6. Jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's `main` routine. If the `main` routine returns, the start-up routine terminates the program with the exit system call.

A.5 Memory Usage

The next few sections elaborate the description of the MIPS architecture presented earlier in the book. Earlier chapters focused primarily on hardware and its relationship with low-level software. These sections focus primarily on how assembly language programmers use MIPS hardware. These sections describe a set of conventions followed on many MIPS systems. For the most part, the hardware does not impose these conventions. Instead, they represent an agreement among programmers to follow the same set of rules so that software written by different people can work together and make effective use of MIPS hardware.

Systems based on MIPS processors typically divide memory into three parts (see Figure A.9). The first part, near the bottom of the address space (starting at address 400000_{hex}), is the *text segment*, which holds the program's instructions.

The second part, above the text segment, is the *data segment*, which is further divided into two parts. *Static data* (starting at address 10000000_{hex}) contains objects whose size is known to the compiler and whose lifetime—the interval during which a program can access them—is the program's entire execution. For example, in C, global variables are statically allocated since they can be referenced anytime during a program's execution. The linker both assigns static objects to locations in the data segment and resolves references to these objects.

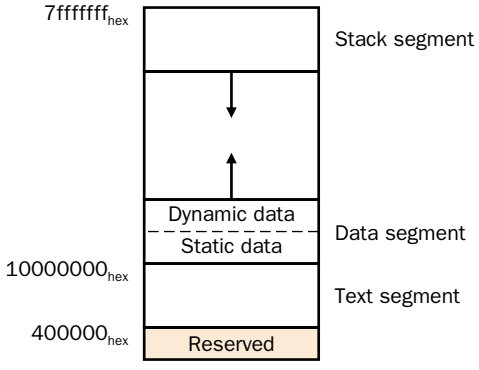

FIGURE A.9 Layout of memory.

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and 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)

FIGURE A.10 MIPS registers and usage convention.

Procedure Calls

This section describes the steps that occur when one procedure (the *caller*) invokes another procedure (the *callee*). Programmers who write in a high-level language (like C or Pascal) never see the details of how one procedure calls another because the compiler takes care of this low-level bookkeeping. However, assembly language programmers must explicitly implement every procedure call and return.

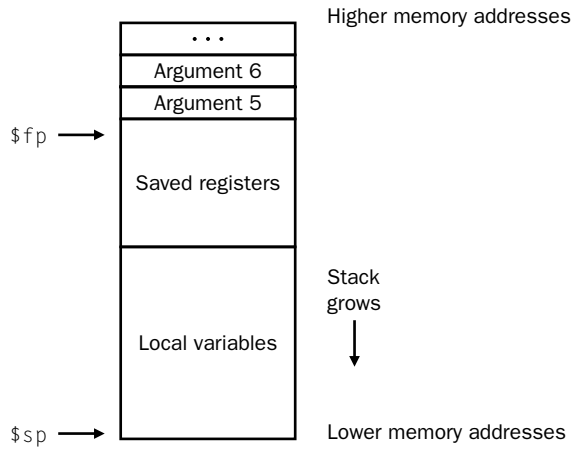

FIGURE A.11 Layout of a stack frame. The frame pointer (`$fp`) points to the first word in the currently executing procedure's stack frame. The stack pointer (`$sp`) points to the last word of frame. The first four arguments are passed in registers, so the fifth argument is the first one stored on the stack.

2. Save callee-saved registers in the frame. A callee must save the values in these registers (`$s0`–`$s7`, `$fp`, and `$ra`) before altering them since the caller expects to find these registers unchanged after the call. Register `$fp` is saved by every procedure that allocates a new stack frame. However, register `$ra` only needs to be saved if the callee itself makes a call. The other callee-saved registers that are used also must be saved.
3. Establish the frame pointer by adding the stack frame's size minus four to `$sp` and storing the sum in register `$fp`.

Hardware Software Interface

The MIPS register use convention provides callee- and caller-saved registers because both types of registers are advantageous in different circumstances. Callee-saved registers are better used to hold long-lived values, such as variables from a user's program. These registers are only saved during a procedure call if the callee expects to use the register. On the other hand, caller-saved registers are better used to hold short-lived quantities that do not persist across a call, such as immediate values in an address calculation. During a call, the callee can also use these registers for short-lived temporaries.

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		

FIGURE A.17 System services.

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

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

A.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 A.18). SPIM simulates two coprocessors. Coprocessor 0 handles exceptions, interrupts, 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.

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: