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 the 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 systems 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 pseudo-instruction 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 an 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 B.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 $v0)
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 B.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 to 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.

Warning: Programs that use these syscalls to read from the terminal should not use memory-mapped I/O (see Section B.8).

`sbrk` returns a pointer to a block of memory containing *n* additional bytes. `exit` stops the program SPIM is running. `exit2` terminates the SPIM program, and the argument to `exit2` becomes the value returned when the SPIM simulator itself terminates.

`print_char` and `read_char` write and read a single character. `open`, `read`, `write`, and `close` are the standard UNIX library calls.

B.10 MIPS R2000 Assembly Language

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 B.10.1). SPIM simulates two coprocessors. Coprocessor 0 handles exceptions and interrupts. Coprocessor 1 is the floating-point unit. SPIM simulates most aspects of this unit.

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 `c` and 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
label	address of label
label ± imm	address of label + or – immediate
label ± imm (register)	address of label + 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

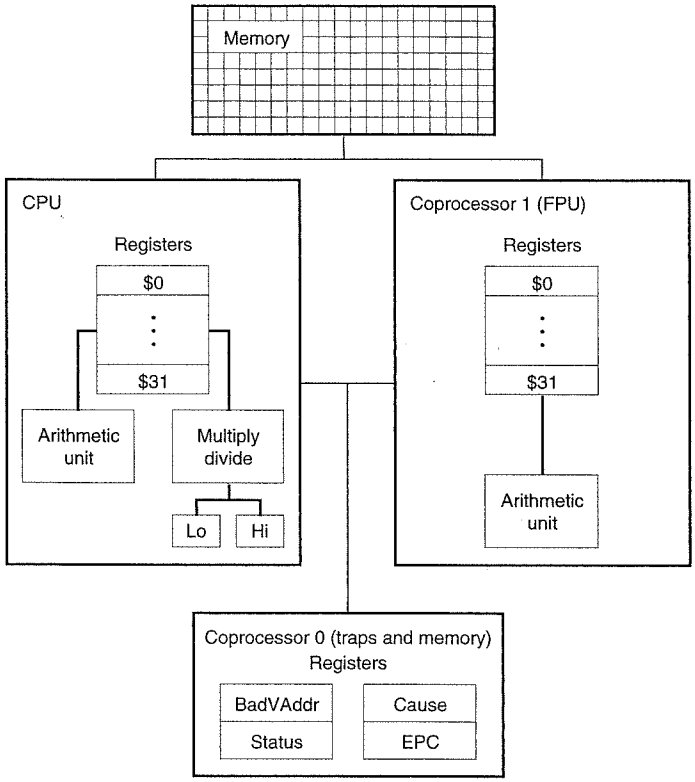


FIGURE B.10.1 MIPS R2000 CPU and FPU.

object must be stored at even addresses, and a full word object must be stored at addresses that are a multiple of four. However, MIPS provides some instructions to manipulate unaligned data (`lwl`, `lwr`, `swl`, and `swr`).

Elaboration: The MIPS assembler (and SPIM) synthesizes the more complex addressing modes by producing one or more instructions before the load or store to compute a complex address. For example, suppose that the label `table` referred to memory location 0×10000004 and a program contained the instruction

```
ld $a0, table + 4($a1)
```

The assembler would translate this instruction into the instructions

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

The first instruction loads the upper bits of the label's address into register `$at`, which is the register that the assembler reserves for its own use. The second instruction adds the contents of register `$a1` to the label's partial address. Finally, the load instruction uses the hardware address mode to add the sum of the lower bits of the label's address and the offset from the original instruction to the value in register `$at`.

