

and is read-only. If this bit is 1, the transmitter is ready to accept a new character for output. If it is 0, the transmitter is still busy writing the previous character. Bit 1 is “interrupt enable” and is readable and writable. If this bit is set to 1, then the terminal requests an interrupt at hardware level 0 whenever the transmitter is ready for a new character and the ready bit becomes 1.

The final device register is the *Transmitter Data register* (at address `ffff000chex`). When a value is written into this location, its low-order 8 bits (i.e., an ASCII character as in Figure 2.21 in Chapter 2) are sent to the console. When the Transmitter Data register is written, the ready bit in the Transmitter Control register is reset to 0. This bit stays 0 until enough time has elapsed to transmit the character to the terminal; then the ready bit becomes 1 again. The Transmitter Data register should only be written when the ready bit of the Transmitter Control register is 1. If the transmitter is not ready, writes to the Transmitter Data register are ignored (the write appears to succeed but the character is not output).

Real computers require time to send characters to a console or terminal. These time lags are simulated by SPIM. For example, after the transmitter starts to write a character, the transmitter’s ready bit becomes 0 for a while. SPIM measures time in instructions executed, not in real clock time. This means that the transmitter does not become ready again until the processor executes a fixed number of instructions. If you stop the machine and look at the ready bit, it will not change. However, if you let the machine run, the bit eventually changes back to 1.

## **A.9** **SPIM**

SPIM is a software simulator that runs assembly language programs written for processors that implement the MIPS32 architecture, specifically Release 1 of this architecture with a fixed memory mapping, no caches, and only coprocessors 0 and 1.<sup>2</sup> SPIM’s name is just MIPS spelled backwards. SPIM can read and immediately execute assembly language files. SPIM is a self-contained system for running MIPS programs. It contains a debugger and provides a few operating system–like services. SPIM is much slower than a real computer (100 or more

---

2. Earlier versions of SPIM (before 7.0) implemented the MIPS-I architecture used in the original MIPS R2000 processors. This architecture is almost a proper subset of the MIPS32 architecture, with the difference being the manner in which exceptions are handled. MIPS32 also introduced approximately 60 new instructions, which are supported by SPIM. Programs that ran on the earlier versions of SPIM and did not use exceptions should run unmodified on newer versions of SPIM. Programs that used exceptions will require minor changes.

times). However, its low cost and wide availability cannot be matched by real hardware!

An obvious question is, Why use a simulator when most people have PCs that contain processors that run significantly faster than SPIM? One reason is that the processor in PCs are Intel 80x86s, whose architecture is far less regular and far more complex to understand and program than MIPS processors. The MIPS architecture may be the epitome of a simple, clean RISC machine.

In addition, simulators can provide a better environment for assembly programming than an actual machine because they can detect more errors and provide a better interface than an actual computer.

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

## Simulation of a Virtual Machine

The basic MIPS architecture is difficult to program directly because of 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 present an interface designed for compilers rather than assembly language programmers. A good part of the programming complexity results from delayed instructions. A *delayed branch* requires two cycles to execute (see Elaborations on pages 382 and 423 of Chapter 6). 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. It can also be a *nop* (no operation) that does nothing. Similarly, *delayed loads* require 2 cycles to bring a value from memory, so the instruction immediately following a load cannot use the value (see Section 6.2 of Chapter 6).

MIPS wisely chose to hide this complexity by having its assembler implement a **virtual machine**. This virtual computer appears to have nondelayed branches and loads and a richer instruction set than the actual hardware. The assembler *reorganizes* (rearranges) instructions to fill the delay slots. The virtual computer also provides *pseudoinstructions*, which appear as real instructions in assembly language programs. The hardware, however, knows nothing about pseudoinstructions, so the assembler translates them into equivalent sequences of actual machine instructions. For example, the MIPS hardware only provides instructions to branch when a register is equal to or not equal to 0. Other conditional branches, such as one that branches when one register is greater than another, are synthesized by comparing the two registers and branching when the result of the comparison is true (nonzero).



**virtual machine** A virtual computer that appears to have nondelayed branches and loads and a richer instruction set than the actual hardware.



By default, SPIM simulates the richer virtual machine, since this is the machine that most programmers will find useful. However, SPIM can also simulate the delayed branches and loads in the actual hardware. Below, we describe the virtual machine and only mention in passing features that do not belong to the actual hardware. In doing so, we follow the convention of MIPS assembly language programmers (and compilers), who routinely use the extended machine as if it was implemented in silicon.

## Getting Started with SPIM

The rest of this appendix introduces SPIM and the MIPS R2000 Assembly language. Many details should never concern you; however, the sheer volume of information can sometimes obscure the fact that SPIM is a simple, easy-to-use program. This section starts with a quick tutorial on using SPIM, which should enable you to load, debug, and run simple MIPS programs.

SPIM comes in different versions for different types of computer systems. The one constant is the simplest version, called `spim`, which is a command-line-driven program that runs in a console window. It operates like most programs of this type: you type a line of text, hit the return key, and `spim` executes your command. Despite its lack of a fancy interface, `spim` can do everything that its fancy cousins can do.

There are two fancy cousins to `spim`. The version that runs in the X-windows environment of a UNIX or Linux system is called `xspim`. `xspim` is an easier program to learn and use than `spim` because its commands are always visible on the screen and because it continually displays the machine's registers and memory. The other fancy version is called `pcspim` and runs on Microsoft Windows. The UNIX and Windows versions of [SPIM](#) are on the CD (click on Tutorials) . Tutorials on `xspim`, `pcspim`, `spim`, and [spim command-line options](#)  are on the CD (click on Software).

If you are going to run `spim` on a PC running Microsoft Windows, you should first look at the tutorial on [PCSpim](#)  on this CD. If you are going to run `spim` on a computer running UNIX or Linux, you should read the tutorial on [xspim](#)  (click on Tutorials).

## Surprising Features

Although SPIM faithfully simulates the MIPS computer, SPIM is a simulator and certain things are not identical to an actual computer. The most obvious differences are that instruction timing and the memory system are not identical. SPIM does not simulate caches or memory latency, nor does it accurately reflect floating-point operation or multiply and divide instruction delays. In addition, the floating-point instructions do not detect many error conditions, which would cause exceptions on a real machine.

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

## Byte Order

Processors can number bytes within a word so the byte with the lowest number is either the leftmost or rightmost one. The convention used by a machine is called its *byte order*. MIPS processors can operate with either *big-endian* or *little-endian* byte order. For example, in a big-endian machine, the directive `.byte 0, 1, 2, 3` would result in a memory word containing

Byte #			
0	1	2	3

while in a little-endian machine, the word would contain

Byte #			
3	2	1	0

SPIM operates with both byte orders. SPIM's byte order is the same as the byte order of the underlying machine that runs the simulator. For example, on a Intel 80x86, SPIM is little-endian, while on a Macintosh or Sun SPARC, SPIM is big-endian.

## 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 Figure A.9.1) into register `$v0` and arguments into registers `$a0–$a3` (or `$f12` for floating-point values). System calls that return values put their results in register `$v0` (or `$f0` for floating-point results). For example, the following code prints “the answer = 5”:

```
.data
str:
.asciiz "the answer = "
.text
```

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_char	11	\$a0 = char	
read_char	12		char (in \$a0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

FIGURE A.9.1 System services.

```

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

```

The `print_int` system call 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; and `print_string` is passed a pointer to a null-terminated string, which it writes to the console.

The system calls `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 fewer than  $n - 1$  characters are on the current line, `read_string` reads up to and including the newline and again null-terminates the string.