SPIM S20: A MIPS R2000 Simulator*

"\(\frac{1}{32}\) of 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, 1991 by James R. Larus
Dec. 23, 1991
(Revision 9 corresponding to SPIM Version 4.4)

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.

---

1 I am 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, 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, the terminal version, and xspim, the X version, have 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.

-ssiz 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.

-li 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.

3.1 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.

  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 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.

3.2 X-Window Interface

The X version of SPIM, xspim, looks different, but should function the same 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.

    load
    Read a source or executable file into memory.

    run
    Start the program running.

    step
    Single-step through a program.
Figure 1: X-window interface to SPIM.
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 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.

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 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 pseudo instruction 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 (which it only simulates in bare mode).

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.

\(^2\)These instructions are real—not pseudo—MIPS instructions. SPIM translates assembler pseudo instructions to 1–3 MIPS instructions before storing the program in memory. Although the instructions in memory look different from the source program, the translation is straightforward and can be understood with a bit of practice.
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:

```plaintext
.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:

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

SPIM supports a subset of the assembler directives provided by the MIPS assembler:

```plaintext
.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.

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

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

.data
The following data items 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.

.half h1, ..., hn
Store the n 16-bit quantities in successive memory halfwords.
<table>
<thead>
<tr>
<th>Service</th>
<th>Type Code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 = integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12 = float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12 = double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 = string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>float</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td>double</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 = buffer, $a1 = length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>address</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: System services.

.kdata
The following data items should be stored in the kernel data segment.

.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 operating-system-like services through the system call (syscall) instruction. To request a service, a program loads the type code (see Table 1) into register $v0 and the arguments into registers $a0...$a3 (or $f12 for floating point values).³ 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: .ascii "the answer = "
.text
li $v0, 4       # print_str
la $a0, str
```

³In earlier versions of SPIM (before 4.0), the type code was passed in register $a0 and the values were slightly different. Programs written for the older versions of SPIM need to be modified to reflect these changes.
syscall
li $v0, 1  # print_int
li $a0, 5
syscall

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 in 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.

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

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

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

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\. Register \$0\ always contains the hardwired value 0. MIPS has established a set of conventions how the registers should be used. These suggestions are guidelines, which are not enforced by the hardware. However a program that violates them will not work properly with other software. Table 2 lists the registers and describes their intended use.

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

Registers \$a0–\$a3\ (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers \$v0\ and \$v1\ (2, 3) are used to return values from functions. Registers \$t0–\$t9\ (8–15, 24, 25) are caller-saved registers used for temporary
quantities that do not need to be preserved across calls. Registers $s0$–$s7$ (16–23) are callee-
saved registers that hold long-lived values that should be preserved across calls.

Register $sp$ (29) is the stack pointer, which points to the first free location on the stack. Register $fp$ (30) is the frame pointer. 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 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:

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadVAddr</td>
<td>8</td>
<td>Memory address at which address exception occurred</td>
</tr>
<tr>
<td>Status</td>
<td>12</td>
<td>Contains interrupt enable bits</td>
</tr>
<tr>
<td>Cause</td>
<td>13</td>
<td>Type of exception</td>
</tr>
<tr>
<td>EPC</td>
<td>14</td>
<td>Address of instruction that caused exception</td>
</tr>
</tbody>
</table>

These registers are part of coprocessor 0's register set and can be accessed by the lwz, mfz, mtz, and swz instructions.

7.2 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:

```
        0 1 2 3
       --------------------
Byte #  0 1 2 3
```

or little-endian byte order:

```
       3 2 1 0
       --------------------
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 HP Bobcat, 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:

---

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

11
### 7.4 Load and Store Instructions

**la Rdest, address**

Load computed address, not the contents of the location, into register Rdest.

**lb Rdest, address**

Load the byte at address into register Rdest (and sign-extend it).

**ld Rdest, address**

Load the 64-bit quantity at address into registers Rdest and Rdest + 1.

**lh Rdest, address**

Load the 16-bit quantity (halfword) at address into register Rdest (and sign-extend it).

**lw Rdest, address**

Load the 32-bit quantity (word) at address into register Rdest.

**lwz Rdest, address**

Load the word at address into register Rdest of coprocessor z (0–3).

