

Getting Started with SPIM

The rest of this appendix contains a complete and rather detailed description of SPIM. Many details should never concern you; however, the sheer volume of information can obscure the fact that SPIM is a simple, easy-to-use program. This section contains a quick tutorial on SPIM that should enable you to load, debug, and run simple MIPS programs.

SPIM comes in multiple versions. One version, called *spim*, is a command-line-driven program and requires only an alphanumeric terminal to display it. It operates like most programs of this type: you type a line of text, hit the enter key, and *spim* executes your command.

A fancier version, called *xspim*, runs in the X-windows environment of the Unix system and therefore requires a bit-mapped display to run it. *xspim*, however, is a much easier program to learn and use because its commands are always visible on the screen and because it continually displays the machine's registers. Another version, *PCSpim*, is compatible with Windows 3.1, Windows 95, and Windows NT. The Unix, Windows, and DOS versions of SPIM are available through www.mkp.com/cod2e.htm. This section of the document describes *PCSpim*, the Windows version of SPIM under Windows 95.

Installation and Graphic Interface Description

To install the Windows version of SPIM, you can download the installation file, *spimwin.exe*, through www.mkp.com/cod2e.htm. Execute the installation file and follow the installation procedure. In Windows 95, you can simply activate the icon associated with the file just like any Windows program, or you can select *Start->Run* and type in the directory path and filename. The installation program will execute and inform you when the installation process is complete. When the installation is complete, a group folder with executable file, help files, and uninstaller program is created.

To start *PCSpim* for Windows, you simply activate the icon labeled *PCSpim for Windows* like any other Windows program. For example, in Windows 95, you can use select *Start->Programs->PCSpim for Windows ->PCSpim for Windows* from the Windows 95 task bar. In Windows 3.1, you can select the application from the File Manager.

When *PCSpim* starts up, it brings up a large window on your screen (see Figure A.1). The application window is divided into four parts:

- The top section is the menu bar. The menu bar allows you to select *File* operations, set *Simulator* settings, select *Windows* views, and obtain online *Help* information.
- The next section below the menu bar is the toolbar. The toolbar provides quick mouse access to many tools used in *PCSpim* for Windows.
- The large section in the middle of the application window is the window display section. There are four display windows: *Registers*, *Text Segment*, *Data Segment*, and *Messages*. To change the view of these four windows, you can select a tiled view from the menu bar: *Windows->Tile*. All of the display windows will be empty when you first execute the program. The following list describes each display window.
 - The Register window display shows the values of all registers in the MIPS CPU and FPU.
 - The Text Segment window display shows instructions both from your program and the system code that is loaded automatically when *PCSpim* is running.
 - The Data Segment window display shows the data loaded into your program's memory and the data of the program's stack.
 - The Messages window display is the where *PCSpim* uses to write messages. This is where error messages appear.

When selected, PCSpim for Windows will record the position of its windows when you exit, and restore them to the same location the next time you run PCSpim.

- **Bare machine**

When selected, you can simulate a bare MIPS machine without pseudoinstructions or the additional addressing modes provided by the assembler.

- **Allow pseudo instructions**

If this setting is selected, pseudoinstructions are allowed in your program; otherwise, if the setting is not selected, they are not allowed.

- **Load trap file**

If this setting is selected, the standard exception handler and start-up code is loaded. When an exception occurs, SPIM jumps to location 80000080hex, which must contain code to service the exception. In addition, the trap handler contains start-up code that invokes the routine main. Without the start-up routine, SPIM begins execution at the instruction labeled __start. The default trap file comes with PCSpim, but you can choose another using Browse button.

- **Mapped I/O**

If this setting is selected, the memory-mapped I/O facility is enabled. Programs that use SPIM syscalls to read from the terminal cannot also use memory-mapped I/O.

- **Quiet**

When this setting is enabled, PCSpim does not print a message at exceptions; otherwise, a message is printed when an exception occurs.