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. Instruction opcodes are reserved words that *cannot* be used as 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
```

Numbers are base 10 by default. If they are preceded by *0x*, they are interpreted as hexadecimal. Hence, 256 and 0x100 denote the same value.

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 MIPS assembler directives:

`.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 *str* in memory, but do not null-terminate it.

`.asciiz str`	Store the string *str* in memory and null-terminate it.
`.byte b1,..., bn`	Store the *n* values in successive bytes of memory.
`.data <addr>`	Subsequent items are stored in the data segment. If the optional argument *addr* is present, subsequent items are stored starting 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 label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.
`.float f1,..., fn`	Store the *n* floating-point single precision numbers in successive memory locations.
`.globl sym`	Declare that label *sym* is global and can be referenced from other files.
`.half h1,..., hn`	Store the *n* 16-bit quantities in successive memory halfwords.
`.kdata <addr>`	Subsequent data items are stored in the kernel data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.ktext <addr>`	Subsequent items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.set noat` and `.set at`	The first directive prevents SPIM from complaining about subsequent instructions that use register `$at`. The second directive re-enables the warning. Since pseudoinstructions expand into code that uses register `$at`, programmers must be very careful about leaving values in this register.
`.space n`	Allocates *n* bytes of space in the current segment (which must be the data segment in SPIM).

`.text <addr>` Subsequent 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, subsequent items are stored starting 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`).

Encoding MIPS Instructions

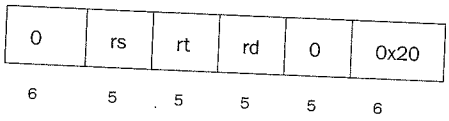
Figure B.10.2 explains how a MIPS instruction is encoded in a binary number. Each column contains instruction encodings for a field (a contiguous group of bits) from an instruction. The numbers at the left margin are values for a field. For example, the `j` opcode has a value of 2 in the opcode field. The text at the top of a column names a field and specifies which bits it occupies in an instruction. For example, the `op` field is contained in bits 26–31 of an instruction. This field encodes most instructions. However, some groups of instructions use additional fields to distinguish related instructions. For example, the different floating-point instructions are specified by bits 0–5. The arrows from the first column show which opcodes use these additional fields.