**lwl Rdest, address**

Load the left (right) bytes from the word at the possibly-unaligned address into register Rdest.

**sb Rsource, address**

Store the low byte from register Rsource at address.

**sd Rsource, address**

Store the 64-bit quantity in registers Rsource and Rsource + 1 at address.

**sh Rsource, address**

Store the low halfword from register Rsource at address.

**sw Rsource, address**

Store the word from register Rsource at address.
\texttt{swcz \textit{Rsource}, address} \hspace{1.9cm} \textit{Store Word Coprocessor}

Store the word from register \textit{Rsource} of coprocessor \texttt{z} at \textit{address}.

\texttt{swl \textit{Rsource}, address} \hspace{1.9cm} \textit{Store Word Left}

\texttt{swr \textit{Rsource}, address} \hspace{1.9cm} \textit{Store Word Right}

Store the left (right) bytes from register \textit{Rsource} at the possibly-unaligned \textit{address}.

\texttt{ulh \textit{Rdest}, address} \hspace{1.9cm} \textit{Unaligned Load Halfword} \footnote{1}

\texttt{ulhu \textit{Rdest}, address} \hspace{1.9cm} \textit{Unaligned Load Halfword Unsigned} \footnote{1}

Load the 16-bit quantity (halfword) at the possibly-unaligned \textit{address} into register \textit{Rdest} (and sign-extend it).

\texttt{ulw \textit{Rdest}, address} \hspace{1.9cm} \textit{Unaligned Load Word} \footnote{1}

Load the 32-bit quantity (word) at the possibly-unaligned \textit{address} into register \textit{Rdest}.

\texttt{ush \textit{Rsource}, address} \hspace{1.9cm} \textit{Unaligned Store Halfword} \footnote{1}

Store the low halfword from register \textit{Rsource} at the possibly-unaligned \textit{address}.

\texttt{usw \textit{Rsource}, address} \hspace{1.9cm} \textit{Unaligned Store Word} \footnote{1}

Store the word from register \textit{Rsource} at the possibly-unaligned \textit{address}.

### 7.5 Exception and Trap Instructions

\texttt{rfe} \hspace{1.9cm} \textit{Return From Exception}

Restore the Status register.

\texttt{syscall} \hspace{1.9cm} \textit{System Call}

Register $\texttt{v0}$ contains the number of the system call (see Table 1) provided by SPIM.

\texttt{break \texttt{n}} \hspace{1.9cm} \textit{Break}

Cause exception \texttt{n}. Exception 1 is reserved for the debugger.

\texttt{nop} \hspace{1.9cm} \textit{No operation}

Do nothing.

### 7.6 Constant-Manipulating Instructions

\texttt{li \textit{Rdest}, imm} \hspace{1.9cm} \textit{Load Immediate} \footnote{1}

Move the immediate into register \textit{Rdest}.

\texttt{li.d \textit{FRdest}, float} \hspace{1.9cm} \textit{Load Immediate Double} \footnote{1}

Move the double-precision floating point number into floating point registers \textit{FRdest} and \textit{FRdest + 1}.

\texttt{li.s \textit{FRdest}, float} \hspace{1.9cm} \textit{Load Immediate Single} \footnote{1}

Move the single-precision floating point number into floating point register \textit{FRdest}.

\texttt{lui \textit{Rdest}, integer} \hspace{1.9cm} \textit{Load Upper Immediate}

Load the lower halfword of the integer into the upper halfword of register \textit{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.

\textit{abs Rdest, Rsource} \hspace{2cm} \textit{Absolute Value} ^1
Put the absolute value of the integer from register \texttt{Rsource} in register \texttt{Rdest}.

\textit{add Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{Addition (with overflow)}
\textit{addi Rdest, Rsrlc1, Imm} \hspace{2cm} \textit{Addition Immediate (with overflow)}
\textit{addu Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{Addition (without overflow)}
\textit{addiu Rdest, Rsrlc1, Imm} \hspace{2cm} \textit{Addition Immediate (without overflow)}
Put the sum of the integers from register \texttt{Rsrlc1} and Src2 (or \texttt{Imm}) into register \texttt{Rdest}.

\textit{and Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{AND}
\textit{andi Rdest, Rsrlc1, Imm} \hspace{2cm} \textit{AND Immediate}
Put the logical AND of the integers from register \texttt{Rsrlc1} and Src2 (or \texttt{Imm}) into register \texttt{Rdest}.

