SPIM S20: A MIPS R2000 SIMULATOR

by

James R. Larus

Computer Sciences Technical Report #966

September 1990

SPIM S20: A MIPS R2000 Simulator [*]

"$\frac{1}{25}^{\text{th}}$ the performance at none of the cost"

James R. Larus
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706, USA
608-262-9519

September 5, 1990
(Revision 3)

1 Introduction

SPIM S20 is a simulator that runs programs for the MIPS R2000/R3000 RISC computers.[1] SPIM can read and immediately execute files containing assembly language or MIPS executable files. SPIM is a self-contained system for running these programs and contains a debugger and interface to a few operating system services.

The architecture of the MIPS computers is simple and regular, which makes it easy to learn and understand. The processor contains 32 general-purpose registers and a well-designed instruction set that make it a propitious target for generating code in a compiler.

However, the obvious question is: why use a simulator when many people have workstations that contain a hardware, and hence significantly faster, implementation of this computer? One reason is that these workstations are not available to most undergraduates since they are used for research. Another reason is that these machine will not persist for many years because of the rapid progress leading to new and faster computers. Unfortunately, the trend is to make computers faster by executing several instructions concurrently, which makes their architecture more difficult to understand and program. The MIPS architecture may be the epitome of a simple, clean RISC machine.

In addition, simulators can provide a better environment for low-level programming than an actual machine because they can detect more errors and provide more features than an actual computer. For example, SPIM has an X-window interface that is ahead of the debuggers for the actual machines.

[*]The students in CS536, Spring 1990, painfully found the last few bugs in an "already-debugged" simulator. I am grateful for their patience and persistence. Alan Yuen-wui Siow wrote the X-window interface, which David Wood debugged with malicious glee.

[1]For a description of the real machines, see Gerry Kane, *MIPS RISC Architecture,* Prentice Hall, 1989.

Finally, simulators are a useful tool for studying computers and the programs that run on them. Because they are implemented in software, not silicon, they can be easily modified to add new instructions, build new systems such as multiprocessors, or simply to collect data.

2 Simulation of a Virtual Machine

The MIPS architecture, like that of most RISC computers, is difficult to program directly because of its delayed branches and loads and restricted address modes. This difficulty is tolerable since these computers were designed to be programmed in high-level languages and so present an interface designed for compilers, not programmers. A *delayed branch* takes two cycles to execute. In the second cycle, the instruction immediately following the branch executes. This instruction can perform useful work that normally would have been done before the branch or it can be a `nop` (no operation). Similarly, *delayed loads* take two cycles so the instruction immediately following a load cannot use the value from memory.

MIPS wisely chose to hide this complexity by implementing a *virtual machine* with their assembler. This virtual computer appears to have non-delayed branches and loads and a richer instruction set than the actual hardware. The assembler *reorganizes* (rearranges) instructions to fill the delay slots. It also simulates the additional, or *pseudo*, instructions by generating short sequences of actual instructions.

By default, SPIM simulates the richer, virtual machine. It can also simulate the actual hardware. We will describe the virtual machine and only mention in passing features that do not belong to the actual hardware. In doing so, we are following the convention of MIPS assembly language programmers (and compilers), who routinely take advantage of the extended machine. Instructions marked with a dagger (†) are pseudo instructions.

3 SPIM Interface

SPIM provides both a simple terminal and a X-window interface. Both provide equivalent functionality, but the X interface is superior.

SPIM has the following command-line options:

`-bare`
: Simulate a bare MIPS machine without pseudo instructions or the additional addressing modes provided by the assembler. Implies `-quiet`.

`-asm`
: Simulate the virtual MIPS machine provided by the assembler. This is the default.

`-notrap`
: Do not load the standard trap handler. This trap handler has two functions that must be assumed by the user's program. First, it handles traps. When a trap occurs, SPIM jumps to location 0x80000080, which should contain code to service the exception. Second, this file contains startup code that invokes the routine `main`. Without the trap handler, execution begins at the instruction labeled `__start`.

`-trap`
: Load the standard trap handler. This is the default.

`-noquiet`
: Print a message when an exception occurs. This is the default.

`-quiet`
: Do not print a message at an exception.

`-file`
: Load and execute the assembly code in the file.

`-execute`
: Load and execute the code in the MIPS executable file *a.out*. The program cannot invoke any operating system services (e.g., input or output) since SPIM does not simulate the MIPS kernel traps.

3.1 Terminal Interface

The terminal interface provides the following commands:

`exit`
: Exit the simulator.

`read "file"`
: Read *file* of assembly language commands into SPIM's memory.

`load "file"`
: Synonym for `read`.

`execute "a.out"`
: Read the MIPS a.out executable *file* into SPIM's memory.

`run <addr>`
: Start running a program. If the optional address is provided, the program starts at that address. Otherwise, the program starts at the global symbol `__start`, which is defined by the default trap handler to invoke the routine at the global symbol `main`.

`step <N>`
: Step the program for N (default: 1) instructions. Print instructions as they execute.

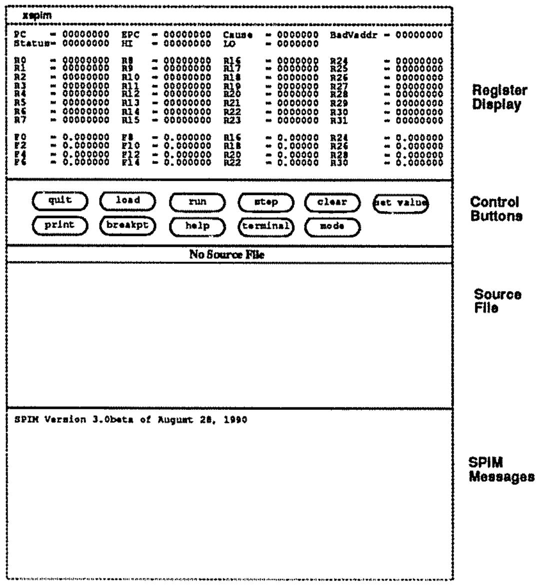
`continue`
: Continue program execution without stepping.

`print $N`
: Print register N.

`print $fN`
: Print floating point register N.

`print addr`
: Print the contents of memory at address $ADDR$.

`print_sym`
: Print the contents of the symbol table, i.e., the addresses of the global (but not local) symbols.



```
PC     = 00000000  EPC  = 00000000  Cause  = 0000000  BadVaddr = 00000000
Status= 00000000   HI   = 00000000  LO     = 0000000

R0     = 00000000  R8   = 00000000  R16    = 0000000  R24      = 00000000
R1     = 00000000  R9   = 00000000  R17    = 0000000  R25      = 00000000
R2     = 00000000  R10  = 00000000  R18    = 0000000  R26      = 00000000
R3     = 00000000  R11  = 00000000  R19    = 0000000  R27      = 00000000
R4     = 00000000  R12  = 00000000  R20    = 0000000  R28      = 00000000
R5     = 00000000  R13  = 00000000  R21    = 0000000  R29      = 00000000
R6     = 00000000  R14  = 00000000  R22    = 0000000  R30      = 00000000
R7     = 00000000  R15  = 00000000  R23    = 0000000  R31      = 00000000

F0     = 0.000000  F8   = 0.000000  R16    = 0.00000  R24      = 0.000000
F2     = 0.000000  F10  = 0.000000  R18    = 0.00000  R26      = 0.000000
F4     = 0.000000  F12  = 0.000000  R20    = 0.00000  R28      = 0.000000
F6     = 0.000000  F14  = 0.000000  R22    = 0.00000  R30      = 0.000000
```

```
SPIM Version 3.0beta of August 28, 1990
```

Figure 1: X-window interface to SPIM.

`reinitialize`
Clear the memory and registers.

`breakpoint addr`
Set a breakpoint at address *ADDR*.

`delete addr`
Delete all breakpoints at address *ADDR*.

`list`
List all breakpoints.

`.`
Rest of line is an assembly instruction that is stored in memory.

`<nl>`
A newline reexecutes previous command.

`?`
Print a help message.

Most commands can be abbreviated to their unique prefix e.g., `ex`, `re`, `l`, `ru`, `s`, `p`. More dangerous commands, such as `reinitialize`, require a longer prefix.

3.2 X-Window Interface

The X interface window has four panes (see Figure 1). The top pane displays the registers. It is continually updated, except while a program is running.