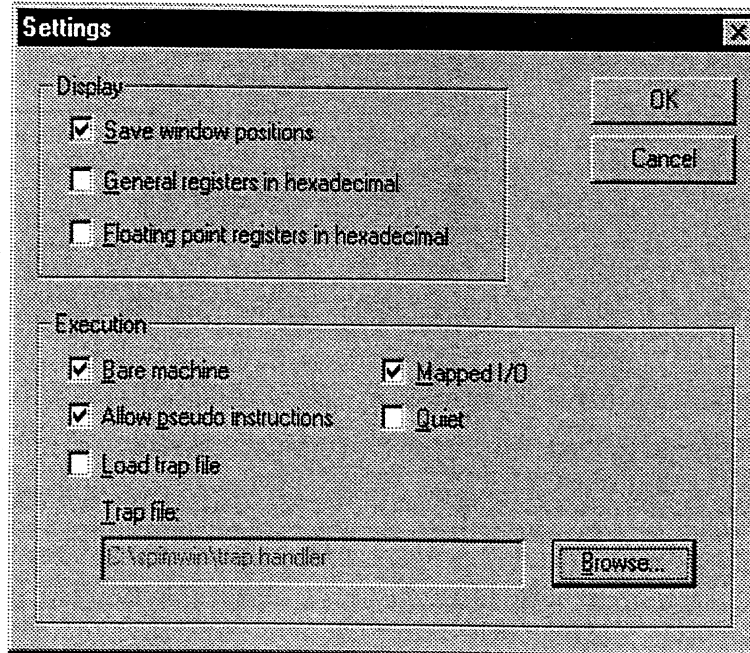


Figure A.2. PCSpim simulator setting dialog box

PCSpim performs other functions that are occasionally useful. When you are more comfortable with PCSpim, you should look at the description in the online help to see what they do and how they can save you time and effort. You can view the online help available with the simulator by selecting *Help->Help_topics* from the menu bar.

Simulator Setting

PCSpim has a graphical interface to view the current setting of the simulator (see FigureA.2). When you start PCSpim, you do not have to enter with any command line parameters. However, you should check your simulator settings either on PCSpim's status bar or the simulator setting dialog box before you load your program. To view or change PCSpim settings in the simulator setting dialog box, select *Simulator->Settings* from the menu bar.

It is very important to set the simulator in the correct setting for your program. PCSpim determines how to load and how your program executes from these settings, so an incorrect setting may cause errors when you run your program. If the simulator setting is incorrect and the program is unable to load correctly, PCSpim allows you to change the simulator settings and reload your program. If you want to change PCSpim settings after you load your program, you should reload your program by selecting *Simulator->Reload* from the menu bar.

The following paragraphs describes the operation of each of the settings in the simulator setting dialog box shown in Figure A.2. Most of the functions are similar to SPIM, its counterparts in the terminal interface version without the graphical interface.

- **Display**

You can select to view the register contents in decimal or hexadecimal notation. If the check boxes for general registers or floating point registers are selected, a check mark will appear and the register contents will be displayed in hexadecimal notation.

- **Save window positions**

```
[0x00400000] 0x8fa40000 lw $4, 0($29); 89: lw $a0, 0($sp)
```

The first number on the line, in square brackets, is the hexadecimal memory address of the instruction. The second number is the instruction's numerical encoding, again displayed as a hexadecimal number. The third item is the instruction's mnemonic description. Everything following the semicolon is the actual line from your assembly file that produced the instruction. The number 89 is the line number in that file. Sometimes nothing is on the line after the semicolon. This means that the instruction was produced by SPIM as part of translating a pseudoinstruction.

To run your program, click on the *Go* button in the toolbar. Alternatively, you can select *Simulator->Go* from the menu bar. Your program will begin execution. If you want to stop the execution of your program, select *Simulator->Break* from the menu bar. Alternatively, you can type *Control-C* when PCSpim application window is in focus. A dialog box will appear and ask if you want to continue execution. Select *No* to break the execution. Before doing anything, you can look at memory and registers contents in the Register display window to find out what your program was doing. When you understand what happened, you can either continue the program by selecting *Simulator->Continue* or stop your program by selecting *Simulator->Break* from the menu bar.

If your program reads or writes from the terminal, PCSpim pops up another window called the console. All characters that your program writes appear on the console, and everything that you type as input to your program should be typed in this window.

Suppose your program does not do what you expect. What can you do? SPIM has two features that help debug your program. The first, and perhaps the most useful, is single-stepping, which allows you to run your program an instruction at a time. Select *Simulator->Single_Step* to execute only one instruction. Alternatively, you can press the F10 function key to single step. Each time you step through a program, PCSpim will execute the next instruction in your program, updates the display, and returns control to you. You can also choose the number of instructions in your program to step by selecting *Simulator->Multiple_Step* instead of single stepping through your program. A dialog box will appear and ask you the number of instructions to step.

What do you do if your program runs for a long time before the bug arises? You could single-step until you get to the bug, but that can take a long time, and it is easy to get so bored and inattentive that you step past the problem. A better alternative is to use a breakpoint, which tells PCSpim to stop your program immediately before it executes a particular instruction. Select *Simulator->Breakpoints* from the menu bar. The PCSpim program pops up a dialog box window with two boxes. The top box is for you to enter breakpoint address and the second box is a list of active breakpoints. Type in the first box the address of the instruction at which you want to stop. Or, if the instruction has a global label, you can just type the name of the label. Labeled breakpoints are a particularly convenient way to stop at the first instruction of a procedure. To actually set the breakpoint, and click on the button labeled *Add*. When you are done adding breakpoints, click on the button labeled *Close*. You can then run your program.

When the simulator is about to execute the breakpointed instruction, PCSpim pops up a dialog box with the instruction's address and asks if you want to continue the execution. The *Yes* button continues running your program and the *No* button stops your program. If you want to delete a breakpoint, you can select *Simulator->Breakpoints* from the menu bar, click on the address in the dialog box, and click on the button labeled *Remove*.

Single-stepping and setting breakpoints will probably help you find a bug in your program quickly. How do you fix it? Go back to the editor that you used to create your program and change your source file. After you have made the changes to your source file, simply reload it into PCSpim for Windows by choosing *Simulator->Reload<filename>* from the menu bar. This causes PCSpim to clear its memory and registers and return the processor to the state it was in when PCSpim first started. Once the simulator has reinitialized itself, it will reload your recently modified file.

- asm Simulate the virtual MIPS machine provided by the assembler. This is the default.
- pseudo Allow the input assembly code to contain pseudoinstructions. This is the default.
- nopseudo Do not allow pseudoinstructions in the input assembly code.
- notrap Do not load the standard exception handler and start-up code. This exception handler handles exceptions. When an exception occurs, SPIM jumps to location 80000080_{hex}, which must contain code to service the exception. In addition, this file contains start-up code that invokes the routine `main`. Without the start-up routine, SPIM begins execution at the instruction labeled `__start`.
- trap Load the standard exception handler and start-up code. This is the default.
- noquiet Print a message when an exception occurs. This is the default.
- quiet Do not print a message at exceptions.
- nomapped_io Disable the memory-mapped I/O facility. This is the default.
- mapped_io Enable the memory-mapped I/O facility. Programs that use SPIM syscalls to read from the terminal *cannot* also use memory-mapped I/O.
- file Load and execute the assembly code in the file.
- execute Load and execute the code in the MIPS executable file *a.out*. This command is only available when SPIM runs on a system containing a MIPS processor.
- s <seg> size Sets the initial size of memory segment *seg* to be *size* bytes. The memory segments are named: `text`, `data`, `stack`, `ktext`, and `kdata`. The `text` segment contains instructions from a program. The `data` segment holds the program's data. The `stack` segment holds its runtime stack. In addition to running a program, SPIM also executes system code that handles interrupts and exceptions. This code resides in a separate part of the address space called the *kernel*. The `ktext` segment holds this code's instructions, and `kdata` holds its data. There is no `kstack` segment since the system code uses the same stack as the program. For example, the pair of arguments `-sdata 2000000` starts the user data segment at 2,000,000 bytes.
- l <seg> size Sets the limit on how large memory segment *seg* can grow to be *size* bytes. The memory segments that can grow are `data`, `stack`, and `kdata`.

Loading and Running a Program

Let's see how to load and run a program. The first thing to do is to select the open file icon from the toolbar. Alternatively, you can select from the menu bar: *File->Open*. A file open dialog box will appear for you to select the appropriate assembly file. Select the appropriate assembly file and click on the button labeled *Open* in the dialog box. If simulator settings are not correct for the file, and it fails to load, PCSpim will provide you an opportunity to change simulator settings and automatically reload the file.

