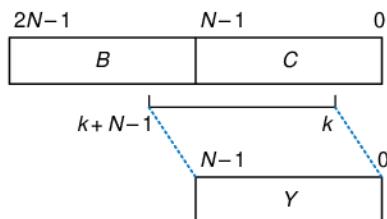


called  $B$  and the least significant  $N$  bits are called  $C$ . By choosing appropriate values of  $B$ ,  $C$ , and  $k$ , the funnel shifter can perform any type of shift or rotate. Explain what these values should be in terms of  $A$ ,  $shamt$ , and  $N$  for

- (a) logical right shift of  $A$  by  $shamt$ .
- (b) arithmetic right shift of  $A$  by  $shamt$ .
- (c) left shift of  $A$  by  $shamt$ .
- (d) right rotate of  $A$  by  $shamt$ .
- (e) left rotate of  $A$  by  $shamt$ .



**Figure 5.64** Funnel shifter

**Exercise 5.18** Find the critical path for the  $4 \times 4$  multiplier from Figure 5.18 in terms of an AND gate delay ( $t_{AND}$ ) and a full adder delay ( $t_{FA}$ ). What is the delay of an  $N \times N$  multiplier built in the same way?

**Exercise 5.19** Design a multiplier that handles two's complement numbers.

**Exercise 5.20** A *sign extension unit* extends a two's complement number from  $M$  to  $N$  ( $N > M$ ) bits by copying the most significant bit of the input into the upper bits of the output (see Section 1.4.6). It receives an  $M$ -bit input,  $A$ , and produces an  $N$ -bit output,  $Y$ . Sketch a circuit for a sign extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

**Exercise 5.21** A *zero extension unit* extends an unsigned number from  $M$  to  $N$  bits ( $N > M$ ) by putting zeros in the upper bits of the output. Sketch a circuit for a zero extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

**Exercise 5.22** Compute  $111001.000_2 / 001100.000_2$  in binary using the standard division algorithm from elementary school. Show your work.

**Exercise 5.23** What is the range of numbers that can be represented by the following number systems?

- (a) 24-bit unsigned fixed-point numbers with 12 integer bits and 12 fraction bits
- (b) 24-bit sign and magnitude fixed-point numbers with 12 integer bits and 12 fraction bits
- (c) 24-bit two's complement fixed-point numbers with 12 integer bits and 12 fraction bits

**Exercise 5.24** Express the following base 10 numbers in 16-bit fixed-point sign/magnitude format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

- (a) -13.5625
- (b) 42.3125
- (c) -17.15625

**Exercise 5.25** Express the base 10 numbers in Exercise 5.24 in 16-bit fixed-point two's complement format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

**Exercise 5.26** Express the base 10 numbers in Exercise 5.24 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

**Exercise 5.27** Convert the following two's complement binary fixed-point numbers to base 10.

- (a) 0101.1000
- (b) 1111.1111
- (c) 1000.0000

**Exercise 5.28** When adding two floating-point numbers, the number with the smaller exponent is shifted. Why is this? Explain in words and give an example to justify your explanation.

**Exercise 5.29** Add the following IEEE 754 single-precision floating-point numbers.

- (a) C0D20004 + 72407020
- (b) C0D20004 + 40DC0004
- (c) (5FBE4000 + 3FF80000) + DFDE4000  
(Why is the result counterintuitive? Explain.)

**Exercise 5.30** Expand the steps in section 5.3.2 for performing floating-point addition to work for negative as well as positive floating-point numbers.

**Exercise 5.31** Consider IEEE 754 single-precision floating-point numbers.

- (a) How many numbers can be represented by IEEE 754 single-precision floating-point format? You need not count  $\pm\infty$  or NaN.
- (b) How many additional numbers could be represented if  $\pm\infty$  and NaN were not represented?
- (c) Explain why  $\pm\infty$  and NaN are given special representations.

**Exercise 5.32** Consider the following decimal numbers: 245 and 0.0625.

- (a) Write the two numbers using single-precision floating-point notation. Give your answers in hexadecimal.
- (b) Perform a magnitude comparison of the two 32-bit numbers from part (a). In other words, interpret the two 32-bit numbers as two's complement numbers and compare them. Does the integer comparison give the correct result?
- (c) You decide to come up with a new single-precision floating-point notation. Everything is the same as the IEEE 754 single-precision floating-point standard, except that you represent the exponent using two's complement instead of a bias. Write the two numbers using your new standard. Give your answers in hexadecimal.
- (d) Does integer comparison work with your new floating-point notation from part (d)?
- (e) Why is it convenient for integer comparison to work with floating-point numbers?

**Exercise 5.33** Design a single-precision floating-point adder using your favorite HDL. Before coding the design in an HDL, sketch a schematic of your design. Simulate and test your adder to prove to a skeptic that it functions correctly. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

**Exercise 5.34** In this problem, you will explore the design of a 32-bit floating-point multiplier. The multiplier has two 32-bit floating-point inputs and produces a 32-bit floating-point output. You may consider positive numbers only

and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

- (a) Write the steps necessary to perform 32-bit floating-point multiplication.
- (b) Sketch the schematic of a 32-bit floating-point multiplier.
- (c) Design a 32-bit floating-point multiplier in an HDL. Simulate and test your multiplier to prove to a skeptic that it functions correctly.

**Exercise 5.35** In this problem, you will explore the design of a 32-bit prefix adder.

- (a) Sketch a schematic of your design.
- (b) Design the 32-bit prefix adder in an HDL. Simulate and test your adder to prove that it functions correctly.
- (c) What is the delay of your 32-bit prefix adder from part (a)? Assume that each two-input gate delay is 100 ps.
- (d) Design a pipelined version of the 32-bit prefix adder. Sketch the schematic of your design. How fast can your pipelined prefix adder run? Make the design run as fast as possible.
- (e) Design the pipelined 32-bit prefix adder in an HDL.

**Exercise 5.36** An incrementer adds 1 to an  $N$ -bit number. Build an 8-bit incrementer using half adders.

**Exercise 5.37** Build a 32-bit synchronous *Up/Down counter*. The inputs are *Reset* and *Up*. When *Reset* is 1, the outputs are all 0. Otherwise, when *Up* = 1, the circuit counts up, and when *Up* = 0, the circuit counts down.

**Exercise 5.38** Design a 32-bit counter that adds 4 at each clock edge. The counter has reset and clock inputs. Upon reset, the counter output is all 0.

**Exercise 5.39** Modify the counter from Exercise 5.38 such that the counter will either increment by 4 or load a new 32-bit value,  $D$ , on each clock edge, depending on a control signal, *PCSrc*. When *PCSrc* = 1, the counter loads the new value  $D$ .

**Exercise 5.40** An  $N$ -bit *Johnson counter* consists of an  $N$ -bit shift register with a reset signal. The output of the shift register ( $S_{out}$ ) is inverted and fed back to the input ( $S_{in}$ ). When the counter is reset, all of the bits are cleared to 0.

- (a) Show the sequence of outputs,  $Q_{3:0}$ , produced by a 4-bit Johnson counter starting immediately after the counter is reset.

- (b) How many cycles elapse until an  $N$ -bit Johnson counter repeats its sequence? Explain.
- (c) Design a decimal counter using a 5-bit Johnson counter, ten AND gates, and inverters. The decimal counter has a clock, a reset, and ten one-hot outputs,  $Y_{9:0}$ . When the counter is reset,  $Y_0$  is asserted. On each subsequent cycle, the next output should be asserted. After ten cycles, the counter should repeat. Sketch a schematic of the decimal counter.
- (d) What advantages might a Johnson counter have over a conventional counter?

**Exercise 5.41** Write the HDL for a 4-bit scannable flip-flop like the one shown in Figure 5.37. Simulate and test your HDL module to prove that it functions correctly.

**Exercise 5.42** The English language has a good deal of redundancy that allows us to reconstruct garbled transmissions. Binary data can also be transmitted in redundant form to allow error correction. For example, the number 0 could be coded as 00000 and the number 1 could be coded as 11111. The value could then be sent over a noisy channel that might flip up to two of the bits. The receiver could reconstruct the original data because a 0 will have at least three of the five received bits as 0's; similarly a 1 will have at least three 1's.

- (a) Propose an encoding to send 00, 01, 10, or 11 encoded using five bits of information such that all errors that corrupt one bit of the encoded data can be corrected. Hint: the encodings 00000 and 11111 for 00 and 11, respectively, will not work.
- (b) Design a circuit that receives your five-bit encoded data and decodes it to 00, 01, 10, or 11, even if one bit of the transmitted data has been changed.
- (c) Suppose you wanted to change to an alternative 5-bit encoding. How might you implement your design to make it easy to change the encoding without having to use different hardware?

**Exercise 5.43** Flash EEPROM, simply called Flash memory, is a fairly recent invention that has revolutionized consumer electronics. Research and explain how Flash memory works. Use a diagram illustrating the floating gate. Describe how a bit in the memory is programmed. Properly cite your sources.

**Exercise 5.44** The extraterrestrial life project team has just discovered aliens living on the bottom of Mono Lake. They need to construct a circuit to classify the aliens by potential planet of origin based on measured features

available from the NASA probe: greenness, brownness, sliminess, and ugliness. Careful consultation with xenobiologists leads to the following conclusions:

- ▶ If the alien is green and slimy or ugly, brown, and slimy, it might be from Mars.
- ▶ If the critter is ugly, brown, and slimy, or green and neither ugly nor slimy, it might be from Venus.
- ▶ If the beastie is brown and neither ugly nor slimy or is green and slimy, it might be from Jupiter.

Note that this is an inexact science; for example, a life form which is mottled green and brown and is slimy but not ugly might be from either Mars or Jupiter.

- (a) Program a  $4 \times 4 \times 3$  PLA to identify the alien. You may use dot notation.
- (b) Program a  $16 \times 3$  ROM to identify the alien. You may use dot notation.
- (c) Implement your design in an HDL.

**Exercise 5.45** Implement the following functions using a single  $16 \times 3$  ROM. Use dot notation to indicate the ROM contents.

- (a)  $X = AB + B\bar{C}D + \bar{A}\bar{B}$
- (b)  $Y = AB + BD$
- (c)  $Z = A + B + C + D$

**Exercise 5.46** Implement the functions from Exercise 5.45 using an  $4 \times 8 \times 3$  PLA. You may use dot notation.

**Exercise 5.47** Specify the size of a ROM that you could use to program each of the following combinational circuits. Is using a ROM to implement these functions a good design choice? Explain why or why not.

- (a) a 16-bit adder/subtractor with  $C_{in}$  and  $C_{out}$
- (b) an  $8 \times 8$  multiplier
- (c) a 16-bit priority encoder (see Exercise 2.25)

**Exercise 5.48** Consider the ROM circuits in Figure 5.65. For each row, can the circuit in column I be replaced by an equivalent circuit in column II by proper programming of the latter's ROM?

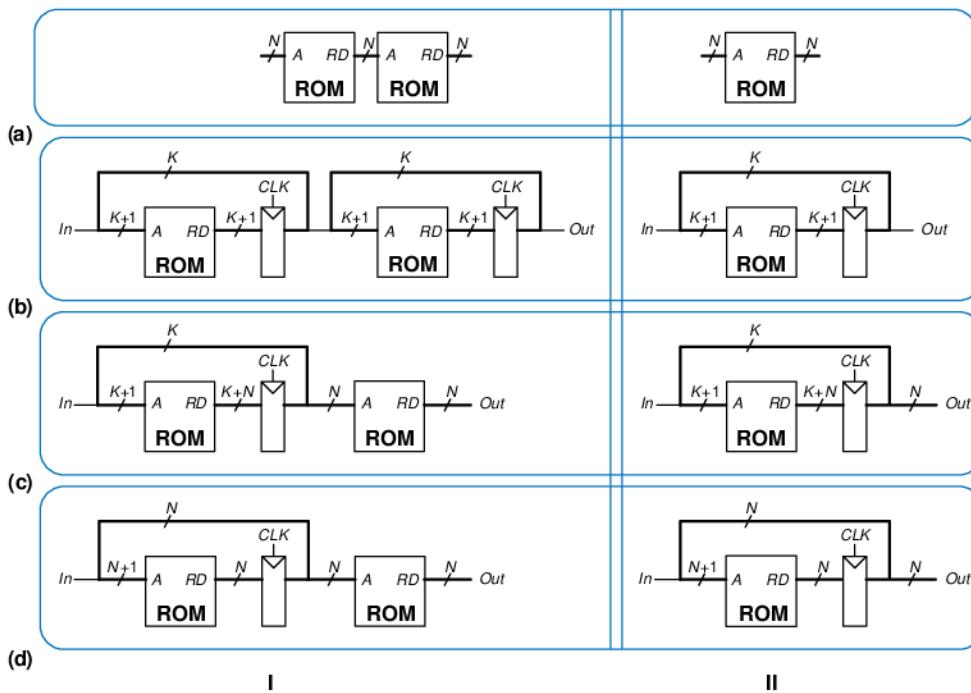


Figure 5.65 ROM circuits

**Exercise 5.49** Give an example of a nine-input function that can be performed using only one Spartan FPGA CLB. Give an example of an eight-input function that cannot be performed using only one CLB.