The next pane contains the buttons that control the simulator.

quit
Exit from the simulator.

load
Read a source or executable file into memory.

run
Start the program running.

step
Single-step through a program.

clear
Reinitialize registers or memory.

set value
Set the value in a register or memory location.

print
Print the value in a register or memory location.

breakpoint
Set or delete a breakpoint or list all breakpoints.

help
Print a help message.

terminals
Raise or hide terminal windows.

mode
Set SPIM operating modes.

The next pane displays the source file loaded into SPIM. Remember that the instructions displayed in this pane may include assembler pseudo instructions that expand into several actual MIPS instructions.

The bottom pane is used to display messages from the simulator. It does not contain output from an executing program. That appears in a separate window, called the Console, which pops up when the program produces output.

4 Surprising Features

Although SPIM faithfully simulates the MIPS computer, it is a simulator and certain things are not identical to the actual computer. The most prominent is that SPIM does not represent instructions as binary numbers, but rather uses a structured representation that is easier to interpret. This difference means that programs cannot write into memory and expect to execute an instruction or read from memory and decode an instruction. In fact, SPIM prevents programs from reading or writing the text segment or executing in the data segment.

Another surprise (which happens on the real machine also) is that a pseudo instruction expands into several machine instructions. When you single-step or examine memory, the instructions that you see will be slightly different from the source program.

5 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. 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` directive.

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

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

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

`.data`
Next data values should be stored in the data segment.

`.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.

Service	**Type Code**	**Arguments**
print_int	1	$a2 = integer
print_float	2	$f12 = float
print_double	3	$f12 = double
print_string	4	$a2 = string
read_int	5	
read_float	6	
read_string	7	$a2 = buffer, $a3 = length
sbrk	8	$a2 = amount
exit	9	

Table 1: System services.

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

`.ktext`
: The next items are put in the kernel text segment. In SPIM, these items must be instructions or words (see the `.word` directive below).

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

`.text`
: The next items are put in the user text segment. In SPIM, these items must be instructions or words (see the `.word` directive below).

`.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`).