\textit{div Rsrlc1, Rsrlc2} \hspace{2cm} \textit{Divide (with overflow)} ^1
\textit{divu Rsrlc1, Rsrlc2} \hspace{2cm} \textit{Divide (without overflow)} ^1
Divide the contents of the two registers. Leave the quotient in register L0 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.

\textit{div Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{Divide (with overflow)} ^1
\textit{divu Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{Divide (without overflow)} ^1
Put the quotient of the integers from register \texttt{Rsrlc1} and Src2 into register \texttt{Rdest}.

\textit{mul Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{Multiply (without overflow)} ^1
\textit{mulu Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{Multiply (with overflow)} ^1
\textit{mulul Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{Unsigned Multiply (with overflow)} ^1
Put the product of the integers from register \texttt{Rsrlc1} and Src2 into register \texttt{Rdest}.

\textit{mult Rsrlc1, Rsrlc2} \hspace{2cm} \textit{Multiply}
\textit{multu Rsrlc1, Rsrlc2} \hspace{2cm} \textit{Unsigned Multiply}
Multiply the contents of the two registers. Leave the low-order word of the product in register L0 and the high-word in register HI.

\textit{neg Rdest, Rsource} \hspace{2cm} \textit{Negate Value (with overflow)} ^1
\textit{negu Rdest, Rsource} \hspace{2cm} \textit{Negate Value (without overflow)} ^1
Put the negative of the integer from register \texttt{Rsource} into register \texttt{Rdest}.

\textit{nor Rdest, Rsrlc1, Src2} \hspace{2cm} \textit{NOR}
Put the logical NOR of the integers from register \texttt{Rsrlc1} and Src2 into register \texttt{Rdest}.

\textit{not Rdest, Rsource} \hspace{2cm} \textit{NOT} ^1
Put the logical negation of the integer from register \texttt{Rsource} into register \texttt{Rdest}.
or $R_{dest}$, $R_{src1}$, $ Src2$

OR

ori $R_{dest}$, $R_{src1}$, $ Imm$

OR Immediate

Put the logical OR of the integers from register $R_{src1}$ and $Src2$ (or $Imm$) into register $R_{dest}$.

rem $R_{dest}$, $R_{src1}$, $ Src2$

Remainder

remu $R_{dest}$, $R_{src1}$, $ Src2$

Unsigned Remainder

Put the remainder from dividing the integer in register $R_{src1}$ by the integer in $Src2$ into register $R_{dest}$. 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 $R_{dest}$, $R_{src1}$, $ Src2$

Rotate Left

ror $R_{dest}$, $R_{src1}$, $ Src2$

Rotate Right

Rotate the contents of register $R_{src1}$ left (right) by the distance indicated by $Src2$ and put the result in register $R_{dest}$.

sll $R_{dest}$, $R_{src1}$, $ Src2$

Shift Left Logical

sllv $R_{dest}$, $R_{src1}$, $R_{src2}$

Shift Left Logical Variable

sra $R_{dest}$, $R_{src1}$, $ Src2$

Shift Right Arithmetic

srav $R_{dest}$, $R_{src1}$, $ Src2$

Shift Right Variable

srl $R_{dest}$, $R_{src1}$, $ Src2$

Shift Right Logical

srli $R_{dest}$, $R_{src1}$, $ Src2$

Shift Right Logical Variable

Shift the contents of register $R_{src1}$ left (right) by the distance indicated by $Src2$ ($R_{src2}$) and put the result in register $R_{dest}$.

sub $R_{dest}$, $R_{src1}$, $ Src2$

Subtract (with overflow)

subu $R_{dest}$, $R_{src1}$, $ Src2$

Subtract (without overflow)

Put the difference of the integers from register $R_{src1}$ and $Src2$ into register $R_{dest}$.

xor $R_{dest}$, $R_{src1}$, $ Src2$

XOR

xori $R_{dest}$, $R_{src1}$, $ Imm$

XOR Immediate

Put the logical XOR of the integers from register $R_{src1}$ and $Src2$ (or $Imm$) into register $R_{dest}$.