If you change your mind, click on the button labeled *Cancel*, and PCSpim removes the dialog box. When you load an assembly file, PCSpim removes dialog box, then loads your program and redraws the screen to display its instructions and data. If you have not done so, change the view of the four display windows to a tiled format by selecting from the menu bar: *Windows->Tile*. You should be able to see the program in the Text segment window display.

Each instruction in the Text segment window display is shown on a line that looks like

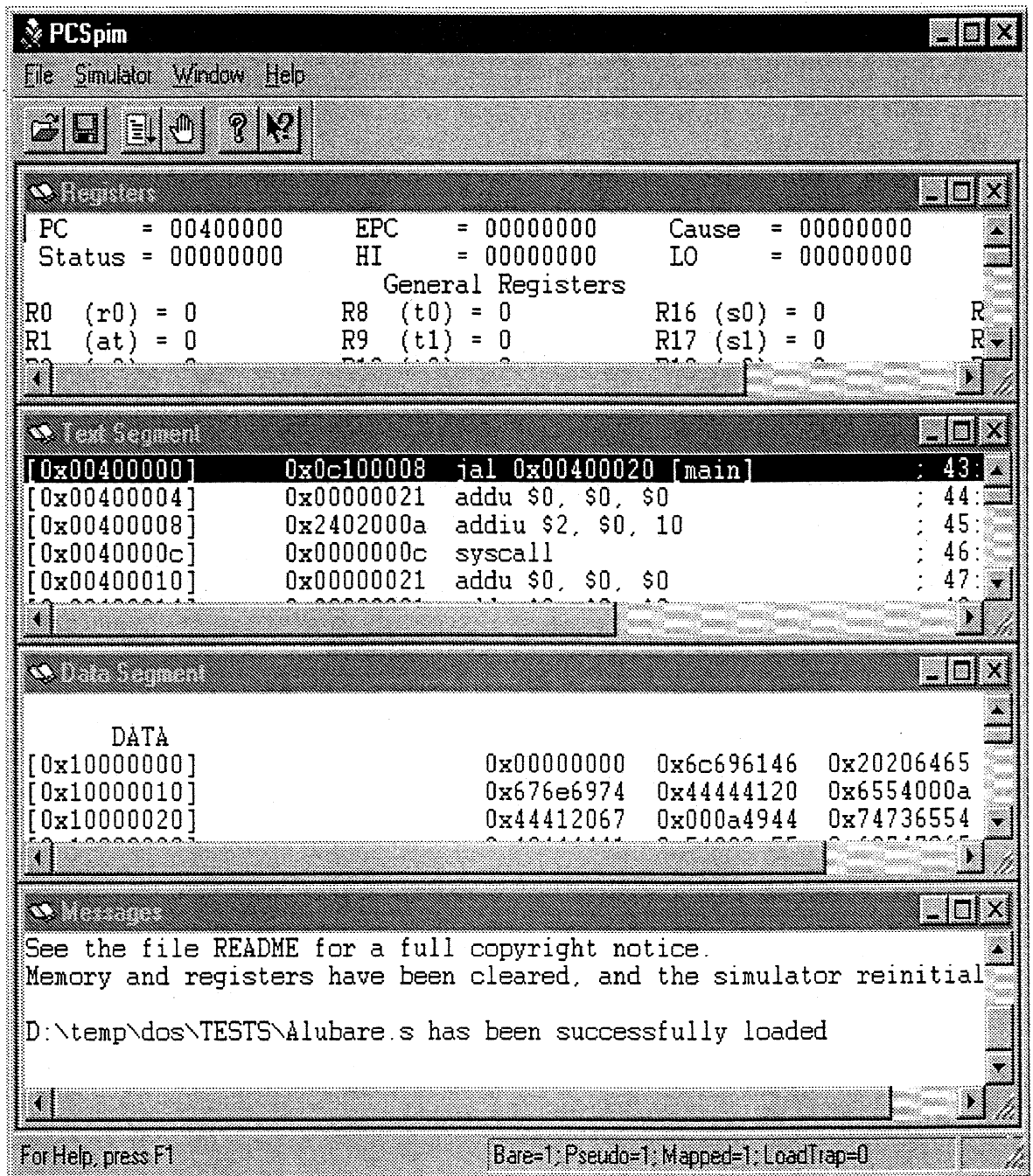


Figure A.1. PCSpim's window interface

- The Status bar section is at the bottom of the application window. The status bar provides information and the current settings of the simulator.