**Exercise 5.50** How many Spartan FPGA CLBs are required to perform each of the following functions? Show how to configure one or more CLBs to perform the function. You should be able to do this by inspection, without performing logic synthesis.

- The combinational function from Exercise 2.7(c).
- The combinational function from Exercise 2.9(c).
- The two-output function from Exercise 2.15.
- The function from Exercise 2.24.
- A four-input priority encoder (see Exercise 2.25).
- An eight-input priority encoder (see Exercise 2.25).
- A 3:8 decoder.

- (h) A 4-bit carry propagate adder (with no carry in or out).
- (i) The FSM from Exercise 3.19.
- (j) The Gray code counter from Exercise 3.24.

**Exercise 5.51** Consider the Spartan CLB shown in Figure 5.58. It has the following specifications:  $t_{pd} = t_{cd} = 2.7$  ns per CLB;  $t_{\text{setup}} = 3.9$  ns,  $t_{\text{hold}} = 0$  ns, and  $t_{pcq} = 2.8$  ns for all flip-flops.

- (a) What is the minimum number of Spartan CLBs required to implement the FSM of Figure 3.26?
- (b) Without clock skew, what is the fastest clock frequency at which this FSM will run reliably?
- (c) With 5 ns of clock skew, what is the fastest frequency at which the FSM will run reliably?

**Exercise 5.52** You would like to use an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. The design is to be implemented as an FSM using a Spartan XC3S200 FPGA, a chip from the Spartan 3 series family. It is considerably faster than the original Spartan FPGA. According to the data sheet, the FPGA has timing characteristics shown in Table 5.5. Assume that the design is small enough that wire delay is negligible.

**Table 5.5 Spartan 3 XC3S200 timing**

Name	Value (ns)
$t_{pcq}$	0.72
$t_{\text{setup}}$	0.53
$t_{\text{hold}}$	0
$t_{pd}$ (per CLB)	0.61
$t_{\text{skew}}$	0

You would like your FSM to run at 100 MHz. What is the maximum number of CLBs on the critical path? What is the fastest speed at which the FSM will run?

## Interview Questions

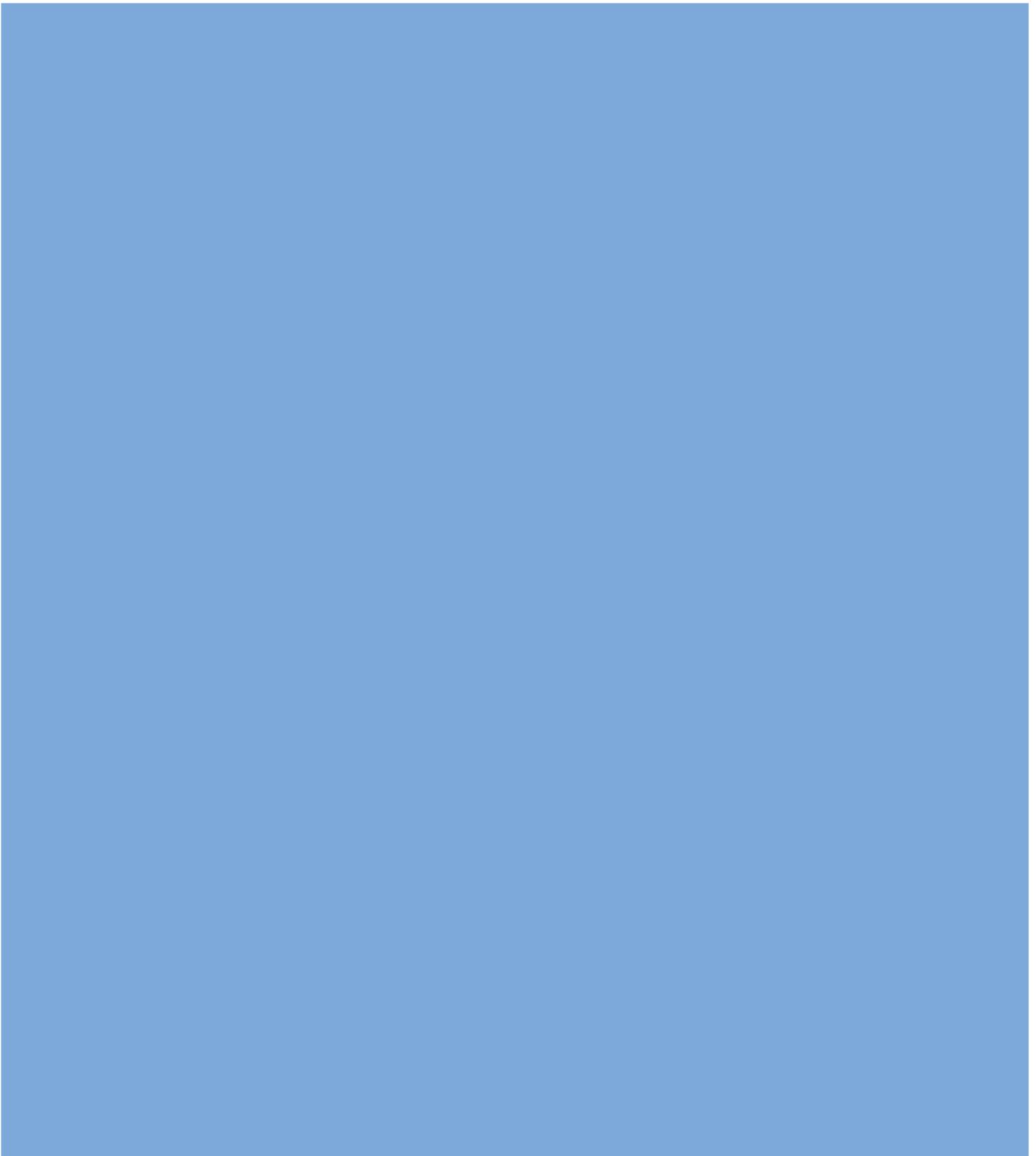
---

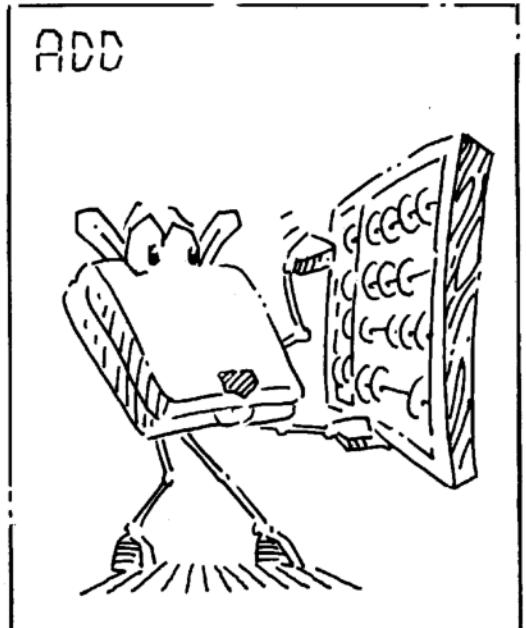
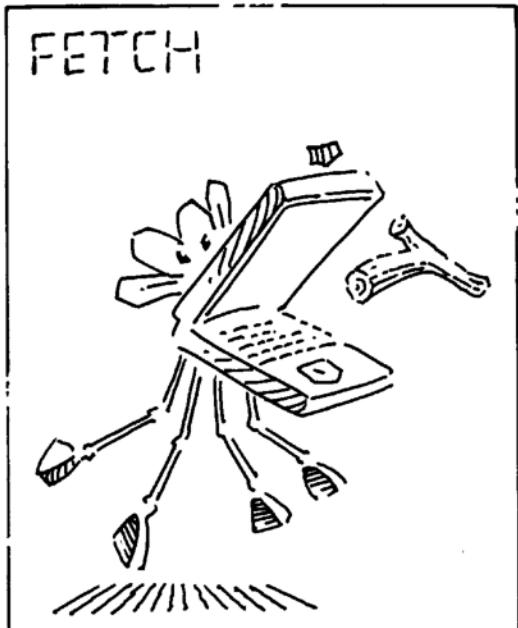
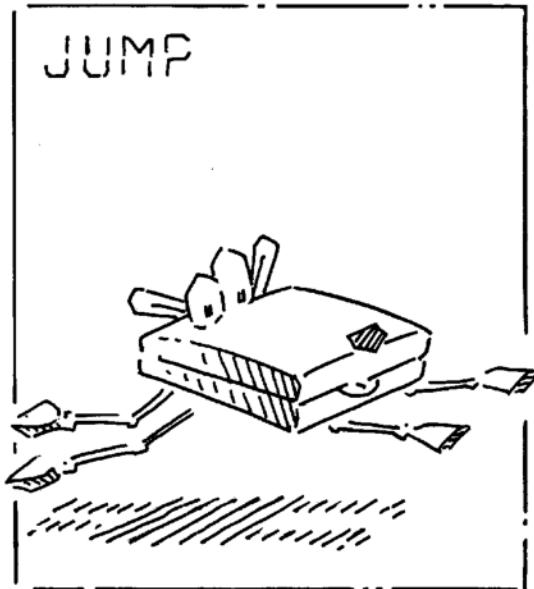
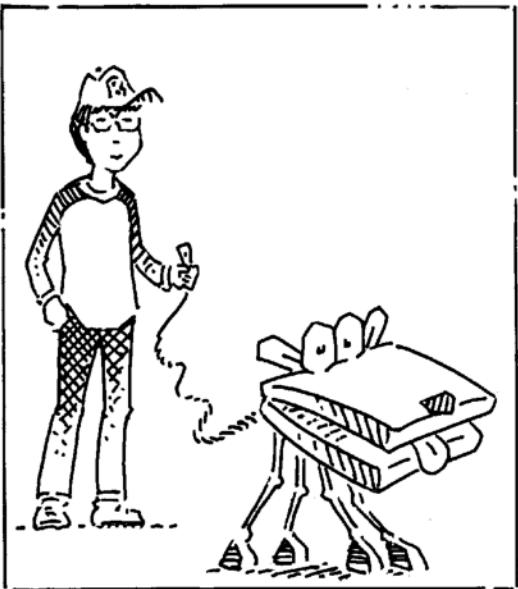
The following exercises present questions that have been asked at interviews for digital design jobs.

**Question 5.1** What is the largest possible result of multiplying two unsigned  $N$ -bit numbers?

**Question 5.2** *Binary coded decimal (BCD)* representation uses four bits to encode each decimal digit. For example  $42_{10}$  is represented as  $01000010_{BCD}$ . Explain in words why processors might use BCD representation.

**Question 5.3** Design hardware to add two 8-bit unsigned BCD numbers (see Question 5.2). Sketch a schematic for your design, and write an HDL module for the BCD adder. The inputs are  $A$ ,  $B$ , and  $C_{in}$ , and the outputs are  $S$  and  $C_{out}$ .  $C_{in}$  and  $C_{out}$  are 1-bit carries, and  $A$ ,  $B$ , and  $S$  are 8-bit BCD numbers.





# 6

## Architecture

### 6.1 INTRODUCTION

The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the *architecture* of a computer (see Figure 1.1). The architecture is the programmer’s view of a computer. It is defined by the instruction set (language), and operand locations (registers and memory). Many different architectures exist, such as IA-32, MIPS, SPARC, and PowerPC.

The first step in understanding any computer architecture is to learn its language. The words in a computer’s language are called *instructions*. The computer’s vocabulary is called the instruction set. All programs running on a computer use the same *instruction set*. Even complex software applications, such as word processing and spreadsheet applications, are eventually compiled into a series of simple instructions such as add, subtract, and jump. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, from registers, or from the instruction itself.

Computer hardware understands only 1’s and 0’s, so instructions are encoded as binary numbers in a format called *machine language*. Just as we use letters to encode human language, computers use binary numbers to encode machine language. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be tedious, so we prefer to represent the instructions in a symbolic format, called *assembly language*.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions, such as add, subtract, and jump, that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.