6 System Calls

SPIM provides a small set of services through the system call (`syscall`) instruction. To request a service, a program loads the type code (see Table 1) into register `$a0` and the arguments into registers `$a1`...`$a3` (or `$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:    .asciz "the answer = "
        .text
        li $a0, 4         # print_str
        la $a1, str
        syscall
        li $a0, 1         # print_int
        li $a1, 5
        syscall
```

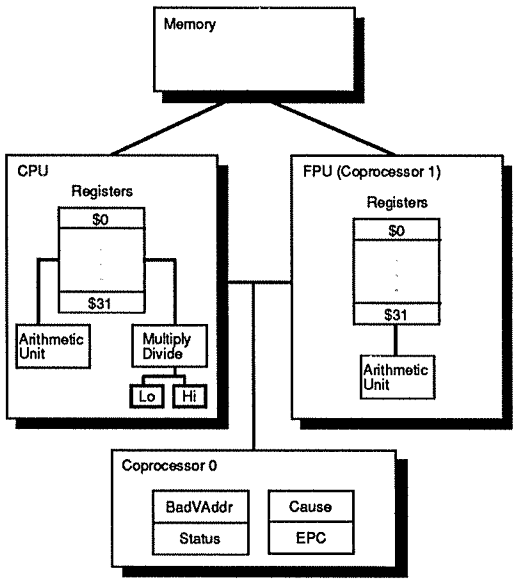


Figure 2: MIPS R2000 CPU and FPU

7 Description of the Machine

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

7.1 CPU Registers

The MIPS (and SPIM) central processing unit contains 32 general purpose registers that are numbered 0–31. Register *n* is designated by `$n` (though SPIM prints it as `Rn`). Register 0 always contains the hardwired value 0. Table 2 lists the registers and describes the convention governing their use.

In addition, coprocessor 0 contains registers that are used for exception handling. SPIM does not implement all of these registers, since they are not of much use in a simulator (or are part of the memory system). However, it does provide the following:

Register Name	**Number**	**Usage**
BadVAddr	8	Memory address at which address exception occurred
Status	12	Contains interrupt enable bits
Cause	13	Type of exception
EPC	14	Address of instruction that caused exception

These registers are part of coprocessor 0's register set and can be accessed by the `lwc0`, `mfc0`, `mtc0`, and `swc0` instructions.

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
s8	30	Saved temporary (preserved across call)
ra	31	Return address (used by function call)

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

7.2 Byte Order

Processors can number the bytes within a word to make the byte with the lowest numer either the leftmost or rightmost one. The convention used by a machine is its *byte order*. MIPS processors can operate with either *big-endian* byte order:

Byte #

0	1	2	3

or *little-endian* byte order:

Byte #

3	2	1	0

SPIM also operates with both byte orders. SPIM's byte order is determined by the byte order of the underlying hardware that is running the simulator. On a DECstation 3100, SPIM is little-endian, while on a Sun 4 or PC/RT, SPIM is big-endian.

7.3 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 ± immediate
symbol ± imm (register)	address of symbol ± (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 an even address and a full word object must be stored at an address that is a multiple of 4. However, MIPS provides some instructions for manipulating unaligned data.

7.4 Load and Store Instructions

`la Rdest, address` *Load Address* †
Load computed *address*, not the contents of the location, into register `Rdest`.

`lb Rdest, address` *Load Byte*
`lbu Rdest, address` *Load Unsigned Byte*
Load the byte at *address* into register `Rdest` (and sign-extend it).

`ld Rdest, address` *Load Double-Word* †
Load the 64-bit quantity at *address* into registers `Rdest` and `Rdest + 1`.

`lh Rdest, address` *Load Halfword*
`lhu Rdest, address` *Load Unsigned Halfword*
Load the 16-bit quantity (halfword) at *address* into register `Rdest` (and sign-extend it).

`lw Rdest, address` *Load Word*
Load the 32-bit quantity (word) at *address* into register `Rdest`.

`lwc`*z* `Rdest, address` *Load Word Coprocessor*
Load the word at *address* into register `Rdest` of coprocessor *z* (0–3).

`lwl Rdest, address` *Load Word Left*
`lwr Rdest, address` *Load Word Right*
Load the left (right) bytes from the word at the possibly-unaligned *address* into register `Rdest`.

`sb Rsource, address` *Store Byte*
Store the low byte from register `Rsource` at *address*.

`sd Rsource, address` *Store Double-Word* †
Store the 64-bit quantity in registers `Rsource` and `Rsource + 1` at *address*.

`sh Rsource, address` *Store Halfword*
Store the low halfword from register `Rsource` at *address*.

`sw Rsource, address` *Store Word*
Store the word from register `Rsource` at *address*.

`swc`*z* `Rsource, address` *Store Word Coprocessor*
Store the word from register `Rsource` of coprocessor *z* at *address*.

`swl Rsource, address` *Store Word Left*
`swr Rsource, address` *Store Word Right*
Store the left (right) bytes from register `Rsource` at the possibly-unaligned *address*.

`ulh Rdest, address` *Unaligned Load Halfword* †
`ulhu Rdest, address` *Unaligned Load Halfword Unsigned* †
Load the 16-bit quantity (halfword) at the possibly-unaligned *address* into register `Rdest` (and sign-extend it).

`ulw Rdest, address` *Unaligned Load Word* †
Load the 32-bit quantity (word) at the possibly-unaligned *address* into register `Rdest`.

`ush Rsource, address` *Unaligned Store Halfword* †
Store the low halfword from register `Rsource` at the possibly-unaligned *address*.

`usw Rsource, address` *Unaligned Store Word* †
Store the word from register `Rsource` at the possibly-unaligned *address*.

7.5 Exception and Trap Instructions