SPIM Command-Line Options

The Windows version of SPIM accepts the following command-line options:

- bare Simulate a bare MIPS machine without pseudoinstructions or the additional addressing modes provided by the assembler. Implies quiet.

SPIM S20: A MIPS R2000 Simulator*

“ $\frac{1}{25}$ 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

Copyright ©1990–1997 by James R. Larus
(This document may be copied without royalties,
so long as this copyright notice remains on it.)

1 SPIM

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. 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 32-bit 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 generally available. 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 a X-window interface that is better than most debuggers for the actual machines.

*I grateful to the many students at UW who used SPIM in their courses and happily found bugs in a professor's code. In particular, 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.

¹For a description of the real machines, see Gerry Kane and Joe Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.

Finally, simulators are an 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.

1.1 Simulation of a Virtual Machine

The MIPS architecture, like that of most RISC computers, is difficult to program directly because of its delayed branches, delayed 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 good part of the complexity results from delayed instructions. 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 loaded from memory.

MIPS wisely choose 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, *pseudoinstructions* 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 pseudoinstructions.

1.2 SPIM Interface

SPIM provides a simple terminal and a X-window interface. Both provide equivalent functionality, but the X interface is generally easier to use and more informative.

`spim`, the terminal version, and `xspim`, the X version, have the following command-line options:

-bare

Simulate a bare MIPS machine without pseudoinstructions 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.

-pseudo

Accept pseudoinstructions in assembly code.

-nopseudo

Do not accept pseudoinstructions in assembly code.

-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.

-trap_file

Load the trap handler in the file.

-noquiet

Print a message when an exception occurs. This is the default.

-quiet

Do not print a message at an exception.

-nomapped_io

Disable the memory-mapped IO facility (see Section 5).

-mapped_io

Enable the memory-mapped IO facility (see Section 5). Programs that use SPIM syscalls (see Section 1.5) to read from the terminal should not also use memory-mapped IO.

-file

Load and execute the assembly code in the file.

-s seg size Sets the initial size of memory segment *seg* to be *size* bytes. The memory segments are named: `text`, `data`, `stack`, `ktext`, and `kdata`. For example, the pair of arguments `-sdata 2000000` starts the user data segment at 2,000,000 bytes.

-lseg size Sets the limit on how large memory segment *seg* can grow to be *size* bytes. The memory segments that can grow are: `data`, `stack`, and `kdata`.

1.2.1 Terminal Interface

The terminal interface (`spim`) provides the following commands:

exit

Exit the simulator.

read "file"

Read *file* of assembly language commands into SPIM's memory. If the file has already been read into SPIM, the system should be cleared (see `reinitialize`, below) or global symbols will be multiply defined.

load "file"

Synonym for `read`.

run <addr>

Start running a program. If the optional address *addr* 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 call the routine at the global symbol `main` with the usual MIPS calling convention.

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.

reinitialize
 Clear the memory and registers.

breakpoint addr
 Set a breakpoint at address *addr*. *addr* can be either a memory address or symbolic label.

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.

1.2.2 X-Window Interface

The X version of SPIM, *xspim*, looks different, but should operate in the same manner as *spim*. The X window has five panes (see Figure 1). The top pane displays the contents of 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.