- 6.1 [Introduction](#)
- 6.2 [Assembly Language](#)
- 6.3 [Machine Language](#)
- 6.4 [Programming](#)
- 6.5 [Addressing Modes](#)
- 6.6 [Lights, Camera, Action: Compiling, Assembling, and Loading](#)
- 6.7 [Odds and Ends\\*](#)
- 6.8 [Real World Perspective: IA-32 Architecture\\*](#)
- 6.9 [Summary](#)  
[Exercises](#)  
[Interview Questions](#)

What is the best architecture to study when first learning the subject?

Commercially successful architectures such as IA-32 are satisfying to study because you can use them to write programs on real computers. Unfortunately, many of these architectures are full of warts and idiosyncrasies accumulated over years of haphazard development by different engineering teams, making the architectures difficult to understand and implement.

Many textbooks teach imaginary architectures that are simplified to illustrate the key concepts.

We follow the lead of David Patterson and John Hennessy in their text, *Computer Organization and Design*, by focusing on the MIPS architecture. Hundreds of millions of MIPS microprocessors have shipped, so the architecture is commercially very important. Yet it is a clean architecture with little odd behavior. At the end of this chapter, we briefly visit the IA-32 architecture to compare and contrast it with MIPS.

A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) sell various microprocessors belonging to the same IA-32 architecture. They all can run the same programs, but they use different underlying hardware and therefore offer trade-offs in performance, price, and power. Some microprocessors are optimized for high-performance servers, whereas others are optimized for long battery life in laptop computers. The specific arrangement of registers, memories, ALUs, and other building blocks to form a microprocessor is called the *microarchitecture* and will be the subject of Chapter 7. Often, many different microarchitectures exist for a single architecture.

In this text, we introduce the MIPS architecture that was first developed by John Hennessy and his colleagues at Stanford in the 1980s. MIPS processors are used by, among others, Silicon Graphics, Nintendo, and Cisco. We start by introducing the basic instructions, operand locations, and machine language formats. We then introduce more instructions used in common programming constructs, such as branches, loops, array manipulations, and procedure calls.

Throughout the chapter, we motivate the design of the MIPS architecture using four principles articulated by Patterson and Hennessy: (1) simplicity favors regularity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises.

## 6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations are written in assembly language. We then define the MIPS instruction operands: registers, memory, and constants.

We assume that you already have some familiarity with a *high-level programming language* such as C, C++, or Java. (These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.)

### 6.2.1 Instructions

The most common operation computers perform is addition. Code Example 6.1 shows code for adding variables b and c and writing the result to a. The program is shown on the left in a high-level language (using the syntax of C, C++, and Java), and then rewritten on the right in MIPS assembly language. Note that statements in a C program end with a semicolon.

**Code Example 6.1 ADDITION**

High-Level Code	MIPS Assembly Code
a = b + c;	add a, b, c

**Code Example 6.2 SUBTRACTION**

High-Level Code	MIPS Assembly Code
a = b - c;	sub a, b, c

The first part of the assembly instruction, add, is called the *mnemonic* and indicates what operation to perform. The operation is performed on b and c, the *source operands*, and the result is written to a, the *destination operand*.

Code Example 6.2 shows that subtraction is similar to addition. The instruction format is the same as the add instruction except for the operation specification, sub. This consistent instruction format is an example of the first design principle:

**Design Principle 1:** Simplicity favors regularity.

Instructions with a consistent number of operands—in this case, two sources and one destination—are easier to encode and handle in hardware. More complex high-level code translates into multiple MIPS instructions, as shown in Code Example 6.3.

In the high-level language examples, single-line comments begin with // and continue until the end of the line. Multiline comments begin with /\* and end with \*/. In assembly language, only single-line comments are used. They begin with # and continue until the end of the line. The assembly language program in Code Example 6.3 requires a temporary variable, t, to store the intermediate result. Using multiple assembly

*mnemonic* (pronounced ni-mon-ik) comes from the Greek word μνημεστηθαι, to remember. The assembly language mnemonic is easier to remember than a machine language pattern of 0's and 1's representing the same operation.

**Code Example 6.3 MORE COMPLEX CODE**

High-Level Code	MIPS Assembly Code
a = b + c - d; // single-line comment /* multiple-line comment */	sub t, c, d # t = c - d add a, b, t # a = b + t

language instructions to perform more complex operations is an example of the second design principle of computer architecture:

**Design Principle 2:** Make the common case fast.

The MIPS instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, MIPS is a *reduced instruction set computer* (*RISC*) architecture. Architectures with many complex instructions, such as Intel's IA-32 architecture, are *complex instruction set computers* (*CISC*). For example, IA-32 defines a “string move” instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small. For example, an instruction set with 64 simple instructions would need  $\log_2 64 = 6$  bits to encode the operation. An instruction set with 256 complex instructions would need  $\log_2 256 = 8$  bits of encoding per instruction. In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

### 6.2.2 Operands: Registers, Memory, and Constants

An instruction operates on *operands*. In Code Example 6.1 the variables *a*, *b*, and *c* are all operands. But computers operate on 1’s and 0’s, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be *constants* stored in the instruction itself. Computers use various locations to hold operands, to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. MIPS is called a 32-bit architecture because it operates on 32-bit data. (The MIPS architecture has been extended to 64 bits in commercial products, but we will consider only the 32-bit form in this book.)

#### Registers

Instructions need to access operands quickly so that they can run fast. But operands stored in memory take a long time to retrieve. Therefore,

most architectures specify a small number of registers that hold commonly used operands. The MIPS architecture uses 32 registers, called the *register set* or *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

**Design Principle 3:** Smaller is faster.

Looking up information from a small number of relevant books on your desk is a lot faster than searching for the information in the stacks at a library. Likewise, reading data from a small set of registers (for example, 32) is faster than reading it from 1000 registers or a large memory. A small register file is typically built from a small SRAM array (see Section 5.5.3). The SRAM array uses a small decoder and bitlines connected to relatively few memory cells, so it has a shorter critical path than a large memory does.

Code Example 6.4 shows the `add` instruction with register operands. MIPS register names are preceded by the \$ sign. The variables `a`, `b`, and `c` are arbitrarily placed in `$s0`, `$s1`, and `$s2`. The name `$s1` is pronounced “register s1” or “dollar s1”. The instruction adds the 32-bit values contained in `$s1` (`b`) and `$s2` (`c`) and writes the 32-bit result to `$s0` (`a`).

MIPS generally stores variables in 18 of the 32 registers: `$s0 – $s7`, and `$t0 – $t9`. Register names beginning with `$s` are called *saved* registers. Following MIPS convention, these registers store variables such as `a`, `b`, and `c`. Saved registers have special connotations when they are used with procedure calls (see Section 6.4.6). Register names beginning with `$t` are called *temporary* registers. They are used for storing temporary variables. Code Example 6.5 shows MIPS assembly code using a temporary register, `$t0`, to store the intermediate calculation of `c – d`.

**Code Example 6.4 REGISTER OPERANDS**

High-Level Code	MIPS Assembly Code
<code>a = b + c;</code>	# \$s0 = a, \$s1 = b, \$s2 = c add \$s0, \$s1, \$s2      # a = b + c

**Code Example 6.5 TEMPORARY REGISTERS**

High-Level Code	MIPS Assembly Code
<code>a = b + c - d;</code>	# \$s0 = a, \$s1 = b, \$s2 = c, \$s3 = d  sub \$t0, \$s2, \$s3      # t = c - d add \$s0, \$s1, \$t0      # a = b + t

---

**Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE**

Translate the following high-level code into assembly language. Assume variables a–c are held in registers \$s0–\$s2 and f–j are in \$s3–\$s7.

```
a = b - c;
f = (g + h) - (i + j);
```

**Solution:** The program uses four assembly language instructions.

```
# MIPS assembly code
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h,
# $s6 = i, $s7 = j
sub $s0, $s1, $s2    # a = b - c
add $t0, $s4, $s5    # $t0 = g + h
add $t1, $s6, $s7    # $t1 = i + j
sub $s3, $t0, $t1    # f = (g + h) - (i + j)
```

---

### The Register Set

The MIPS architecture defines 32 registers. Each register has a name and a number ranging from 0 to 31. Table 6.1 lists the name, number, and use for each register. \$0 always contains the value 0 because this constant is so frequently used in computer programs. We have also discussed the \$s and \$t registers. The remaining registers will be described throughout this chapter.

**Table 6.1 MIPS register set**

Name	Number	Use
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2–3	procedure return values
\$a0-\$a3	4–7	procedure arguments
\$t0-\$t7	8–15	temporary variables
\$s0-\$s7	16–23	saved variables
\$t8-\$t9	24–25	temporary variables
\$k0-\$k1	26–27	operating system (OS) temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	procedure return address

Word Address	Data	
...	...	...
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

**Figure 6.1 Word-addressable memory**

### Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 32 variables. However, data can also be stored in memory. When compared to the register file, memory has many data locations, but accessing it takes a longer amount of time. Whereas the register file is small and fast, memory is large and slow. For this reason, commonly used variables are kept in registers. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. As described in Section 5.5, memories are organized as an array of data words. The MIPS architecture uses 32-bit memory addresses and 32-bit data words.

MIPS uses a byte-addressable memory. That is, each byte in memory has a unique address. However, for explanation purposes only, we first introduce a word-addressable memory, and afterward describe the MIPS byte-addressable memory.

Figure 6.1 shows a memory array that is *word-addressable*. That is, each 32-bit data word has a unique 32-bit address. Both the 32-bit word address and the 32-bit data value are written in hexadecimal in Figure 6.1. For example, data 0xF2F1AC07 is stored at memory address 1. Hexadecimal constants are written with the prefix 0x. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

MIPS uses the *load word* instruction, `lw`, to read a data word from memory into a register. Code Example 6.6 loads memory word 1 into `$s3`.

The `lw` instruction specifies the *effective address* in memory as the sum of a *base address* and an *offset*. The base address (written in parentheses in the instruction) is a register. The offset is a constant (written before the parentheses). In Code Example 6.6, the base address

---

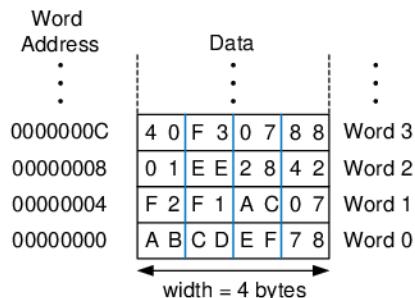
#### Code Example 6.6 READING WORD-ADDRESSABLE MEMORY

##### Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
lw $s3, 1($0)      # read memory word 1 into $s3
```

**Code Example 6.7 WRITING WORD-ADDRESSABLE MEMORY****Assembly Code**

```
# This assembly code (unlike MIPS) assumes word-addressable memory
sw    $s7, 5($0)      # write $s7 to memory word 5
```



**Figure 6.2** Byte-addressable memory

is \$0, which holds the value 0, and the offset is 1, so the `lw` instruction reads from memory address  $(\$0 + 1) = 1$ . After the load word instruction (`lw`) is executed, `$s3` holds the value 0xF2F1AC07, which is the data value stored at memory address 1 in Figure 6.1.

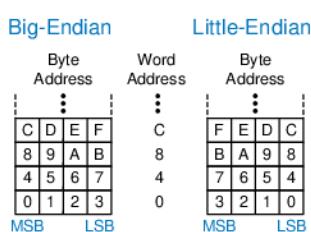
Similarly, MIPS uses the *store word* instruction, `sw`, to write a data word from a register into memory. Code Example 6.7 writes the contents of register `$s7` into memory word 5. These examples have used `$0` as the base address for simplicity, but remember that any register can be used to supply the base address.

The previous two code examples have shown a computer architecture with a word-addressable memory. The MIPS memory model, however, is byte-addressable, *not* word-addressable. Each data byte has a unique address. A 32-bit word consists of four 8-bit bytes. So each word address is a multiple of 4, as shown in Figure 6.2. Again, both the 32-bit word address and the data value are given in hexadecimal.

Code Example 6.8 shows how to read and write words in the MIPS byte-addressable memory. The word address is four times the word number. The MIPS assembly code reads words 0, 2, and 3 and writes words 1, 8, and 100. The offset can be written in decimal or hexadecimal.

The MIPS architecture also provides the `lb` and `sb` instructions that load and store single bytes in memory rather than words. They are similar to `lw` and `sw` and will be discussed further in Section 6.4.5.

Byte-addressable memories are organized in a *big-endian* or *little-endian* fashion, as shown in Figure 6.3. In both formats, the *most significant byte* (MSB) is on the left and the *least significant byte* (LSB) is on the right. In big-endian machines, bytes are numbered starting with 0



**Figure 6.3** Big- and little-endian memory addressing

**Code Example 6.8** ACCESSING BYTE-ADDRESSABLE MEMORY**MIPS Assembly Code**