7.8 Comparison Instructions

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

seq $R_{dest}$, $R_{src1}$, $ Src2$

Set Equal

Set register $R_{dest}$ to 1 if register $R_{src1}$ equals $Src2$ and to 0 otherwise.

sgge $R_{dest}$, $R_{src1}$, $ Src2$

Set Greater Than Equal

sggeu $R_{dest}$, $R_{src1}$, $ Src2$

Set Greater Than Equal Unsigned

Set register $R_{dest}$ to 1 if register $R_{src1}$ is greater than or equal to $Src2$ and to 0 otherwise.

sgt $R_{dest}$, $R_{src1}$, $ Src2$

Set Greater Than

sgtu $R_{dest}$, $R_{src1}$, $ Src2$

Set Greater Than Unsigned

Set register $R_{dest}$ to 1 if register $R_{src1}$ is greater than $Src2$ and to 0 otherwise.

sle $R_{dest}$, $R_{src1}$, $ Src2$

Set Less Than Equal

sleu $R_{dest}$, $R_{src1}$, $ Src2$

Set Less Than Equal Unsigned

Set register $R_{dest}$ to 1 if register $R_{src1}$ is less than or equal to $Src2$ and to 0 otherwise.
\texttt{slt Rdest, Rs\textsubscript{src}1, Src2} \hspace{1cm} \text{Set Less Than}
\texttt{slti Rdest, Rs\textsubscript{src}1, Imm} \hspace{1cm} \text{Set Less Than Immediate}
\texttt{sltu Rdest, Rs\textsubscript{src}1, Src2} \hspace{1cm} \text{Set Less Than Unsigned}
\texttt{sltiu Rdest, Rs\textsubscript{src}1, Imm} \hspace{1cm} \text{Set Less Than Unsigned Immediate}

Set register \texttt{Rdest} to 1 if register \texttt{Rs\textsubscript{src}1} is less than \texttt{Src2} (or \texttt{Imm}) and to 0 otherwise.

\texttt{sne Rdest, Rs\textsubscript{src}1, Src2} \hspace{1cm} \text{Set Not Equal} \textsuperscript{1}

Set register \texttt{Rdest} to 1 if register \texttt{Rs\textsubscript{src}1} is not equal to \texttt{Src2} and to 0 otherwise.

\section*{7.9 Branch and Jump Instructions}

In all instructions below, \texttt{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 \textit{jump} instruction contains a 26 bit address field.

\texttt{b label} \hspace{1cm} \text{Branch instruction} \textsuperscript{1}

Unconditionally branch to the instruction at the label.

\texttt{bczt label} \hspace{1cm} \text{Branch Coprocessor z True}
\texttt{bczf label} \hspace{1cm} \text{Branch Coprocessor z False}

Conditionally branch to the instruction at the label if coprocessor \texttt{z}'s condition flag is true (false).

\texttt{beg Rs\textsubscript{src}1, Src2, label} \hspace{1cm} \text{Branch on Equal}

Conditionally branch to the instruction at the label if the contents of register \texttt{Rs\textsubscript{src}1} equals \texttt{Src2}.

\texttt{begz Rs\textsubscript{source}, label} \hspace{1cm} \text{Branch on Equal Zero} \textsuperscript{1}

Conditionally branch to the instruction at the label if the contents of \texttt{Rs\textsubscript{source}} equals 0.

\texttt{bge Rs\textsubscript{src}1, Src2, label} \hspace{1cm} \text{Branch on Greater Than Equal} \textsuperscript{1}
\texttt{bgeu Rs\textsubscript{src}1, Src2, label} \hspace{1cm} \text{Branch on GTE Unsigned} \textsuperscript{1}

Conditionally branch to the instruction at the label if the contents of register \texttt{Rs\textsubscript{src}1} are greater than or equal to \texttt{Src2}.

\texttt{bgez Rs\textsubscript{source}, label} \hspace{1cm} \text{Branch on Greater Than Equal Zero}

Conditionally branch to the instruction at the label if the contents of \texttt{Rs\textsubscript{source}} are greater than or equal to 0.

\texttt{bgezal Rs\textsubscript{source}, label} \hspace{1cm} \text{Branch on Greater Than Equal Zero And Link}

Conditionally branch to the instruction at the label if the contents of \texttt{Rs\textsubscript{source}} are greater than or equal to 0. Save the address of the next instruction in register 31.

