.

**bnez Rsrc, label** *Branch on Not Equal Zero <sup>†</sup>*

Conditionally branch to the instruction at the label if the contents of **Rsrc** are not equal to 0.

**j label** *Jump*

Unconditionally jump to the instruction at the label.

## 2.7 Load Instructions

<code>la Rdest, address</code>	<i>Load Address</i> <sup>†</sup>
Load computed <i>address</i> , not the contents of the location, into register <b>Rdest</b> .	
<code>lw Rdest, address</code>	<i>Load Word</i>
Load the 32-bit quantity (word) at <i>address</i> into register <b>Rdest</b> .	
<code>lwcx Rdest, address</code>	<i>Load Word Coprocessor</i>
Load the word at <i>address</i> into register <b>Rdest</b> of coprocessor <i>z</i> (0–3).	
<code>lwl Rdest, address</code>	<i>Load Word Left</i>
<code>lwr Rdest, address</code>	<i>Load Word Right</i>
Load the left (right) bytes from the word at the possibly-unaligned <i>address</i> into register <b>Rdest</b> .	

## 2.8 Store Instructions

<code>sw Rsrc, address</code>	<i>Store Word</i>
Store the word from register <b>Rsrc</b> at <i>address</i> .	
<code>swcx Rsrc, address</code>	<i>Store Word Coprocessor</i>
Store the word from register <b>Rsrc</b> of coprocessor <i>z</i> at <i>address</i> .	
<code>swl Rsrc, address</code>	<i>Store Word Left</i>
<code>swr Rsrc, address</code>	<i>Store Word Right</i>
Store the left (right) bytes from register <b>Rsrc</b> at the possibly-unaligned <i>address</i> .	

## 2.9 Data Movement Instructions

<code>move Rdest, Rsrc</code>	<i>Move</i> <sup>†</sup>
Move the contents of <b>Rsrc</b> to <b>Rdest</b> .	

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 instructions described above are pseudoinstructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.

<code>mfhi Rdest</code>	<i>Move From hi</i>
<code>mflo Rdest</code>	<i>Move From lo</i>
Move the contents of the <i>hi</i> ( <i>lo</i> ) register to register <b>Rdest</b> .	
<code>mthi Rdest</code>	<i>Move To hi</i>
<code>mtlo Rdest</code>	<i>Move To lo</i>
Move the contents register <b>Rdest</b> to the <i>hi</i> ( <i>lo</i> ) register.	

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

**mfcz Rdest, CPsrc** *Move From Coprocessor z*  
 Move the contents of coprocessor *z*'s register **CPsrc** to CPU register **Rdest**.

**mfc1.d Rdest, FRsrc1** *Move Double From Coprocessor 1* <sup>†</sup>  
 Move the contents of floating point registers **FRsrc1** and **FRsrc1 + 1** to CPU registers **Rdest** and **Rdest + 1**.

**mtcz Rsrc, CPdest** *Move To Coprocessor z*  
 Move the contents of CPU register **Rsrc** to coprocessor *z*'s register **CPdest**.

## 2.10 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. To simplify matters, floating point operations only use even-numbered registers—including instructions that operate on single floats.

Values are moved in or out of these registers a word (32-bits) at a time by **lwc1**, **swc1**, **mtc1**, and **mfc1** instructions described above or by the **l.s**, **l.d**, **s.s**, and **s.d** pseudoinstructions described below. The flag set by floating point comparison operations is read by the CPU with its **bc1t** and **bc1f** instructions.

In all instructions below, **FRdest**, **FRsrc1**, **FRsrc2**, and **FRsrc** are floating point registers (e.g., **\$f2**).

**abs.d FRdest, FRsrc** *Floating Point Absolute Value Double*  
**abs.s FRdest, FRsrc** *Floating Point Absolute Value Single*  
 Compute the absolute value of the floating float double (single) in register **FRsrc** and put it in register **FRdest**.

**add.d FRdest, FRsrc1, FRsrc2** *Floating Point Addition Double*  
**add.s FRdest, FRsrc1, FRsrc2** *Floating Point Addition Single*  
 Compute the sum of the floating float doubles (singles) in registers **FRsrc1** and **FRsrc2** and put it in register **FRdest**.