Instruction Format

The rest of this appendix describes both the instructions implemented by actual MIPS hardware and the pseudoinstructions provided by the MIPS assembler. The two types of instructions are easily distinguished. Actual instructions depict the fields in their binary representation. For example, in

Addition (with overflow)

`add rd, rs, rt`



the `add` instruction consists of six fields. Each field's size in bits is the small number below the field. This instruction begins with six bits of 0s. Register specifiers begin with an *r*, so the next field is a 5-bit register specifier called `rs`. This is the same register that is the second argument in the symbolic assembly at the left of this line. Another common field is $\texttt{imm}_{16}$, which is a 16-bit immediate number.

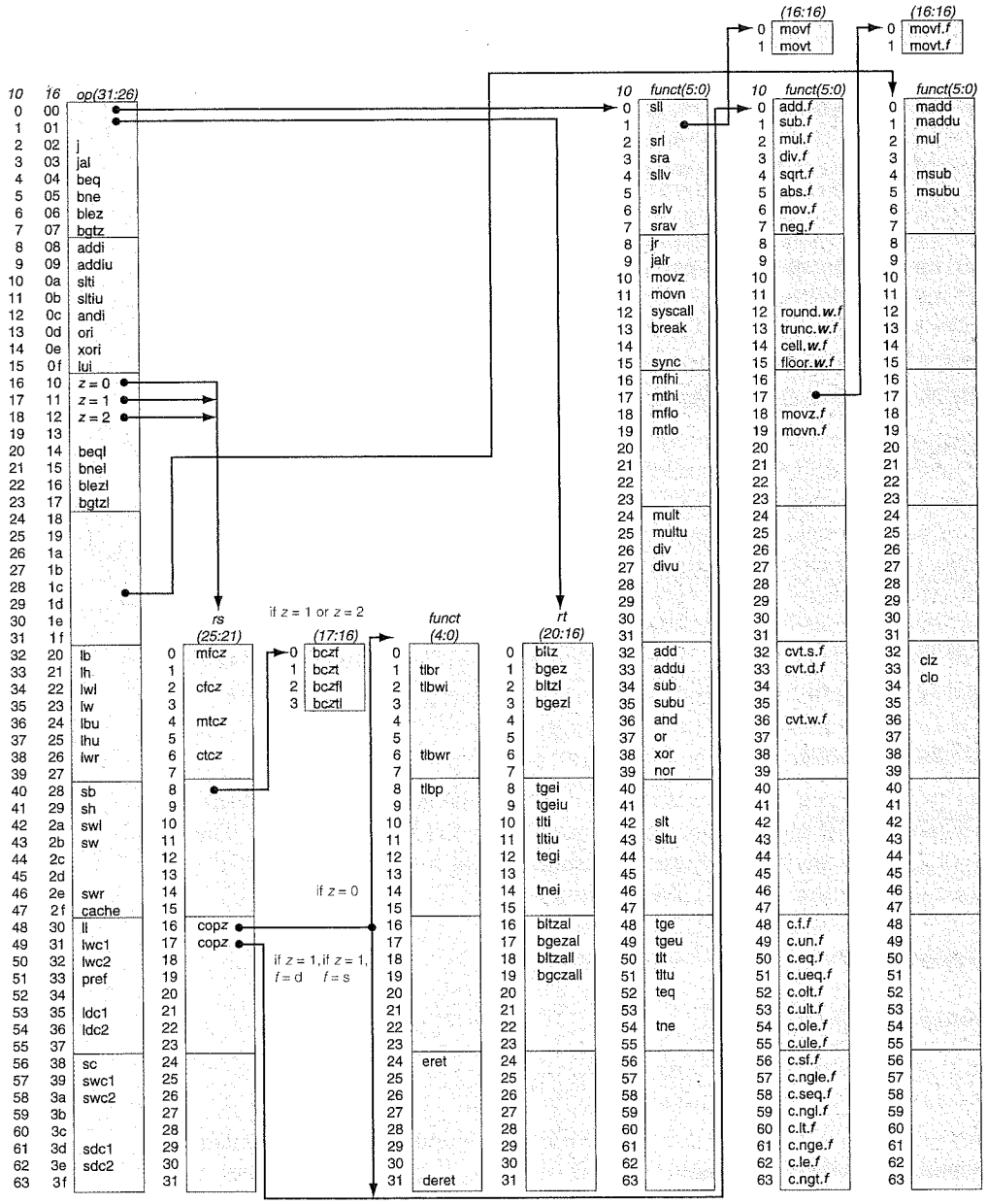


FIGURE B.10.2 MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10, and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for six op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses "*f*" to mean "s" if rs = 16 and op = 17 or "d" if rs = 17 and op = 17. The second field (rs) uses "*z*" to mean "0", "1", "2", or "3" if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if *z* = 0, the operations are specified in the fourth field (bits 4 to 0); if *z* = 1, then the operations are in the last field with *f* = s. If rs = 17 and *z* = 1, then the operations are in the last field with *f* = d.

Pseudoinstructions follow roughly the same conventions, but omit instruction encoding information. For example:

Multiply (without overflow)

```
mul rdest, rsrc1, src2     pseudoinstruction
```

In pseudoinstructions, `rdest` and `rsrc1` are registers and `src2` is either a register or an immediate value. In general, the assembler and SPIM translate a more general form of an instruction (e.g., `add $v1, $a0, 0x55`) to a specialized form (e.g., `addi $v1, $a0, 0x55`).

Arithmetic and Logical Instructions

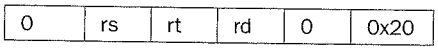
Absolute value

```
abs rdest, rsrc     pseudoinstruction
```

Put the absolute value of register `rsrc` in register `rdest`.

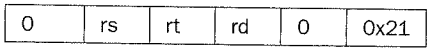
Addition (with overflow)

```
add rd, rs, rt
```



Addition (without overflow)

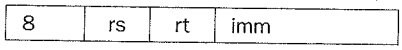
```
addu rd, rs, rt
```



Put the sum of registers `rs` and `rt` into register `rd`.

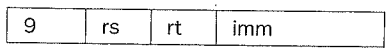
Addition immediate (with overflow)

```
addi rt, rs, imm
```



Addition immediate (without overflow)

```
addiu rt, rs, imm
```



Put the sum of register `rs` and the sign-extended immediate into register `rt`.