`rfe` *Return From Exception*
Restore the Status register.

`syscall` *System Call*
Register `a1` contains the number of the system call (see `spim-syscall.h`) provided by SPIM.

`break n` *Break*
Cause exception *n*. Exception 1 is reserved for the debugger.

`nop` *No operation*
Do nothing.

7.6 Constant-Manipulating Instructions

`li Rdest, imm` *Load Immediate* †
Move the immediate into register `Rdest`.

`li.d FRdest, float` *Load Immediate Double* †
Move the double-precision floating point number into floating point registers `FRdest` and `FRdest + 1`.

`li.s FRdest, float` *Load Immediate Single* †
Move the single-precision floating point number into floating point register `FRdest`.

`lui Rdest, integer` *Load Upper Immediate*
Load the lower halfword of the integer into the upper halfword of register `Rdest`. The lower bits of the register are set to 0.

7.7 Arithmetic and Logical Instructions

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

`abs Rdest, Rsource` *Absolute Value* †
Put the absolute value of the integer from register `Rsource` in register `Rdest`.

`add Rdest, Rsrc1, Src2` *Addition (with overflow)*
`addi Rdest, Rsrc1, Imm` *Addition Immediate (with overflow)*
`addu Rdest, Rsrc1, Src2` *Addition (without overflow)*
`addiu Rdest, Rsrc1, Imm` *Addition Immediate (without overflow)*
Put the sum of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`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 Rdest, Rsrc1, Src2` *Divide (with overflow)* †
`divu Rdest, Rsrc1, Src2` *Divide (without overflow)* †
Put the quotient of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`mul Rdest, Rsrc1, Src2` *Multiply (without overflow)* †
`mulo Rdest, Rsrc1, Src2` *Multiply (with overflow)* †

`mulou Rdest, Rsrc1, Src2` *Unsigned Multiply (with overflow)* †
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`.

`neg Rdest, Rsource` *Negate Value (with overflow)* †
`negu Rdest, Rsource` *Negate Value (without overflow)* †
Put the negative of the integer from register `Rsource` 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, Rsource` *NOT* †
Put the logical negation of the integer from register `Rsource` 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* †
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †
Put the remainder of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

`rol Rdest, Rsrc1, Src2` *Rotate Left* †
`ror Rdest, Rsrc1, Src2` *Rotate Right* †
Rotate the contents of register `Rsrc1` left (right) by the distance indicated by `Src2` and put the result in register `Rdest`.

`sll Rdest, Rsrc1, Src2` *Shift Left Logical*
`sllv Rdest, Rsrc1, Rsrc2` *Shift Left Logical Variable*
`sra Rdest, Rsrc1, Src2` *Shift Right Arithmetic*
`srav Rdest, Rsrc1, Rsrc2` *Shift Right Arithmetic Variable*
`srl Rdest, Rsrc1, Src2` *Shift Right Logical*
`srlv Rdest, Rsrc1, Rsrc2` *Shift Right Logical Variable*
Shift the contents of register `Rsrc1` left (right) by the distance indicated by `Src2` (`Rsrc2`) and put the result in register `Rdest`.

`sub Rdest, Rsrc1, Src2` *Subtract (with overflow)*
`subu Rdest, Rsrc1, Src2` *Subtract (without overflow)*
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`.

7.8 Comparison Instructions

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

`seq Rdest, Rsrc1, Src2` *Set Equal* †
Set register `Rdest` to 1 if register `Rsrc1` equals `Src2` and to be 0 otherwise.

`sge Rdest, Rsrc1, Src2` *Set Greater Than Equal* †
`sgeu Rdest, Rsrc1, Src2` *Set Greater Than Equal Unsigned* †
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* †
`sgtu Rdest, Rsrc1, Src2` *Set Greater Than Unsigned* †
Set register `Rdest` to 1 if register `Rsrc1` is greater than `Src2` and to 0 otherwise.

`sle Rdest, Rsrc1, Src2` *Set Less Than Equal* †
`sleu Rdest, Rsrc1, Src2` *Set Less Than Equal Unsigned* †
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* †
Set register `Rdest` to 1 if register `Rsrc1` is not equal to `Src2` and to 0 otherwise.

7.9 Branch and Jump Instructions

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

`b label` *Branch instruction* †
Unconditionally branch to the instruction at the label.

`bc`*z*`t label` *Branch Coprocessor z True*
`bc`*z*`f label` *Branch Coprocessor z False*
Conditionally branch to the instruction at the label if coprocessor *z*'s condition flag is true (false).

`beq Rsrc1, Src2, label` *Branch on Equal*
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` equals `Src2`.

`beqz Rsource, label` *Branch on Equal Zero* †
Conditionally branch to the instruction at the label if the contents of `Rsource` equals 0.

`bge Rsrc1, Src2, label` *Branch on Greater Than Equal* †
`bgeu Rsrc1, Src2, label` *Branch on GTE Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than or equal to `Src2`.

`bgez Rsource, label` *Branch on Greater Than Equal Zero*
Conditionally branch to the instruction at the label if the contents of `Rsource` are greater than or equal to 0.

`bgezal Rsource, label` *Branch on Greater Than Equal Zero And Link*