```
lw $s0, 0($0)      # read data word 0 (0xABCD E F78) into $s0
lw $s1, 8($0)       # read data word 2 (0x01EE 2842) into $s1
lw $s2, 0xC($0)     # read data word 3 (0x40F3 0788) into $s2
sw $s3, 4($0)       # write $s3 to data word 1
sw $s4, 0x20($0)    # write $s4 to data word 8
sw $s5, 400($0)     # write $s5 to data word 100
```

at the big (most significant) end. In little-endian machines, bytes are numbered starting with 0 at the little (least significant) end. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word differ.

**Example 6.2** BIG- AND LITTLE-ENDIAN MEMORY

Suppose that \$s0 initially contains 0x23456789. After the following program is run on a big-endian system, what value does \$s0 contain? In a little-endian system? `lb $s0, 1($0)` loads the data at byte address  $(1 + \$0) = 1$  into the least significant byte of \$s0. `lb` is discussed in detail in Section 6.4.5.

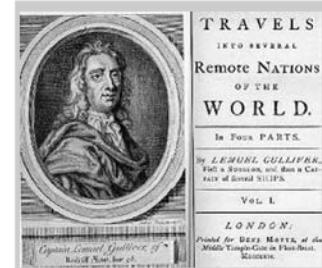
```
sw $s0, 0($0)
lb $s0, 1($0)
```

**Solution:** Figure 6.4 shows how big- and little-endian machines store the value 0x23456789 in memory word 0. After the load byte instruction, `lb $s0, 1($0)`, \$s0 would contain 0x00000045 on a big-endian system and 0x00000067 on a little-endian system.



**Figure 6.4** Big-endian and little-endian data storage

IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's IA-32 architecture (found in PCs) uses little-endian addressing. Some MIPS processors are little-endian, and some are big-endian.<sup>1</sup> The choice of endianness is completely arbitrary but leads to hassles when sharing data between big-endian and little-endian computers. In examples in this text, we will use little-endian format whenever byte ordering matters.



The terms big-endian and little-endian come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

The terms were first applied to computer architectures by Danny Cohen in his paper "On Holy Wars and a Plea for Peace" published on April Fools Day, 1980 (USC/ISI IEN 137). (Photo courtesy The Brotherton Collection, IEEDS University Library.)

<sup>1</sup> SPIM, the MIPS simulator that comes with this text, uses the endianness of the machine it is run on. For example, when using SPIM on an Intel IA-32 machine, the memory is little-endian. With an older Macintosh or Sun SPARC machine, memory is big-endian.

In the MIPS architecture, word addresses for `lw` and `sw` must be *word aligned*. That is, the address must be divisible by 4. Thus, the instruction `lw $s0, 7($0)` is an illegal instruction. Some architectures, such as IA-32, allow non-word-aligned data reads and writes, but MIPS requires strict alignment for simplicity. Of course, byte addresses for load byte and store byte, `lb` and `sb`, need not be word aligned.

#### **Constants/Immediates**

Load word and store word, `lw` and `sw`, also illustrate the use of *constants* in MIPS instructions. These constants are called *immediates*, because their values are immediately available from the instruction and do not require a register or memory access. Add immediate, `addi`, is another common MIPS instruction that uses an immediate operand. `addi` adds the immediate specified in the instruction to a value in a register, as shown in Code Example 6.9.

The immediate specified in an instruction is a 16-bit two's complement number in the range  $[-32768, 32767]$ . Subtraction is equivalent to adding a negative number, so, in the interest of simplicity, there is no `subi` instruction in the MIPS architecture.

Recall that the `add` and `sub` instructions use three register operands. But the `lw`, `sw`, and `addi` instructions use two register operands and a constant. Because the instruction formats differ, `lw` and `sw` instructions violate design principle 1: simplicity favors regularity. However, this issue allows us to introduce the last design principle:

---

#### **Code Example 6.9 IMMEDIATE OPERANDS**

High-Level Code	MIPS Assembly Code
<pre>a = a + 4; b = a - 12;</pre>	<pre># \$s0 = a, \$s1 = b addi \$s0, \$s0, 4          # a = a + 4 addi \$s1, \$s0, -12        # b = a - 12</pre>

**Design Principle 4:** Good design demands good compromises.

A single instruction format would be simple but not flexible. The MIPS instruction set makes the compromise of supporting three instruction formats. One format, used for instructions such as `add` and `sub`, has three register operands. Another, used for instructions such as `lw` and `addi`, has two register operands and a 16-bit immediate. A third, to be discussed later, has a 26-bit immediate and no registers. The next section discusses the three MIPS instruction formats and shows how they are encoded into binary.

## 6.3 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's, called *machine language*.

MIPS uses 32-bit instructions. Again, simplicity favors regularity, and the most regular choice is to encode all instructions as words that can be stored in memory. Even though some instructions may not require all 32 bits of encoding, variable-length instructions would add too much complexity. Simplicity would also encourage a single instruction format, but, as already mentioned, that is too restrictive. MIPS makes the compromise of defining three instruction formats: R-type, I-type, and J-type. This small number of formats allows for some regularity among all the types, and thus simpler hardware, while also accommodating different instruction needs, such as the need to encode large constants in the instruction. *R-type* instructions operate on three registers. *I-type* instructions operate on two registers and a 16-bit immediate. *J-type* (jump) instructions operate on one 26-bit immediate. We introduce all three formats in this section but leave the discussion of J-type instructions for Section 6.4.2.

### 6.3.1 R-type Instructions

The name R-type is short for *register-type*. R-type instructions use three registers as operands: two as sources, and one as a destination. Figure 6.5 shows the R-type machine instruction format. The 32-bit instruction has six fields: op, rs, rt, rd, shamt, and funct. Each field is five or six bits, as indicated.

The operation the instruction performs is encoded in the two fields highlighted in blue: op (also called opcode or operation code) and funct (also called the function). All R-type instructions have an opcode of 0. The specific R-type operation is determined by the funct field. For example, the opcode and funct fields for the add instruction are 0 ( $00000_2$ ) and 32 ( $100000_2$ ), respectively. Similarly, the sub instruction has an opcode and funct field of 0 and 34.

The operands are encoded in the three fields: rs, rt, and rd. The first two registers, rs and rt, are the source registers; rd is the destination



**Figure 6.5** R-type machine instruction format

rs is short for “register source.” rt comes after rs alphabetically and usually indicates the second register source.

register. The fields contain the register numbers that were given in Table 6.1. For example, \$s0 is register 16.

The fifth field, shamt, is used only in shift operations. In those instructions, the binary value stored in the 5-bit shamt field indicates the amount to shift. For all other R-type instructions, shamt is 0.

Figure 6.6 shows the machine code for the R-type instructions add and sub. Notice that the destination is the first register in an assembly language instruction, but it is the third register field (rd) in the machine language instruction. For example, the assembly instruction add \$s0, \$s1, \$s2 has rs = \$s1 (17), rt = \$s2 (18), and rd = \$s0 (16).

Tables B.1 and B.2 in Appendix B define the opcode values for all MIPS instructions and the funct field values for R-type instructions.

Assembly Code						Field Values						Machine Code					
	op	rs	rt	rd	shamt	funct		op	rs	rt	rd	shamt	funct				
add \$s0, \$s1, \$s2	0	17	18	16	0	32		000000	10001	10010	10000	00000	1000000	(0x02328020)			
sub \$t0, \$t3, \$t5	0	11	13	8	0	34		000000	01011	01101	01000	00000	100010	(0x016D4022)			

6 bits    5 bits    5 bits    5 bits    5 bits    6 bits    6 bits    5 bits    5 bits    5 bits    5 bits    5 bits    6 bits

**Figure 6.6 Machine code for R-type instructions**

---

### Example 6.3 TRANSLATING ASSEMBLY LANGUAGE TO MACHINE LANGUAGE

Translate the following assembly language statement into machine language.

add \$t0, \$s4, \$s5

**Solution:** According to Table 6.1, \$t0, \$s4, and \$s5 are registers 8, 20, and 21. According to Tables B.1 and B.2, add has an opcode of 0 and a funct code of 32. Thus, the fields and machine code are given in Figure 6.7. The easiest way to write the machine language in hexadecimal is to first write it in binary, then look at consecutive groups of four bits, which correspond to hexadecimal digits (indicated in blue). Hence, the machine language instruction is 0x02954020.

---

Assembly Code						Field Values						Machine Code					
	op	rs	rt	rd	shamt	funct		op	rs	rt	rd	shamt	funct				
add \$t0, \$s4, \$s5	0	20	21	8	0	32		000000	10100	10101	01000	00000	1000000	(0x02954020)			

6 bits    5 bits    5 bits    5 bits    5 bits    6 bits    0    2    9    5    4    0    2    0

**Figure 6.7 Machine code for the R-type instruction of Example 6.3**

### 6.3.2 I-Type Instructions

The name I-type is short for *immediate-type*. I-type instructions use two register operands and one immediate operand. Figure 6.8 shows the I-type machine instruction format. The 32-bit instruction has four fields: op, rs, rt, and imm. The first three fields, op, rs, and rt, are like those of R-type instructions. The imm field holds the 16-bit immediate.

The operation is determined solely by the opcode, highlighted in blue. The operands are specified in the three fields, rs, rt, and imm. rs and imm are always used as source operands. rt is used as a destination for some instructions (such as addi and lw) but as another source for others (such as sw).

Figure 6.9 shows several examples of encoding I-type instructions. Recall that negative immediate values are represented using 16-bit two's complement notation. rt is listed first in the assembly language instruction when it is used as a destination, but it is the second register field in the machine language instruction.

I-type			
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Assembly Code	Field Values				Machine Code				
	op	rs	rt	imm	op	rs	rt	imm	
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	
addi \$s0, \$s1, 5	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
addi \$t0, \$s3, -12	8	19	8	-12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
lw \$t2, 32(\$s0)	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
sw \$s1, 4(\$t1)	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)

**Figure 6.8 I-type instruction format**

Figure 6.9 Machine code for I-type instructions

---

#### Example 6.4 TRANSLATING I-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Translate the following I-type instruction into machine code.

lw \$s3, -24(\$s4)

**Solution:** According to Table 6.1, \$s3 and \$s4 are registers 19 and 20, respectively. Table B.1 indicates that lw has an opcode of 35. rs specifies the base address, \$s4, and rt specifies the destination register, \$s3. The immediate, imm, encodes the 16-bit offset, -24. Thus, the fields and machine code are given in Figure 6.10.

Assembly Code	Field Values	Machine Code
	op      rs      rt      imm	op      rs      rt      imm
lw \$s3, -24(\$s4)	35      20      19      -24	100011 10100 10011 11111111101000
	6 bits    5 bits    5 bits    16 bits	8    E    9    3    F    F    E    8

**Figure 6.10** Machine code for the I-type instruction

I-type instructions have a 16-bit immediate field, but the immediates are used in 32-bit operations. For example, `lw` adds a 16-bit offset to a 32-bit base register. What should go in the upper half of the 32 bits? For positive immediates, the upper half should be all 0's, but for negative immediates, the upper half should be all 1's. Recall from Section 1.4.6 that this is called *sign extension*. An  $N$ -bit two's complement number is sign-extended to an  $M$ -bit number ( $M > N$ ) by copying the sign bit (most significant bit) of the  $N$ -bit number into all of the upper bits of the  $M$ -bit number. Sign-extending a two's complement number does not change its value.

Most MIPS instructions sign-extend the immediate. For example, `addi`, `lw`, and `sw` do sign extension to support both positive and negative immediates. An exception to this rule is that logical operations (`andi`, `ori`, `xori`) place 0's in the upper half; this is called *zero extension* rather than sign extension. Logical operations are discussed further in Section 6.4.1.

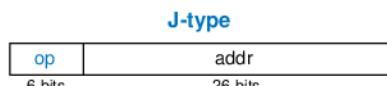
### 6.3.3 J-type Instructions

The name J-type is short for *jump-type*. This format is used only with jump instructions (see Section 6.4.2). This instruction format uses a single 26-bit address operand, `addr`, as shown in Figure 6.11. Like other formats, J-type instructions begin with a 6-bit opcode. The remaining bits are used to specify an address, `addr`. Further discussion and machine code examples of J-type instructions are given in Sections 6.4.2 and 6.5.