\texttt{bgt Rs\textsubscript{src}1, Src2, label} \hspace{1cm} \text{Branch on Greater Than} \textsuperscript{1}
\texttt{bgtu Rs\textsubscript{src}1, Src2, label} \hspace{1cm} \text{Branch on Greater Than Unsigned} \textsuperscript{1}

Conditionally branch to the instruction at the label if the contents of register \texttt{Rs\textsubscript{src}1} are greater than \texttt{Src2}.

\texttt{bgtz Rs\textsubscript{source}, label} \hspace{1cm} \text{Branch on Greater Than Zero}

Conditionally branch to the instruction at the label if the contents of \texttt{Rs\textsubscript{source}} are greater than 0.
ble Rs
bleu Rs
Conditionally branch to the instruction at the label if the contents of register Rs are less than or equal to S.

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

bgezal Rs, label Branch on Greater Than Equal Zero And Link
bgtzal Rs, label Branch on Less Than And Link
Conditionally branch to the instruction at the label if the contents of Rs are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

blt Rs, S Branch on Less Than
bltu Rs, S, label Branch on Less Than Unsigned
Conditionally branch to the instruction at the label if the contents of register Rs are less than S.

bltz Rs, label Branch on Less Than Zero
Conditionally branch to the instruction at the label if the contents of Rs are less than 0.

bne Rs, S, label Branch on Not Equal
Conditionally branch to the instruction at the label if the contents of register Rs are not equal to S.

bnez Rs, label Branch on Not Equal Zero
Conditionally branch to the instruction at the label if the contents of Rs are not equal to 0.

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

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

jr Rs, Jump Register
Unconditionally jump to the instruction whose address is in register Rs.

7.10 Data Movement Instructions

move Rs, Rd Move
Move the contents of Rs to Rd.

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.

\[
\begin{align*}
\text{mfhi \textit{Rdest}} & \quad \text{Move From HI} \\
\text{mflo \textit{Rdest}} & \quad \text{Move From LO} \\
\end{align*}
\]

Move the contents of the HI (LO) register to register \textit{Rdest}.

\[
\begin{align*}
\text{mthi \textit{Rdest}} & \quad \text{Move To HI} \\
\text{mtlo \textit{Rdest}} & \quad \text{Move To LO} \\
\end{align*}
\]

Move the contents register \textit{Rdest} to the HI (LO) register.

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

\[
\begin{align*}
\text{mfcz \textit{Rdest, Copsource}} & \quad \text{Move From Coprocessor } z \\
\end{align*}
\]

Move the contents of coprocessor \textit{z}’s register \textit{Copsource} to CPU register \textit{Rdest}.

\[
\begin{align*}
\text{mfc1.d \textit{Rdest, FRsrc1}} & \quad \text{Move Double From Coprocessor } 1 \\
\end{align*}
\]

Move the contents of floating point registers \textit{FRsrc1} and \textit{FRsrc1 + 1} to CPU registers \textit{Rdest} and \textit{Rdest + 1}.

\[
\begin{align*}
\text{mtcz \textit{Rsource, Copdest}} & \quad \text{Move To Coprocessor } z \\
\end{align*}
\]

Move the contents of CPU register \textit{Rsource} to coprocessor \textit{z}’s register \textit{Copdest}.

### 7.11 Floating Point

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 \textit{lwc1, swc1, mtc1}, and \textit{mfcl} instructions described above or by the \textit{l.s, l.d, s.s, and s.d} pseudo instructions described below. The flag set by floating point comparison operations is read by the CPU with its \textit{bcit} and \textit{bcif} instructions.

In all instructions below, \textit{FRdest, FRsrc1, FRsrc2}, and \textit{FRSource} are floating point registers (e.g., $f2$).

\[
\begin{align*}
\text{abs.d \textit{FRdest, FRsource}} & \quad \text{Floating Point Absolute Value Double} \\
\text{abs.s \textit{FRdest, FRsource}} & \quad \text{Floating Point Absolute Value Single} \\
\end{align*}
\]

Compute the absolute value of the floating float double (single) in register \textit{FRsource} and put it in register \textit{FRdest}.

\[
\begin{align*}
\text{add.d \textit{FRdest, FRsrc1, FRsrc2}} & \quad \text{Floating Point Addition Double} \\
\text{add.s \textit{FRdest, FRsrc1, FRsrc2}} & \quad \text{Floating Point Addition Single} \\
\end{align*}
\]