Conditionally branch to the instruction at the label if the contents of `Rsource` 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* †
`bgtu Rsrc1, Src2, label` *Branch on Greater Than Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than `Src2`.

`bgtz Rsource, label` *Branch on Greater Than Zero*
Conditionally branch to the instruction at the label if the contents of `Rsource` are greater than 0.

`ble Rsrc1, Src2, label` *Branch on Less Than Equal* †
`bleu Rsrc1, Src2, label` *Branch on LTE Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than or equal to `Src2`.

`blez Rsource, label` *Branch on Less Than Equal Zero*
Conditionally branch to the instruction at the label if the contents of `Rsource` are less than or equal to 0.

`blezal Rsource, label` *Branch on Less Than Equal Zero And Link*
`bltzal Rsource, label` *Branch on Less Than And Link*
Conditionally branch to the instruction at the label if the contents of `Rsource` are less than (or equal to) 0. Save the address of the next instruction in register 31.

`blt Rsrc1, Src2, label` *Branch on Less Than* †
`bltu Rsrc1, Src2, label` *Branch on Less Than Unsigned* †
Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than `Src2`.

`bltz Rsource, label` *Branch on Less Than Zero*
Conditionally branch to the instruction at the label if the contents of `Rsource` 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 Rsource, label` *Branch on Not Equal Zero* †
Conditionally branch to the instruction at the label if the contents of `Rsource` are not equal to 0.

`j label` *Jump*
Unconditionally jump to the instruction at the label.

`jal label` *Jump and Link*
`jalr Rsource` *Jump and Link Register*
Unconditionally jump to the instruction at the label or whose address is in register `Rsource`. Save the address of the next instruction in register 31.

`jr Rsource` *Jump Register*
Unconditionally jump to the instruction whose address is in register `Rsource`.

7.10 Data Movement Instructions

`move Rdest, Rsource` *Move*
Move the contents of `Rsource` to `Rdest`.

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

`mfhi Rdest` *Move From HI*
`mflo Rdest` *Move From LO*
Move the contents of the HI (LO) register to register `Rdest`.

`mthi Rdest` *Move To HI*
`mtlo Rdest` *Move To LO*
Move the contents register `Rdest` to the HI (LO) register.

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

`mfc`*z* `Rdest, Copsource` *Move From Coprocessor z*
Move the contents of coprocessor z's register `Copsource` to CPU register `Rdest`.

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

`mtc`*z* `Rsource, Copdest` *Move To Coprocessor z*
Move the contents of CPU register `Rsource` to coprocessor z's register `Copdest`.

7.11 Floating Point Unit

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 only use even-numbered registers—even 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` pseudo instructions described below.

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

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

`cvt.d.s FRdest, Rsource` *Convert Single to Double*
`cvt.d.w FRdest, Rsource` *Convert Integer to Double*
Convert the single precision floating point number or integer in register `Rsource` to a double precision number and put it in register `FRdest`.

`cvt.s.d FRdest, Rsource` *Convert Double to Single*
`cvt.s.w FRdest, Rsource` *Convert Integer to Single*
Convert the double precision floating point number or integer in register `Rsource` to a single precision number and put it in register `FRdest`.

`cvt.w.d FRdest, Rsource` *Convert Double to Integer*
`cvt.w.s FRdest, Rsource` *Convert Single to Integer*
Convert the double or single precision floating point number in register `Rsource` to an integer and put it in register `FRdest`.

`div.d FRdest, FRsrc1, FRsrc2` *Floating Point Divide Double*
`div.s FRdest, FRsrc1, FRsrc2` *Floating Point Divide Single*
Compute the quotient of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †
Load the floating float double (single) at `address` into register `FRdest`.

`mov.d FRdest, Rsource` *Move Floating Point Double*
`mov.s FRdest, Rsource` *Move Floating Point Single*
Move the floating float double (single) from register `Rsource` to register `FRdest`.

`mul.d FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Double*
`mul.s FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Single*
Compute the product of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`neg.d FRdest, Rsource` *Negate Double*
`neg.s FRdest, Rsource` *Negate Single*
Negate the floating point double (single) in register `Rsource` and put it in register `FRdest`.

`s.d FRdest, address` *Store Floating Point Double* †
`s.s FRdest, address` *Store Floating Point Single* †
Store the floating float double (single) in register `FRdest` at `address`.

`sub.d FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Double*
`sub.s FRdest, FRsrc1, FRsrc2` *Floating Point Subtract Single*
Compute the difference of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.