xspim

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

General Registers

R0 (r0) = 00000000	R8 (t0) = 00000000	R16 (s0) = 00000000	R24 (t8) = 00000000
R1 (at) = 00000000	R9 (t1) = 00000000	R17 (s1) = 00000000	R25 (s9) = 00000000
R2 (v0) = 00000000	R10 (t2) = 00000000	R18 (s2) = 00000000	R26 (k0) = 00000000
R3 (v1) = 00000000	R11 (t3) = 00000000	R19 (s3) = 00000000	R27 (k1) = 00000000
R4 (a0) = 00000000	R12 (t4) = 00000000	R20 (s4) = 00000000	R28 (gp) = 00000000
R5 (a1) = 00000000	R13 (t5) = 00000000	R21 (s5) = 00000000	R29 (gp) = 00000000
R6 (a2) = 00000000	R14 (t6) = 00000000	R22 (s6) = 00000000	R30 (s8) = 00000000
R7 (a3) = 00000000	R15 (t7) = 00000000	R23 (s7) = 00000000	R31 (ra) = 00000000

Double Floating Point Registers

FP0 = 0.000000	FP8 = 0.000000	FP16 = 0.000000	FP24 = 0.000000
FP2 = 0.000000	FP10 = 0.000000	FP18 = 0.000000	FP26 = 0.000000
FP4 = 0.000000	FP12 = 0.000000	FP20 = 0.000000	FP28 = 0.000000
FP6 = 0.000000	FP14 = 0.000000	FP22 = 0.000000	FP30 = 0.000000

Single Floating Point Registers

quit
load
run
step
clear
set value

print
breakpt
help
terminal
mode

Text Segments

```
[0x00400000] 0x8fa40000 lw R4, 0(R29) []
[0x00400004] 0x27a50004 addiu R5, R29, 4 []
[0x00400008] 0x24a60004 addiu R6, R5, 4 []
[0x0040000c] 0x00041090 sll R2, R4, 2
[0x00400010] 0x00c23021 addu R5, R5, R2
[0x00400014] 0x0c000000 jal 0x00000000 []
[0x00400018] 0x3402000a ori R0, R0, 10 []
[0x0040001c] 0x0000000c syscall
```

Data Segments

```
[0x10000000] ..[0x10010000] 0x00000000
[0x10010004] 0x74706563 0x206e6f69 0x636f2000
[0x10010010] 0x72727563 0x61206465 0x6920646e 0x726f6e67
[0x10010020] 0x000a6465 0x495b2020 0x7265746e 0x74707572
[0x10010030] 0x0000205d 0x20200000 0x616e555b 0x6e67696c
[0x10010040] 0x61206465 0x65726464 0x69207373 0x6e69206e
[0x10010050] 0x642f7473 0x20617461 0x63746566 0x00205d68
[0x10010060] 0x555b2020 0x696c616e 0x64656e67 0x64646120
[0x10010070] 0x73736572 0x206e6920 0x726f7473 0x00205d65
```

SPIM Version 3.2 of January 14, 1990

SPIM Messages

Figure 1: X-window interface to SPIM.

load

Read a source 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.

terminal

Raise or hide the console window.

mode

Set SPIM operating modes.

The next two panes display the memory contents. The top one shows instructions from the user and kernel text segments.² The first few instructions in the text segment are startup code (`_start`) that loads `argc` and `argv` into registers and invokes the main routine.

The lower of these two panes displays the data and stack segments. Both panes are updated as a program executes.

The bottom pane is used to display messages from the simulator. It does not display output from an executing program. When a program reads or writes, its IO appears in a separate window, called the Console, which pops up when needed.

1.3 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 obvious differences are that instruction timing and the memory systems are not identical. SPIM does not simulate caches or memory latency, nor does it accurately reflect the delays for floating point operations or multiplies and divides.

Another surprise (which occurs on the real machine as well) is that a pseudoinstruction expands into several machine instructions. When single-stepping or examining memory, the instructions that you see are slightly different from the source program. The correspondence between the two sets of instructions is fairly simple since SPIM does not reorganize the instructions to fill delay slots.

²These instructions are real—not pseudo—MIPS instructions. SPIM translates assembler pseudoinstructions to 1–3 MIPS instructions before storing the program in memory. Each source instruction appears as a comment on the first instruction to which it is translated.

1.4 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.

`.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.

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
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		

Table 1: System services.

`.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*.

`.space n`

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

`.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`).