**c.eq.d FRsrc1, FRsrc2** *Compare Equal Double*  
**c.eq.s FRsrc1, FRsrc2** *Compare Equal Single*  
 Compare the floating point double in register **FRsrc1** against the one in **FRsrc2** and set the floating point condition flag true if they are equal.

**c.le.d FRsrc1, FRsrc2** *Compare Less Than Equal Double*  
**c.le.s FRsrc1, FRsrc2** *Compare Less Than Equal Single*  
 Compare the floating point double in register **FRsrc1** against the one in **FRsrc2** and set the floating point condition flag true if the first is less than or equal to the second.

**c.lt.d FRsrc1, FRsrc2** *Compare Less Than Double*  
**c.lt.s FRsrc1, FRsrc2** *Compare Less Than Single*  
 Compare the floating point double in register **FRsrc1** against the one in **FRsrc2** and set the condition flag true if the first is less than the second.

<code>cvt.d.s FRdest, FRsrc</code>	<i>Convert Single to Double</i>
<code>cvt.d.w FRdest, FRsrc</code>	<i>Convert Integer to Double</i>
Convert the single precision floating point number or integer in register <b>FRsrc</b> to a double precision number and put it in register <b>FRdest</b> .	
 <code>cvt.s.d FRdest, FRsrc</code>	 <i>Convert Double to Single</i>
<code>cvt.s.w FRdest, FRsrc</code>	<i>Convert Integer to Single</i>
Convert the double precision floating point number or integer in register <b>FRsrc</b> to a single precision number and put it in register <b>FRdest</b> .	
 <code>cvt.w.d FRdest, FRsrc</code>	 <i>Convert Double to Integer</i>
<code>cvt.w.s FRdest, FRsrc</code>	<i>Convert Single to Integer</i>
Convert the double or single precision floating point number in register <b>FRsrc</b> to an integer and put it in register <b>FRdest</b> .	
 <code>div.d FRdest, FRsrc1, FRsrc2</code>	 <i>Floating Point Divide Double</i>
<code>div.s FRdest, FRsrc1, FRsrc2</code>	<i>Floating Point Divide Single</i>
Compute the quotient of the floating float doubles (singles) in registers <b>FRsrc1</b> and <b>FRsrc2</b> and put it in register <b>FRdest</b> .	
 <code>l.d FRdest, address</code>	 <i>Load Floating Point Double <sup>†</sup></i>
<code>l.s FRdest, address</code>	<i>Load Floating Point Single <sup>†</sup></i>
Load the floating float double (single) at <b>address</b> into register <b>FRdest</b> .	
 <code>mov.d FRdest, FRsrc</code>	 <i>Move Floating Point Double</i>
<code>mov.s FRdest, FRsrc</code>	<i>Move Floating Point Single</i>
Move the floating float double (single) from register <b>FRsrc</b> to register <b>FRdest</b> .	
 <code>mul.d FRdest, FRsrc1, FRsrc2</code>	 <i>Floating Point Multiply Double</i>
<code>mul.s FRdest, FRsrc1, FRsrc2</code>	<i>Floating Point Multiply Single</i>
Compute the product of the floating float doubles (singles) in registers <b>FRsrc1</b> and <b>FRsrc2</b> and put it in register <b>FRdest</b> .	
 <code>neg.d FRdest, FRsrc</code>	 <i>Negate Double</i>
<code>neg.s FRdest, FRsrc</code>	<i>Negate Single</i>
Negate the floating point double (single) in register <b>FRsrc</b> and put it in register <b>FRdest</b> .	
 <code>s.d FRdest, address</code>	 <i>Store Floating Point Double <sup>†</sup></i>
<code>s.s FRdest, address</code>	<i>Store Floating Point Single <sup>†</sup></i>
Store the floating float double (single) in register <b>FRdest</b> at <b>address</b> .	
 <code>sub.d FRdest, FRsrc1, FRsrc2</code>	 <i>Floating Point Subtract Double</i>
<code>sub.s FRdest, FRsrc1, FRsrc2</code>	<i>Floating Point Subtract Single</i>
Compute the difference of the floating float doubles (singles) in registers <b>FRsrc1</b> and <b>FRsrc2</b> and put it in register <b>FRdest</b> .	