Compute the sum of the floating float doubles (singles) in registers \textit{FRsrc1} and \textit{FRsrc2} and put it in register \textit{FRdest}.

\[
\begin{align*}
\text{c.eq.d \textit{FRsrc1, FRsrc2}} & \quad \text{Compare Equal Double} \\
\text{c.eq.s \textit{FRsrc1, FRsrc2}} & \quad \text{Compare Equal Single} \\
\end{align*}
\]
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.

cut.d.s FRdest, FRsource
Convert Single to Double

cut.d.w FRdest, FRsource
Convert Integer to Double

Convert the single precision floating point number or integer in register FRsource to a double precision number and put it in register FRdest.

cut.s.d FRdest, FRsource
Convert Double to Single

cut.s.w FRdest, FRsource
Convert Integer to Single

Convert the double precision floating point number or integer in register FRsource to a single precision number and put it in register FRdest.

cut.w.d FRdest, FRsource
Convert Double to Integer

cut.w.s FRdest, FRsource
Convert Single to Integer

Convert the double or single precision floating point number in register FRsource 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 ¹

l.s FRdest, address
Load Floating Point Single ¹

Load the floating float double (single) at address into register FRdest.

mov.d FRdest, FRsource
Move Floating Point Double

mov.s FRdest, FRsource
Move Floating Point Single

Move the floating float double (single) from register FRsource 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, FRsource
Negate Double

neg.s FRdest, FRsource
Negate Single

Negate the floating point double (single) in register FRsource and put it in register FRdest.
Figure 3: Layout of memory.

\begin{verbatim}
\texttt{s.d} FRdest, address
\texttt{s.s} FRdest, address
\end{verbatim}

Store the floating float double (single) in register FRdest at address.

\begin{verbatim}
\texttt{sub.d} FRdest, FRsrc1, FRsrc2
\texttt{sub.s} FRdest, FRsrc1, FRsrc2
\end{verbatim}

Compute the difference of the floating float doubles (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

8 Memory Usage

The organization of memory in MIPS systems is conventional. A program’s address space is composed of three parts (see Figure 3).

At the bottom of the user address space (0x400000) is the text segment, which holds the instructions for a program.

Above the text segment is the data segment (starting at 0x1000000), which is divided into two parts. The static data portion contains objects whose size and address are known to the compiler and linker. Immediately above these objects is dynamic data. As a program allocates space dynamically (i.e., by malloc), the sbrk system call moves the top of the data segment up.

At the top of the address space (0xffffffff) is the program stack, which grows down, towards the data segment.

9 Calling Convention

The calling convention described in this section is the one used by gcc, not the native MIPS compiler, which uses a more complex convention that is slightly faster.

Figure 4 shows a diagram of a stack frame. A frame consists of the memory between the frame pointer ($fp$), which points to the word immediately after the last argument passed on the stack, and the stack pointer ($sp$), which points to the first free word on the stack. As typical of Unix systems, the stack grows down from higher memory addresses, so the frame pointer is above stack pointer.

The following steps are necessary to effect a call:
Figure 4: Layout of a stack frame. The frame pointer points just below the last argument passed on the stack. The stack pointer points to the first word after the frame.

1. Pass the arguments. By convention, the first four arguments are passed in registers $a0–$a3 (though simpler compilers may choose to ignore this convention and pass all arguments via the stack). The remaining arguments are pushed on the stack.

2. Save the caller-saved registers. This includes registers $t0–t9$, if they contain live values at the call site.

3. Execute a jal instruction.

Within the called routine, the following steps are necessary:

1. Establish the stack frame by subtracting the frame size from the stack pointer.

2. Save the callee-saved registers in the frame. Register $fp$ is always saved. Register $ra$ needs to be saved if the routine itself makes calls. Any of the registers $s0–$s7 that are used by the callee need to be saved.

3. Establish the frame pointer by adding the stack frame size to the address in $sp$.

Finally, to return from a call, a function places the returned value into $v0$ and executes the following steps:

1. Restore any callee-saved registers that were saved upon entry.

2. Pop the stack frame by subtracting the frame size from $sp$.

3. Return by jumping to the address in register $ra$. 