1.5 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
```

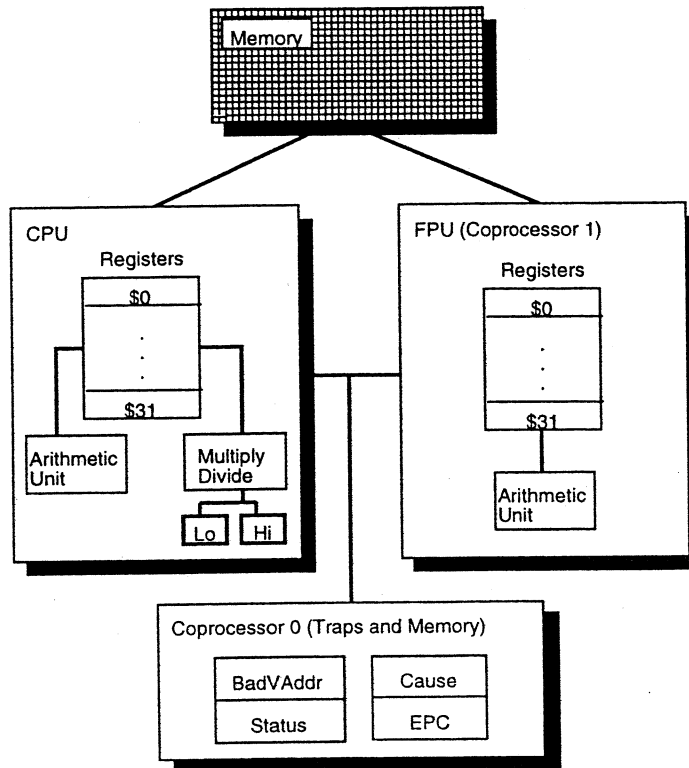


Figure 2: MIPS R2000 CPU and FPU

```

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
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.

`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. **Warning:** programs that use these syscalls to read from the terminal should not use memory-mapped IO (see Section 5).

`sbrk` returns a pointer to a block of memory containing n additional bytes. `exit` stops a program from running.

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	30	Frame pointer
ra	31	Return address (used by function call)

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

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 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.

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

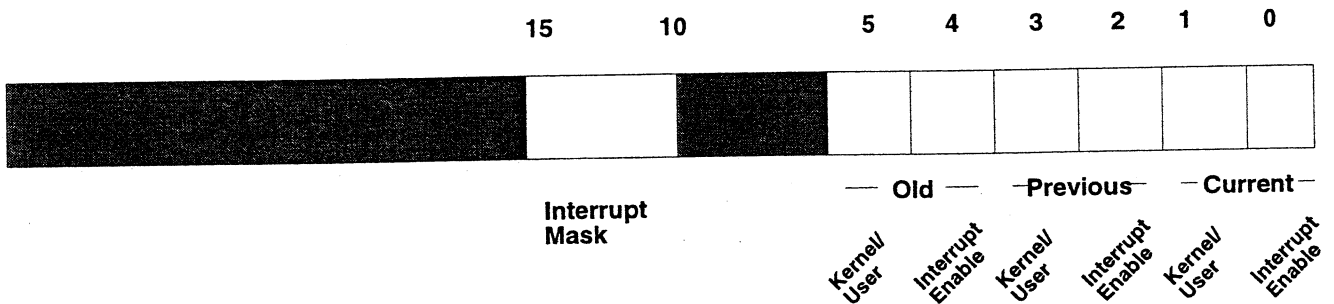


Figure 3: The Status register.

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.³ Register \$fp (30) is the frame pointer.⁴ 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.

In addition, coprocessor 0 contains registers that are useful to handle exceptions. 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, which is not implemented. However, it does provide the following:

Register Name	Number	Usage
BadVAddr	8	Memory address at which address exception occurred
Status	12	Interrupt mask and enable bits
Cause	13	Exception type and pending interrupt bits
EPC	14	Address of instruction that caused exception

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

Figure 3 describes the bits in the Status register that are implemented by SPIM. The interrupt mask contains a bit for each of the five interrupt levels. If a bit is one, interrupts at that level are allowed. If the bit is zero, interrupts at that level are disabled. The low six bits of the Status register implement a three-level stack for the kernel/user and interrupt enable bits. The kernel/user bit is 0 if the program was running in the kernel when the interrupt occurred and 1 if it was in user mode. If the interrupt enable bit is 1, interrupts are allowed.

³In 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.

⁴The MIPS compiler does not use a frame pointer, so this register is used as callee-saved register \$s8.

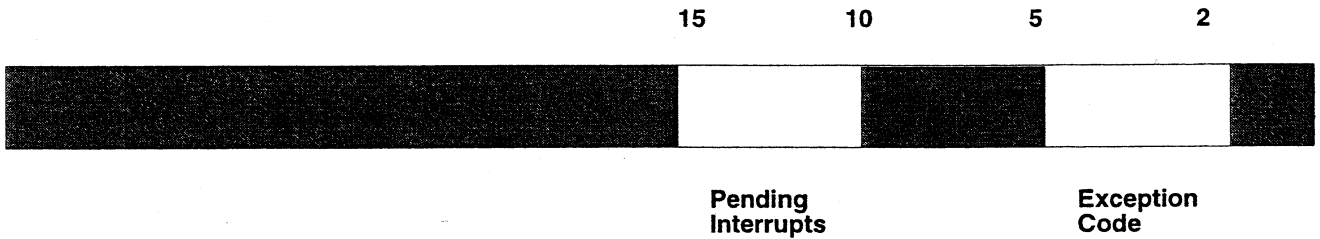


Figure 4: The Cause register.

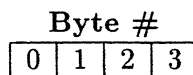
If it is 0, they are disabled. At an interrupt, these six bits are shifted left by two bits, so the current bits become the previous bits and the previous bits become the old bits. The current bits are both set to 0 (i.e., kernel mode with interrupts disabled).

Figure 4 describes the bits in the Cause registers. The five pending interrupt bits correspond to the five interrupt levels. A bit becomes 1 when an interrupt at its level has occurred but has not been serviced. The exception code register contains a code from the following table describing the cause of an exception.

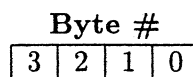
Number	Name	Description
0	INT	External interrupt
4	ADDRL	Address error exception (load or instruction fetch)
5	ADDRS	Address error exception (store)
6	IBUS	Bus error on instruction fetch
7	DBUS	Bus error on data load or store
8	SYSCALL	Syscall exception
9	BKPT	Breakpoint exception
10	RI	Reserved instruction exception
12	OVF	Arithmetic overflow exception

2.2 Byte Order

Processors can number the bytes within a word to make the byte with the lowest number 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:



or *little-endian* byte order:



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

2.3 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 + or - immediate
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.

2.4 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.

abs Rdest, Rsrc *Absolute Value* †
Put the absolute value of the integer from register *Rsrc* 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 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, with overflow)* †
divu Rdest, Rsrc1, Src2 *Divide (unsigned, without overflow)* †
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 (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*, *Rsrc* *Negate Value (with overflow)* †
negu *Rdest*, *Rsrc* *Negate Value (without overflow)* †
 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* †
 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* †
remu *Rdest*, *Rsrc1*, *Src2* *Unsigned Remainder* †
 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.

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*.

2.5 Constant-Manipulating Instructions

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

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

2.6 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* †
Set register `Rdest` to 1 if register `Rsrc1` equals `Src2` and to 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.

2.7 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* †
Unconditionally branch to the instruction at the label.

`bczt label` *Branch Coprocessor z True*
`bczf 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 Rsrc, label *Branch on Equal Zero* †
 Conditionally branch to the instruction at the label if the contents of Rsrc 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 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* †
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 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* †
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 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* †
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 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 †*
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.

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

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

2.8 Load 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`. The byte is sign-extended by the `lb`, but not the `lbu`, instruction.

`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`. The halfword is sign-extended by the `lh`, but not the `lhu`, instruction

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

`lwcz 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`.

`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*. The halfword is sign-extended by the *ulh*, but not the *ulhu*, instruction

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

2.9 Store Instructions

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

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

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

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

swcz Rsrc, address *Store Word Coprocessor*
Store the word from register *Rsrc* of coprocessor *z* at *address*.

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

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

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

2.10 Data Movement Instructions

move Rdest, Rsrc *Move †*
Move the contents of *Rsrc* 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 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.

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.

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 †

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.

2.11 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.