# 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



## 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.

[1]For 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.

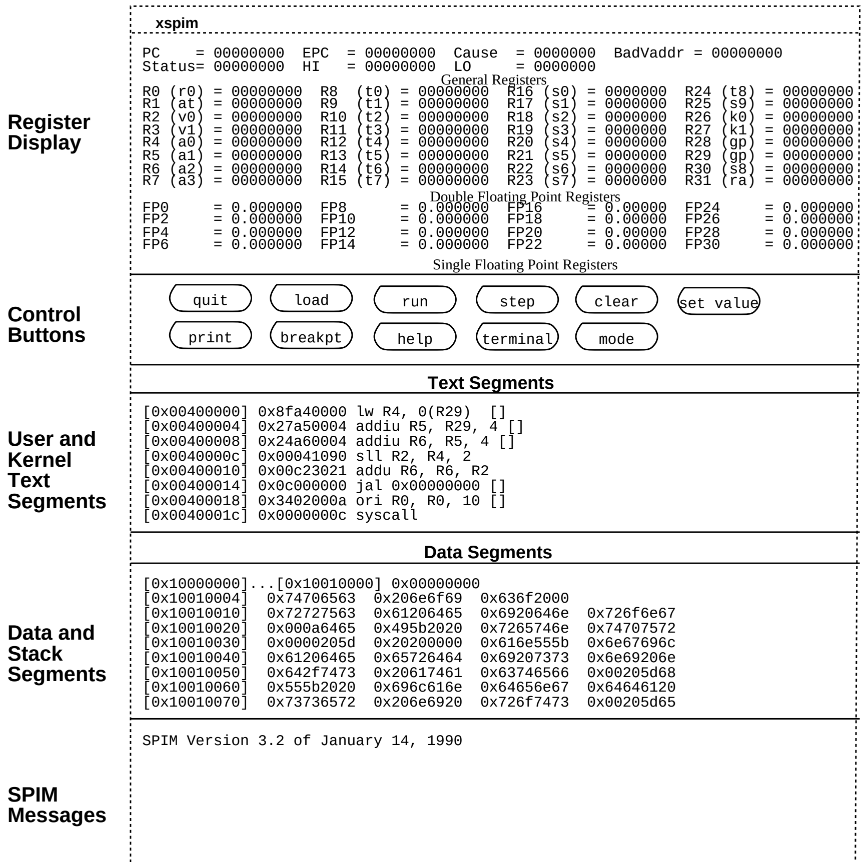


Figure 1: X-window interface to SPIM.

**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 accurate 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.

[2] These 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 `$v0`) |
| exit | 10 | | |
| print_character | 11 | `$a0` = integer | |
| read_character | 12 | char (in `$v0`) | |

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
        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.

# 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 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 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.

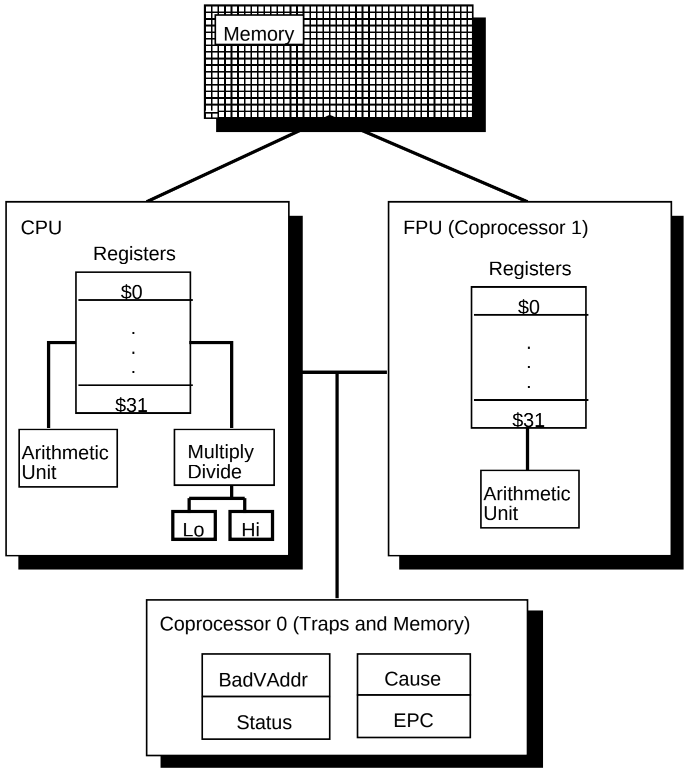


Figure 2: MIPS R2000 CPU and FPU

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

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

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