### 6.3.4 Interpreting Machine Language Code

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all formats start with a 6-bit opcode field. Thus, the best place to begin is to look at the opcode. If it is 0, the instruction is R-type; otherwise it is I-type or J-type.

**Figure 6.11** J-type instruction format



---

**Example 6.5 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE**

Translate the following machine language code into assembly language.

0x2237FFF1  
0x02F34022

**Solution:** First, we represent each instruction in binary and look at the six most significant bits to find the opcode for each instruction, as shown in Figure 6.12. The opcode determines how to interpret the rest of the bits. The opcodes are  $001000_2$  ( $8_{10}$ ) and  $000000_2$  ( $0_{10}$ ), indicating an addi and R-type instruction, respectively. The funct field of the R-type instruction is  $100010_2$  ( $34_{10}$ ), indicating that it is a sub instruction. Figure 6.12 shows the assembly code equivalent of the two machine instructions.

---

	Machine Code						Field Values				Assembly Code	
	op	rs	rt	imm			op	rs	rt	imm		
(0x2237FFF1)	001000	10001	10111	1111	1111	1111 0001	8	17	23	-15	addi \$s7, \$s1, -15	
	2	2	3	7	F	F F 1						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct
(0x02F34022)	000000	10111	10011	01000	00000	100010	0	23	19	8	0	34
	0	2	F	3	4	0	2	2	2			

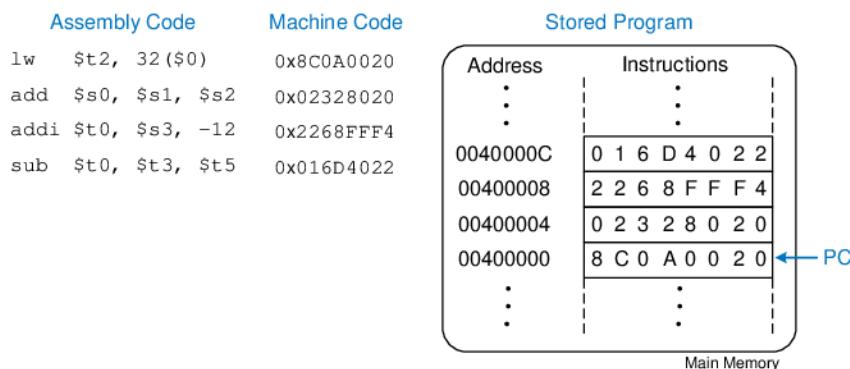
Figure 6.12 Machine code to assembly code translation

### 6.3.5 The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful. Running a different program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. Instead of dedicated hardware, the stored program offers *general purpose* computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simplified to a series of memory reads and instruction executions.

Figure 6.13 shows how machine instructions are stored in memory. In MIPS programs, the instructions are normally stored starting at address 0x00400000. Remember that MIPS memory is byte addressable, so 32-bit (4-byte) instruction addresses advance by 4 bytes, not 1.

**Figure 6.13** Stored program

To run or *execute* the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the *program counter* (PC). The PC is separate from the 32 registers shown previously in Table 6.1.

To execute the code in Figure 6.13, the operating system sets the PC to address 0x00400000. The processor reads the instruction at that memory address and executes the instruction, 0x8C0A0020. The processor then increments the PC by 4, to 0x00400004, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds the state of a program. For MIPS, the architectural state consists of the register file and PC. If the operating system saves the architectural state at some point in the program, it can interrupt the program, do something else, then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.



Ada Lovelace, 1815–1852.  
Wrote the first computer program. It calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the only legitimate child of the poet Lord Byron.

## 6.4 PROGRAMMING

Software languages such as C or Java are called high-level programming languages, because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs such as arithmetic and logical operations, if/else statements, for and while loops, array indexing, and procedure calls. In this section, we explore how to translate these high-level constructs into MIPS assembly code.

### 6.4.1 Arithmetic/Logical Instructions

The MIPS architecture defines a variety of arithmetic and logical instructions. We introduce these instructions briefly here, because they are necessary to implement higher-level constructs.

Source Registers									
	\$s1	1111	1111	1111	1111	0000	0000	0000	0000
	\$s2	0100	0110	1010	0001	1111	0000	1011	0111
Assembly Code								Result	
and \$s3, \$s1, \$s2	\$s3	0100	0110	1010	0001	0000	0000	0000	0000
or \$s4, \$s1, \$s2	\$s4	1111	1111	1111	1111	1111	0000	1011	0111
xor \$s5, \$s1, \$s2	\$s5	1011	1001	0101	1110	1111	0000	1011	0111
nor \$s6, \$s1, \$s2	\$s6	0000	0000	0000	0000	0000	1111	0100	1000

Figure 6.14 Logical operations

### Logical Instructions

MIPS logical operations include `and`, `or`, `xor`, and `nor`. These R-type instructions operate bit-by-bit on two source registers and write the result to the destination register. Figure 6.14 shows examples of these operations on the two source values 0xFFFFF0000 and 0x46A1F0B7. The figure shows the values stored in the destination register, `rd`, after the instruction executes.

The `and` instruction is useful for *masking* bits (i.e., forcing unwanted bits to 0). For example, in Figure 6.14, 0xFFFFF0000 AND 0x46A1F0B7 = 0x46A10000. The `and` instruction masks off the bottom two bytes and places the unmasked top two bytes of `$s2`, 0x46A1, in `$s3`. Any subset of register bits can be masked.

The `or` instruction is useful for combining bits from two registers. For example, 0x347A0000 OR 0x0000072FC = 0x347A72FC, a combination of the two values.

MIPS does not provide a NOT instruction, but `A NOR $0 = NOT A`, so the `NOR` instruction can substitute.

Logical operations can also operate on immediates. These I-type instructions are `andi`, `ori`, and `xori`. `nori` is not provided, because the same functionality can be easily implemented using the other instructions, as will be explored in Exercise 6.11. Figure 6.15 shows examples of the `andi`, `ori`, and `xori` instructions. The figure gives the values of

Source Values									
	\$s1	0000	0000	0000	0000	0000	0000	1111	1111
	imm	0000	0000	0000	0000	1111	1010	0011	0100
Assembly Code								Result	
andi \$s2, \$s1, 0xFA34	\$s2	0000	0000	0000	0000	0000	0011	0100	
ori \$s3, \$s1, 0xFA34	\$s3	0000	0000	0000	0000	1111	1010	1111	1111
xori \$s4, \$s1, 0xFA34	\$s4	0000	0000	0000	0000	1111	1010	1100	1011

Figure 6.15 Logical operations with immediates

the source register and immediate, and the value of the destination register, *rt*, after the instruction executes. Because these instructions operate on a 32-bit value from a register and a 16-bit immediate, they first zero-extend the immediate to 32 bits.

### Shift Instructions

Shift instructions shift the value in a register left or right by up to 31 bits. Shift operations multiply or divide by powers of two. MIPS shift operations are *sll* (shift left logical), *srl* (shift right logical), and *sra* (shift right arithmetic).

As discussed in Section 5.2.5, left shifts always fill the least significant bits with 0's. However, right shifts can be either logical (0's shift into the most significant bits) or arithmetic (the sign bit shifts into the most significant bits). Figure 6.16 shows the machine code for the R-type instructions *sll*, *srl*, and *sra*. *rt* (i.e., *\$s1*) holds the 32-bit value to be shifted, and *shamt* gives the amount by which to shift (4). The shifted result is placed in *rd*.

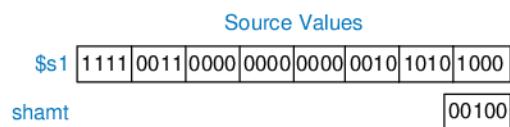
Figure 6.17 shows the register values for the shift instructions *sll*, *srl*, and *sra*. Shifting a value left by *N* is equivalent to multiplying it by  $2^N$ . Likewise, arithmetically shifting a value right by *N* is equivalent to dividing it by  $2^N$ , as discussed in Section 5.2.5.

MIPS also has variable-shift instructions: *sllv* (shift left logical variable), *srlv* (shift right logical variable), and *sraw* (shift right arithmetic variable). Figure 6.18 shows the machine code for these instructions.

Assembly Code		Field Values						Machine Code					
		op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct
<i>sll \$t0, \$s1, 4</i>		0	0	17	8	4	0	000000	00000	10001	01000	00100	000000
<i>srl \$s2, \$s1, 4</i>		0	0	17	18	4	2	000000	00000	10001	10010	00100	000010
<i>sra \$s3, \$s1, 4</i>		0	0	17	19	4	3	000000	00000	10001	10011	00100	000011

6 bits    5 bits    5 bits    5 bits    5 bits    6 bits    6 bits    5 bits    5 bits    5 bits    5 bits    5 bits    6 bits

**Figure 6.16 Shift instruction machine code**



**Figure 6.17 Shift operations**

Assembly Code		Result							
<i>sll \$t0, \$s1, 4</i>	<b>\$t0</b>	0011	0000	0000	0000	0010	1010	1000	<b>0000</b>
<i>srl \$s2, \$s1, 4</i>	<b>\$s2</b>	0000	1111	0011	0000	0000	0000	0010	1010
<i>sra \$s3, \$s1, 4</i>	<b>\$s3</b>	1111	1111	0011	0000	0000	0000	0010	1010

Assembly Code						Field Values						Machine Code					
op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct
sllv \$s3, \$s1, \$s2	0	18	17	19	0	000000	10010	10001	10011	00000	000100	(0x02519804)					
srlv \$s4, \$s1, \$s2	0	18	17	20	0	000000	10010	10001	10100	00000	000110	(0x0251A006)					
srav \$s5, \$s1, \$s2	0	18	17	21	0	000000	10010	10001	10101	00000	000111	(0x0251A807)					

Figure 6.18 Variable-shift instruction machine code

Source Values									
\$s1									1000
\$s2									1000
Assembly Code									
sllv \$s3, \$s1, \$s2									\$s3
srlv \$s4, \$s1, \$s2									\$s4
srav \$s5, \$s1, \$s2									\$s5
Result									
0000 0100 0000 0010 1010 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000									0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000									1111 1111 1111 0011 0000 0100 0000 0010 1010 1000 0000 0000 0000 0000 0000 0000 0000 0000
1111 1111 1111 0011 0000 0100 0000 0010 1010 1000 0000 0000 0000 0000 0000 0000 0000 0000									1111 1111 1111 0011 0000 0100 0000 0010 1010 1000 0000 0000 0000 0000 0000 0000 0000 0000

Figure 6.19 Variable-shift operations

rt (i.e., \$s1) holds the value to be shifted, and the five least significant bits of rs (i.e., \$s2) give the amount to shift. The shifted result is placed in rd, as before. The shamt field is ignored and should be all 0's. Figure 6.19 shows register values for each type of variable-shift instruction.

#### Generating Constants

The addi instruction is helpful for assigning 16-bit constants, as shown in Code Example 6.10.

---

#### Code Example 6.10 16-BIT CONSTANT

High-Level Code	MIPS Assembly code
int a = 0x4f3c;	# \$s0 = a addi \$s0, \$0, 0x4f3c # a = 0x4f3c

---

#### Code Example 6.11 32-BIT CONSTANT

High-Level Code	MIPS Assembly Code
int a = 0x6d5e4f3c;	# \$s0 = a lui \$s0, 0x6d5e          # a = 0x6d5e0000 ori \$s0, \$s0, 0x4f3c      # a = 0x6d5e4f3c

The `int` data type in C refers to a word of data representing a two's complement integer. MIPS uses 32-bit words, so an `int` represents a number in the range  $[-2^{31}, 2^{31}-1]$ .

`hi` and `lo` are not among the usual 32 MIPS registers, so special instructions are needed to access them. `mfhi $s2` (move from `hi`) copies the value in `hi` to `$s2`. `mflo $s3` (move from `lo`) copies the value in `lo` to `$s3`. `hi` and `lo` are technically part of the architectural state; however, we generally ignore these registers in this book.

To assign 32-bit constants, use a load upper immediate instruction (`lui`) followed by an or immediate (`ori`) instruction, as shown in Code Example 6.11. `lui` loads a 16-bit immediate into the upper half of a register and sets the lower half to 0. As mentioned earlier, `ori` merges a 16-bit immediate into the lower half.

#### Multiplication and Division Instructions\*

Multiplication and division are somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. Dividing two 32-bit numbers produces a 32-bit quotient and a 32-bit remainder.

The MIPS architecture has two special-purpose registers, `hi` and `lo`, which are used to hold the results of multiplication and division. `mult $s0, $s1` multiplies the values in `$s0` and `$s1`. The 32 most significant bits are placed in `hi` and the 32 least significant bits are placed in `lo`. Similarly, `div $s0, $s1` computes  $\$s0/\$s1$ . The quotient is placed in `lo` and the remainder is placed in `hi`.

#### 6.4.2 Branching

An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, `if/else` statements, case statements, while loops, and for loops all conditionally execute code depending on some test.

To sequentially execute instructions, the program counter increments by 4 after each instruction. *Branch* instructions modify the program counter to skip over sections of code or to go back to repeat previous code. *Conditional branch* instructions perform a test and branch only if the test is TRUE. *Unconditional branch* instructions, called *jumps*, always branch.

##### Conditional Branches

The MIPS instruction set has two conditional branch instructions: branch if equal (`beq`) and branch if not equal (`bne`). `beq` branches when the values in two registers are equal, and `bne` branches when they are not equal. Code Example 6.12 illustrates the use of `beq`. Note that branches are written as `beq $rs, $rt, imm`, where `$rs` is the first source register. This order is reversed from most I-type instructions.

When the program in Code Example 6.12 reaches the branch if equal instruction (`beq`), the value in `$s0` is equal to the value in `$s1`, so the branch is *taken*. That is, the next instruction executed is the `add` instruction just after the *label* called `target`. The two instructions directly after the branch and before the label are not executed.

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these

---

**Code Example 6.12** CONDITIONAL BRANCHING USING `beq`**MIPS Assembly Code**

```

addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll $s1, $s1, 2      # $s1 = 1 << 2 = 4
beq $s0, $s1, target # $s0 == $s1, so branch is taken
addi $s1, $s1, 1      # not executed
sub $s1, $s1, $s0      # not executed

target:
add $s1, $s1, $s0      # $s1 = 4 + 4 = 8

```

---

labels are translated into instruction addresses (see Section 6.5). MIPS assembly labels are followed by a (:) and cannot use reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help make labels stand out.

Code Example 6.13 shows an example using the branch if not equal instruction (`bne`). In this case, the branch is *not taken* because `$s0` is equal to `$s1`, and the code continues to execute directly after the `bne` instruction. All instructions in this code snippet are executed.

---

**Code Example 6.13** CONDITIONAL BRANCHING USING `bne`**MIPS Assembly Code**

```

addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll $s1, $s1, 2      # $s1 = 1 << 2 = 4
bne $s0, $s1, target # $s0 == $s1, so branch is not taken
addi $s1, $s1, 1      # $s1 = 4 + 1 = 5
sub $s1, $s1, $s0      # $s1 = 5 - 4 = 1

target:
add $s1, $s1, $s0      # $s1 = 1 + 4 = 5

```

---

**Jump**

A program can unconditionally branch, or *jump*, using the three types of jump instructions: `jump (j)`, `jump and link (jal)`, and `jump register (jr)`. `Jump (j)` jumps directly to the instruction at the specified label. `Jump and link (jal)` is similar to `j` but is used by procedures to save a return address, as will be discussed in Section 6.4.6. `Jump register (jr)` jumps to the address held in a register. Code Example 6.14 shows the use of the `jump` instruction (`j`).

After the `j target` instruction, the program in Code Example 6.14 unconditionally continues executing the `add` instruction at the label `target`. All of the instructions between the jump and the label are skipped.

`j` and `jal` are J-type instructions. `jr` is an R-type instruction that uses only the `rs` operand.

**Code Example 6.14 UNCONDITIONAL BRANCHING USING j****MIPS Assembly Code**

```

addi $s0, $0, 4      # $s0 = 4
addi $s1, $0, 1      # $s1 = 1
j target            # jump to target
addi $s1, $s1, 1      # not executed
sub $s1, $s1, $s0      # not executed
target:
add $s1, $s1, $s0      # $s1 = 1 + 4 = 5

```

**Code Example 6.15 UNCONDITIONAL BRANCHING USING jr****MIPS Assembly Code**

```

0x00002000 addi $s0, $0, 0x2010 # $s0 = 0x2010
0x00002004 jr $s0                # jump to 0x00002010
0x00002008 addi $s1, $0, 1      # not executed
0x0000200c sra $s1, $s1, 2      # not executed
0x00002010 lw $s3, 44($s1)      # executed after jr instruction

```

Code Example 6.15 shows the use of the jump register instruction (`jr`). Instruction addresses are given to the left of each instruction. `jr $s0` jumps to the address held in `$s0`, 0x00002010.

**6.4.3 Conditional Statements**

`if` statements, `if/else` statements, and `case` statements are conditional statements commonly used by high-level languages. They each conditionally execute a *block* of code consisting of one or more instructions. This section shows how to translate these high-level constructs into MIPS assembly language.

**If Statements**

An `if` statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.16 shows how to translate an `if` statement into MIPS assembly code.

**Code Example 6.16 if STATEMENT****High-Level Code**

```

if (i == j)
    f = g + h;
f = f - i;

```

**MIPS Assembly Code**

```

# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
bne $s3, $s4, L1      # if i != j, skip if block
add $s0, $s1, $s2      # if block: f = g + h
L1:
sub $s0, $s0, $s3      # f = f - i

```

The assembly code for the `if` statement tests the opposite condition of the one in the high-level code. In Code Example 6.16, the high-level code tests for `i == j`, and the assembly code tests for `i != j`. The `bne` instruction branches (skips the `if` block) when `i != j`. Otherwise, `i == j`, the branch is not taken, and the `if` block is executed as desired.

#### If/Else Statements

`if/else` statements execute one of two blocks of code depending on a condition. When the condition in the `if` statement is met, the *if block* is executed. Otherwise, the *else block* is executed. Code Example 6.17 shows an example `if/else` statement.

Like `if` statements, `if/else` assembly code tests the opposite condition of the one in the high-level code. For example, in Code Example 6.17, the high-level code tests for `i == j`. The assembly code tests for the opposite condition (`i != j`). If that opposite condition is TRUE, `bne` skips the `if` block and executes the `else` block. Otherwise, the `if` block executes and finishes with a jump instruction (`j`) to jump past the `else` block.

---

#### Code Example 6.17 if/else STATEMENT

##### High-Level Code

```
if (i == j)
    f = g + h;

else
    f = f - i;
```

##### MIPS Assembly Code

```
# $s0 = f, $s1 = g,    $s2 = h, $s3 = i, $s4 = j
bne $s3, $s4, else      # if i != j, branch to else
add $s0, $s1, $s2        # if block: f = g + h
j L2                      # skip past the else block
else:
    sub $s0, $s0, $s3      # else block: f = f - i
L2:
```

#### Switch/Case Statements\*

`switch/case` statements execute one of several blocks of code depending on the conditions. If no conditions are met, the `default` block is executed. A `case` statement is equivalent to a series of *nested if/else* statements. Code Example 6.18 shows two high-level code snippets with the same functionality: they calculate the fee for an *ATM (automatic teller machine)* withdrawal of \$20, \$50, or \$100, as defined by `amount`. The MIPS assembly implementation is the same for both high-level code snippets.

#### 6.4.4 Getting Loopy

Loops repeatedly execute a block of code depending on a condition. `for` loops and `while` loops are common loop constructs used by high-level languages. This section shows how to translate them into MIPS assembly language.

**Code Example 6.18** switch/case STATEMENT

High-Level Code	MIPS Assembly Code
<pre> switch (amount) {     case 20: fee = 2; break;     case 50: fee = 3; break;     case 100: fee = 5; break;     default: fee = 0; }  // equivalent function using if/else statements if      (amount == 20)  fee = 2; else if (amount == 50)  fee = 3; else if (amount == 100) fee = 5; else                  fee = 0; </pre>	<pre> # \$s0 = amount, \$s1 = fee case20:     addi \$t0, \$0, 20      # \$t0 = 20     bne \$s0, \$t0, case50 # i == 20? if not,                           # skip to case50     addi \$s1, \$0, 2        # if so, fee = 2     j done                # and break out of case  case50:     addi \$t0, \$0, 50      # \$t0 = 50     bne \$s0, \$t0, casel00 # i == 50? if not,                           # skip to casel00     addi \$s1, \$0, 3        # if so, fee = 3     j done                # and break out of case  casel00:     addi \$t0, \$0, 100     # \$t0 = 100     bne \$s0, \$t0, default # i == 100? if not,                           # skip to default     addi \$s1, \$0, 5        # if so, fee = 5     j done                # and break out of case  default:     add \$s1, \$0, \$0        # charge = 0  done: </pre>

**While Loops**

while loops repeatedly execute a block of code until a condition is *not* met. The while loop in Code Example 6.19 determines the value of  $x$  such that  $2^x = 128$ . It executes seven times, until  $\text{pow} = 128$ .

Like if/else statements, the assembly code for while loops tests the opposite condition of the one given in the high-level code. If that opposite condition is TRUE, the while loop is finished.

**Code Example 6.19** while LOOP

High-Level Code	MIPS Assembly Code
<pre> int pow = 1; int x   = 0;  while (pow != 128) {     pow = pow * 2;     x = x + 1; } </pre>	<pre> # \$s0 = pow, \$s1 = x addi \$s0, \$0, 1      # pow = 1 addi \$s1, \$0, 0      # x = 0  addi \$t0, \$0, 128    # t0 = 128 for comparison while:     beq \$s0, \$t0, done # if pow == 128, exit while     sll \$s0, \$s0, 1    # pow = pow * 2     addi \$s1, \$s1, 1    # x = x + 1     j while done: </pre>

In Code Example 6.19, the while loop compares pow to 128 and exits the loop if it is equal. Otherwise it doubles pow (using a left shift), increments x, and jumps back to the start of the while loop.

### For Loops

for loops, like while loops, repeatedly execute a block of code until a condition is *not* met. However, for loops add support for a *loop variable*, which typically keeps track of the number of loop executions. A general format of the for loop is

```
for (initialization; condition; loop operation)
```

The *initialization* code executes before the for loop begins. The *condition* is tested at the beginning of each loop. If the condition is not met, the loop exits. The *loop operation* executes at the end of each loop.

Code Example 6.20 adds the numbers from 0 to 9. The loop variable, i, is initialized to 0 and is incremented at the end of each loop iteration. At the beginning of each iteration, the for loop executes only when i is not equal to 10. Otherwise, the loop is finished. In this case, the for loop executes 10 times. for loops can be implemented using a while loop, but the for loop is often convenient.

### Magnitude Comparison

So far, the examples have used beq and bne to perform equality or inequality comparisons and branches. MIPS provides the *set less than* instruction, slt, for magnitude comparison. slt sets rd to 1 when rs < rt. Otherwise, rd is 0.

---

#### Code Example 6.20 for LOOP

High-Level Code	MIPS Assembly Code
<pre>int sum = 0;  for (i = 0; i != 10; i = i + 1) {     sum = sum + i; }  // equivalent to the following while loop int sum = 0; int i = 0; while (i != 10) {     sum = sum + i;     i = i + 1; }</pre>	<pre># \$s0 = i, \$s1 = sum add \$s1, \$0, \$0      # sum = 0 addi \$s0, \$0, 0       # i = 0 addi \$t0, \$0, 10      # \$t0 = 10  for: beq \$s0, \$t0, done   # if i == 10, branch to done add \$s1, \$s1, \$s0      # sum = sum + i addi \$s0, \$s0, 1       # increment i j for  done:</pre>

---

**Example 6.6 LOOPS USING s1t**

The following high-level code adds the powers of 2 from 1 to 100. Translate it into assembly language.

```
// high-level code

int sum = 0;
for (i = 1; i < 101; i = i * 2)
    sum = sum + i;
```

**Solution:** The assembly language code uses the set less than (slt) instruction to perform the less than comparison in the for loop.

```
# MIPS assembly code

# $s0 = i, $s1 = sum
addi $s1, $0, 0      # sum = 0
addi $s0, $0, 1      # i = 1
addi $t0, $0, 101    # $t0 = 101

loop:
    slt $t1, $s0, $t0    # if (i < 101) $t1 = 1, else $t1 = 0
    beq $t1, $0, done    # if $t1 == 0 (i >= 101), branch to done
    add $s1, $s1, $s0    # sum = sum + i
    sll $s0, $s0, 1      # i = i * 2
    j loop
done:
```

---

Exercise 6.12 explores how to use slt for other magnitude comparisons including greater than, greater than or equal, and less than or equal.

### 6.4.5 Arrays

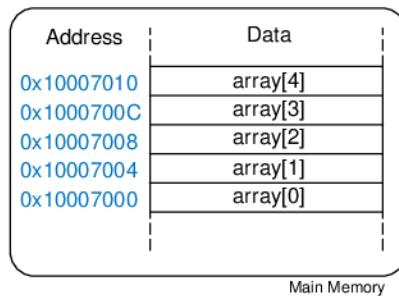
Arrays are useful for accessing large amounts of similar data. An array is organized as sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *size* of the array. This section shows how to access array elements in memory.

#### Array Indexing

Figure 6.20 shows an array of five integers stored in memory. The *index* ranges from 0 to 4. In this case, the array is stored in a processor's main memory starting at *base address* 0x10007000. The base address gives the address of the first array element, `array[0]`.

Code Example 6.21 multiplies the first two elements in `array` by 8 and stores them back in the array.

The first step in accessing an array element is to load the base address of the array into a register. Code Example 6.21 loads the base address



**Figure 6.20** Five-entry array with base address of 0x10007000

---

#### Code Example 6.21 ACCESSING ARRAYS

##### High-Level Code

```
int array [5];

array[0] = array[0] * 8;

array[1] = array[1] * 8;
```

##### MIPS Assembly Code

```
# $s0 = base address of array
lui    $s0, 0x1000      # $s0 = 0x10000000
ori    $s0, $s0, 0x7000  # $s0 = 0x10007000

lw     $t1, 0($s0)      # $t1 = array[0]
sll    $t1, $t1, 3       # $t1 = $t1 << 3 = $t1 * 8
sw     $t1, 0($s0)      # array[0] = $t1

lw     $t1, 4($s0)      # $t1 = array[1]
sll    $t1, $t1, 3       # $t1 = $t1 << 3 = $t1 * 8
sw     $t1, 4($s0)      # array[1] = $t1
```

into \$s0. Recall that the load upper immediate (`lui`) and or immediate (`ori`) instructions can be used to load a 32-bit constant into a register.

Code Example 6.21 also illustrates why `lw` takes a base address and an offset. The base address points to the start of the array. The offset can be used to access subsequent elements of the array. For example, `array[1]` is stored at memory address 0x10007004 (one word or four bytes after `array[0]`), so it is accessed at an offset of 4 past the base address.

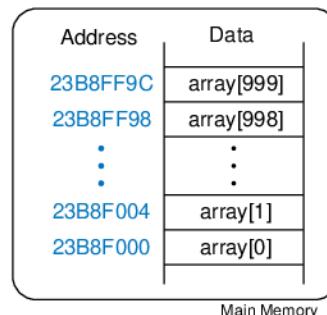
You might have noticed that the code for manipulating each of the two array elements in Code Example 6.21 is essentially the same except for the index. Duplicating the code is not a problem when accessing two array elements, but it would become terribly inefficient for accessing all of the elements in a large array. Code Example 6.22 uses a `for` loop to multiply by 8 all of the elements of a 1000-element array stored at a base address of 0x23B8F000.

Figure 6.21 shows the 1000-element array in memory. The index into the array is now a variable (`i`) rather than a constant, so we cannot take advantage of the immediate offset in `lw`. Instead, we compute the address of the  $i$ th element and store it in `$t0`. Remember that each array element is a word but that memory is byte addressed, so the offset from

**Code Example 6.22** ACCESSING ARRAYS USING A for LOOP

High-Level Code	MIPS Assembly Code
<pre> int i; int array[1000];  for (i=0; i &lt; 1000; i = i + 1) {     array[i] = array[i] * 8; } </pre>	<pre> # \$s0 = array base address, \$s1 = i # initialization code lui    \$s0, 0x23B8      # \$s0 = 0x23B80000 ori    \$s0, \$s0, 0xF000  # \$s0 = 0x23B8F000 addi   \$s1, \$0            # i = 0 addi   \$t2, \$0, 1000     # \$t2 = 1000  loop:     slt    \$t0, \$s1, \$t2      # i &lt; 1000?     beq    \$t0, \$0, done       # if not then done     sll    \$t0, \$s1, 2         # \$t0 = i * 4 (byte offset)     add    \$t0, \$t0, \$s0        # address of array[i]     lw     \$t1, 0(\$t0)        # \$t1 = array[i]     sll    \$t1, \$t1, 3         # \$t1 = array[i] * 8     sw     \$t1, 0(\$t0)        # array[i] = array[i] * 8     addi   \$s1, \$s1, 1          # i = i + 1     j     loop                  # repeat done: </pre>

**Figure 6.21** Memory holding array[1000] starting at base address 0x23B8F000



the base address is  $i * 4$ . Shifting left by 2 is a convenient way to multiply by 4 in MIPS assembly language. This example readily extends to an array of any size.

**Bytes and Characters**

Numbers in the range  $[-128, 127]$  can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type `char` to represent a byte or character.

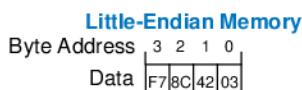
Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange (ASCII)*, which assigns each text character a unique byte value. Table 6.2 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lower-case and upper-case letters differ by 0x20 (32).

Other program languages, such as Java, use different character encodings, most notably *Unicode*. Unicode uses 16 bits to represent each character, so it supports accents, umlauts, and Asian languages. For more information, see [www.unicode.org](http://www.unicode.org).

MIPS provides load byte and store byte instructions to manipulate bytes or characters of data: load byte unsigned (`lbu`), load byte (`lb`), and store byte (`sb`). All three are illustrated in Figure 6.22.

**Table 6.2 ASCII encodings**

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	0	5F	_	6F	o		



Registers	
\$s1	00 00 00 8C lbu \$s1, 2(\$0)
\$s2	FFFF FF 8C lb \$s2, 2(\$0)
\$s3	XXXX XX 9B sb \$s3, 3(\$0)

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (-), to represent characters. For example, the letters A, B, C, and D were represented as .-, - . . . , -.-., and -.., respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D, were represented as 00011, 11001, 01110, and 01001. However, the 32 possible encodings of this 5-bit code were not sufficient for all the English characters. But 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

**Figure 6.22 Instructions for loading and storing bytes**

Load byte unsigned (`lbu`) zero-extends the byte, and load byte (`lb`) sign-extends the byte to fill the entire 32-bit register. Store byte (`sb`) stores the least significant byte of the 32-bit register into the specified byte address in memory. In Figure 6.22, `lbu` loads the byte at memory address 2 into the least significant byte of `$s1` and fills the remaining register bits with 0. `lb` loads the sign-extended byte at memory address 2 into `$s2`. `sb` stores the least significant byte of `$s3` into memory byte 3; it replaces `0xF7` with `0x9B`. The more significant bytes of `$s3` are ignored.

---

### **Example 6.7 USING `lb` AND `sb` TO ACCESS A CHARACTER ARRAY**

The following high-level code converts a ten-entry array of characters from lower-case to upper-case by subtracting 32 from each array entry. Translate it into MIPS assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that `$s0` already holds the base address of `chararray`.

```
// high-level code
char chararray[10];
int i;
for (i = 0; i != 10; i = i + 1)
    chararray[i] = chararray[i] - 32;
```

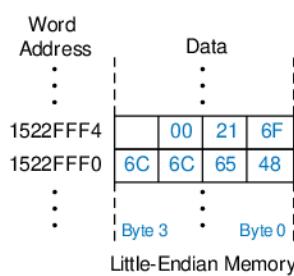
**Solution:**

```
# MIPS assembly code
# $s0 = base address of chararray, $s1 = i

        addi $s1, $0, 0      # i = 0
        addi $t0, $0, 10     # $t0 = 10
loop:   beq $t0, $s1, done # if i == 10, exit loop
        add $t1, $s1, $s0   # $t1 = address of chararray[i]
        lb  $t2, 0($t1)    # $t2 = array[i]
        addi $t2, $t2, -32  # convert to uppercase: $t1 = $t1 - 32
        sb  $t2, 0($t1)    # store new value in array:
                            # chararray[i] = $t1
        addi $s1, $s1, 1    # i = i + 1
        j   loop            # repeat

done:
```

---



**Figure 6.23** The string “Hello!” stored in memory

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, Figure 6.23 shows the string “Hello!” (0x48 6C 6C 65 48 00) stored in memory. The string is seven bytes long and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

#### 6.4.6 Procedure Calls

High-level languages often use *procedures* (also called *functions*) to reuse frequently accessed code and to make a program more readable. Procedures have inputs, called *arguments*, and an output, called the *return value*. Procedures should calculate the return value and cause no other unintended side effects.

When one procedure calls another, the calling procedure, the *caller*, and the called procedure, the *callee*, must agree on where to put the arguments and the return value. In MIPS, the caller conventionally places up to four arguments in registers \$a0-\$a3 before making the procedure call, and the callee places the return value in registers \$v0-\$v1 before finishing. By following this convention, both procedures know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the function of the caller. Briefly, this means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the *return address* in \$ra at the same time it jumps to the callee using the jump and link instruction (*jal*). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the saved registers, \$s0-\$s7, \$ra, and the *stack*, a portion of memory used for temporary variables, unmodified.

This section shows how to call and return from a procedure. It shows how procedures access input arguments and the return value and how they use the stack to store temporary variables.

##### Procedure Calls and Returns

MIPS uses the *jump and link* instruction (*jal*) to call a procedure and the *jump register* instruction (*jr*) to return from a procedure. Code Example 6.23 shows the *main* procedure calling the *simple* procedure. *main* is the caller, and *simple* is the callee. The *simple* procedure is called with no input arguments and generates no return value; it simply returns to the caller. In Code Example 6.23, instruction addresses are given to the left of each MIPS instruction in hexadecimal.

---

##### Code Example 6.23 simple PROCEDURE CALL

High-Level Code	MIPS Assembly Code
<pre>int main() {     simple();     ... } // void means the function returns no value void simple() {     return; }</pre>	<pre>0x00400200 main: jal simple # call procedure 0x00400204      ... 0x00401020 simple: jr \$ra      # return</pre>

Jump and link (`jal`) and jump register (`jr $ra`) are the two essential instructions needed for a procedure call. `jal` performs two functions: it stores the address of the *next* instruction (the instruction after `jal`) in the return address register (`$ra`), and it jumps to the target instruction.

In Code Example 6.23, the `main` procedure calls the `simple` procedure by executing the jump and link (`jal`) instruction. `jal` jumps to the `simple` label and stores `0x00400204` in `$ra`. The `simple` procedure returns immediately by executing the instruction `jr $ra`, jumping to the instruction address held in `$ra`. The `main` procedure then continues executing at this address, `0x00400204`.

#### **Input Arguments and Return Values**

The `simple` procedure in Code Example 6.23 is not very useful, because it receives no input from the calling procedure (`main`) and returns no output. By MIPS convention, procedures use `$a0-$a3` for input arguments and `$v0-$v1` for the return value. In Code Example 6.24, the procedure `diffofsums` is called with four arguments and returns one result.

According to MIPS convention, the calling procedure, `main`, places the procedure arguments, from left to right, into the input registers, `$a0-$a3`. The called procedure, `diffofsums`, stores the return value in the return register, `$v0`.

A procedure that returns a 64-bit value, such as a double-precision floating point number, uses both return registers, `$v0` and `$v1`. When a procedure with more than four arguments is called, the additional input arguments are placed on the stack, which we discuss next.

Code Example 6.24 has some subtle errors. Code Examples 6.25 and 6.26 on page 323 show improved versions of the program.

---

#### **Code Example 6.24 PROCEDURE CALL WITH ARGUMENTS AND RETURN VALUES**

##### **High-Level Code**

```
int main ()
{
    int y;
    ...
    y = diffofsums (2, 3, 4, 5);
    ...
}

int diffofsums (int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;
}
```

##### **MIPS Assembly Code**

```
# $s0 = y
main:
    ...
    addi $a0, $0, 2    # argument 0 = 2
    addi $a1, $0, 3    # argument 1 = 3
    addi $a2, $0, 4    # argument 2 = 4
    addi $a3, $0, 5    # argument 3 = 5
    jal diffofsums    # call procedure
    add $s0, $v0, $0    # y = returned value
    ...

# $s0 = result
diffofsums:
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0    # put return value in $v0
    jr $ra              # return to caller
```

### The Stack

The *stack* is memory that is used to save local variables within a procedure. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. Before explaining how procedures use the stack to store temporary variables, we explain how the stack works.

The stack is a *last-in-first-out (LIFO) queue*. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be pulled (*popped*) off. Each procedure may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack*, is the most recently allocated space. Whereas a stack of dishes grows up in space, the MIPS stack grows *down* in memory. The stack expands to lower memory addresses when a program needs more scratch space.

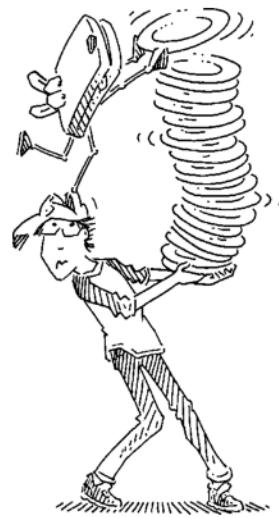
Figure 6.24 shows a picture of the stack. The *stack pointer*, \$sp, is a special MIPS register that points to the top of the stack. A *pointer* is a fancy name for a memory address. It points to (gives the address of) data. For example, in Figure 6.24(a) the stack pointer, \$sp, holds the address value 0x7FFFFFFC and points to the data value 0x12345678. \$sp points to the top of the stack, the lowest accessible memory address on the stack. Thus, in Figure 6.24(a), the stack cannot access memory below memory word 0x7FFFFFFC.

The stack pointer (\$sp) starts at a high memory address and decrements to expand as needed. Figure 6.24(b) shows the stack expanding to allow two more data words of temporary storage. To do so, \$sp decrements by 8 to become 0x7FFFFFF4. Two additional data words, 0xAABBCCDD and 0x11223344, are temporarily stored on the stack.

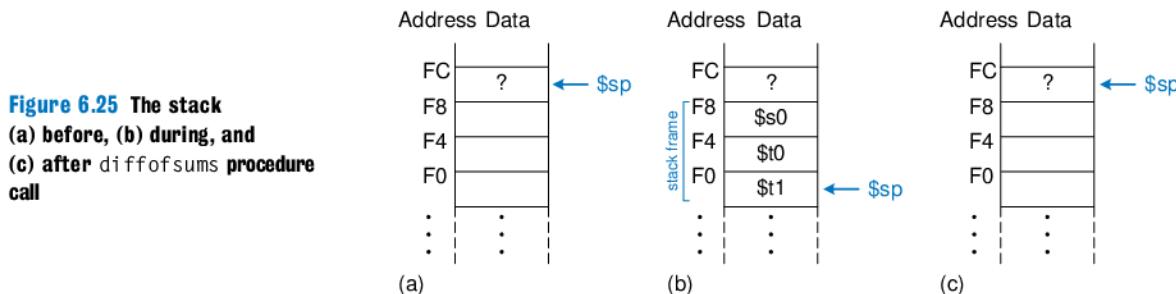
One of the important uses of the stack is to save and restore registers that are used by a procedure. Recall that a procedure should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides the one containing the return value, \$v0. The `diffofsums` procedure in Code Example 6.24 violates this rule because it modifies \$t0, \$t1, and \$s0. If `main` had been using \$t0, \$t1, or \$s0 before the call to `diffofsums`, the contents of these registers would have been corrupted by the procedure call.

To solve this problem, a procedure saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps.

1. Makes space on the stack to store the values of one or more registers.
2. Stores the values of the registers on the stack.
3. Executes the procedure using the registers.
4. Restores the original values of the registers from the stack.
5. Dealлокates space on the stack.



**Figure 6.24 The stack**



Code Example 6.25 shows an improved version of `diffofsums` that saves and restores `$t0`, `$t1`, and `$s0`. The new lines are indicated in blue. Figure 6.25 shows the stack before, during, and after a call to the `diffofsums` procedure from Code Example 6.25. `diffofsums` makes room for three words on the stack by decrementing the stack pointer (`$sp`) by 12. It then stores the current values of `$s0`, `$t0`, and `$t1` in the newly allocated space. It executes the rest of the procedure, changing the values in these three registers. At the end of the procedure, `diffofsums` restores the values of `$s0`, `$t0`, and `$t1` from the stack, deallocates its stack space, and returns. When the procedure returns, `$v0` holds the result, but there are no other side effects: `$s0`, `$t0`, `$t1`, and `$sp` have the same values as they did before the procedure call.

The stack space that a procedure allocates for itself is called its *stack frame*. `diffofsums`'s stack frame is three words deep. The principle of modularity tells us that each procedure should access only its own stack frame, not the frames belonging to other procedures.

#### Preserved Registers

Code Example 6.25 assumes that temporary registers `$t0` and `$t1` must be saved and restored. If the calling procedure does not use those registers, the effort to save and restore them is wasted. To avoid this waste, MIPS divides registers into *preserved* and *nonpreserved* categories. The preserved registers include `$s0-$s7` (hence their name, *saved*). The nonpreserved registers include `$t0-$t9` (hence their name, *temporary*). A procedure must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.26 shows a further improved version of `diffofsums` that saves only `$s0` on the stack. `$t0` and `$t1` are nonpreserved registers, so they need not be saved.

Remember that when one procedure calls another, the former is the *caller* and the latter is the *callee*. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a

**Code Example 6.25 PROCEDURE SAVING REGISTERS ON THE STACK****MIPS Assembly Code**

```
# $s0 = result
diffofsums:
    addi $sp, $sp, -12 # make space on stack to store three registers
    sw $s0, 8($sp) # save $s0 on stack
    sw $t0, 4($sp) # save $t0 on stack
    sw $t1, 0($sp) # save $t1 on stack
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0 # put return value in $v0
    lw $t1, 0($sp) # restore $t1 from stack
    lw $t0, 4($sp) # restore $t0 from stack
    lw $s0, 8($sp) # restore $s0 from stack
    addi $sp, $sp, 12 # deallocate stack space
    jr $ra # return to caller
```

**Code Example 6.26 PROCEDURE SAVING PRESERVED REGISTERS ON THE STACK****MIPS Assembly Code**

```
# $s0 = result
diffofsums:
    addi $sp, $sp, -4 # make space on stack to store one register
    sw $s0, 0($sp) # save $s0 on stack
    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f + g) - (h + i)
    add $v0, $s0, $0 # put return value in $v0
    lw $s0, 0($sp) # restore $s0 from stack
    addi $sp, $sp, 4 # deallocate stack space
    jr $ra # return to caller
```

nonpreserved register, the caller needs to save that nonpreserved register before making the procedure call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Table 6.3 summarizes which registers are preserved. \$s0-\$s7 are generally used to hold local variables within a procedure, so they must be saved. \$ra must also be saved, so that the procedure knows where to return. \$t0-\$t9 are used to hold temporary results before they are assigned to local variables. These calculations typically complete before a procedure call is made, so they are not preserved, and it is rare that the caller needs to save them. \$a0-\$a3 are often overwritten in the process of calling a procedure. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called procedure returns. \$v0-\$v1 certainly should not be preserved, because the callee returns its result in these registers.

**Table 6.3 Preserved and nonpreserved registers**

Preserved	Nonpreserved
Saved registers: \$s0-\$s7	Temporary registers: \$t0-\$t9
Return address: \$ra	Argument registers: \$a0-\$a3
Stack pointer: \$sp	Return value registers: \$v0-\$v1
Stack above the stack pointer	Stack below the stack pointer

The stack above the stack pointer is automatically preserved as long as the callee does not write to memory addresses above \$sp. In this way, it does not modify the stack frame of any other procedures. The stack pointer itself is preserved, because the callee deallocates its stack frame before returning by adding back the same amount that it subtracted from \$sp at the beginning of the procedure.

#### Recursive Procedure Calls

A procedure that does not call others is called a *leaf* procedure; an example is `diffsums`. A procedure that does call others is called a *nonleaf* procedure. As mentioned earlier, nonleaf procedures are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another procedure, and then restore those registers afterward. Specifically, the caller saves any nonpreserved registers (\$t0-\$t9 and \$a0-\$a3) that are needed after the call. The callee saves any of the preserved registers (\$s0-\$s7 and \$ra) that it intends to modify.

A *recursive* procedure is a nonleaf procedure that calls itself. The factorial function can be written as a recursive procedure call. Recall that  $\text{factorial}(n) = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$ . The factorial function can be rewritten recursively as  $\text{factorial}(n) = n \times \text{factorial}(n - 1)$ . The factorial of 1 is simply 1. Code Example 6.27 shows the factorial function written as a recursive procedure. To conveniently refer to program addresses, we assume that the program starts at address 0x90.

The factorial procedure might modify \$a0 and \$ra, so it saves them on the stack. It then checks whether  $n < 2$ . If so, it puts the return value of 1 in \$v0, restores the stack pointer, and returns to the caller. It does not have to reload \$ra and \$a0 in this case, because they were never modified. If  $n > 1$ , the procedure recursively calls `factorial(n-1)`. It then restores the value of  $n$  (\$a0) and the return address (\$ra) from the stack, performs the multiplication, and returns this result. The multiply instruction (`mul $v0, $a0, $v0`) multiplies \$a0 and \$v0 and places the result in \$v0. It is discussed further in Section 6.7.1.

**Code Example 6.27** factorial RECURSIVE PROCEDURE CALL**High-Level Code**

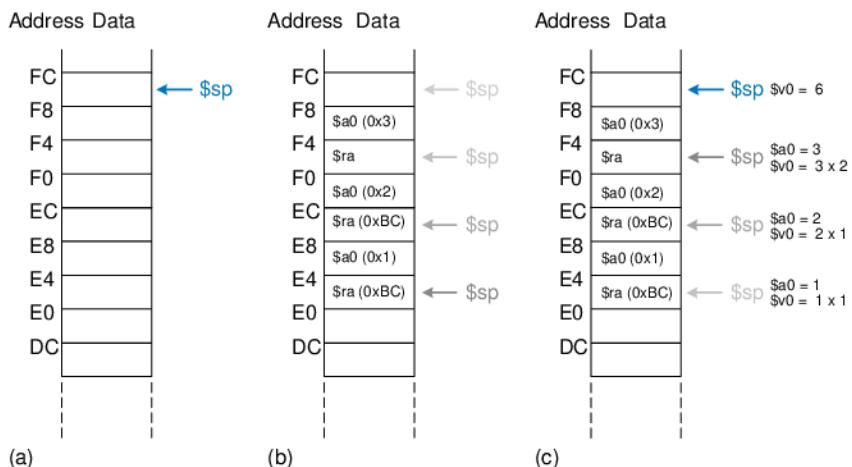
```
int factorial ( int n ) {
    if ( n <= 1 )
        return 1;

    else
        return ( n * factorial ( n-1 ) );
}
```

**MIPS Assembly Code**

```
0x90 factorial: addi $sp, $sp, -8      # make room on stack
0x94           sw   $a0, 4($sp)          # store $a0
0x98           sw   $ra, 0($sp)          # store $ra
0x9C           addi $t0, $0, 2          # $t0 = 2
0xA0           slt  $t0, $a0, $t0       # a <= 1 ?
0xA4           beq  $t0, $0, .else      # no: goto else
0xA8           addi $v0, $0, 1          # yes: return 1
0xAC           addi $sp, $sp, 8          # restore $sp
0xB0           jr   $ra              # return
0xB4 .else:    addi $a0, $a0, -1       # n = n - 1
0xB8           jal   factorial        # recursive call
0xBC           lw    $ra, 0($sp)        # restore $ra
0xC0           lw    $a0, 4($sp)        # restore $a0
0xC4           addi $sp, $sp, 8          # restore $sp
0xC8           mul   $v0, $a0, $v0       # n * factorial (n-1)
0xCC           jr   $ra              # return
```

Figure 6.26 shows the stack when executing `factorial(3)`. We assume that `$sp` initially points to `0xFC`, as shown in Figure 6.26(a). The procedure creates a two-word stack frame to hold `$a0` and `$ra`. On the first invocation, `factorial` saves `$a0` (holding  $n = 3$ ) at `0xF8` and `$ra` at `0xF4`, as shown in Figure 6.26(b). The procedure then changes `$a0` to  $n = 2$  and recursively calls `factorial(2)`, making `$ra` hold `0xBC`. On the second invocation, it saves `$a0` (holding  $n = 2$ ) at `0xF0` and `$ra` at `0xEC`. This time, we know that `$ra` contains `0xBC`. The procedure then changes `$a0` to  $n = 1$  and recursively calls `factorial(1)`. On the third invocation, it saves `$a0` (holding  $n = 1$ ) at `0xE8` and `$ra` at `0xE4`. This time, `$ra` again contains `0xBC`. The third invocation of



**Figure 6.26** Stack during factorial procedure call when  $n = 3$ : (a) before call, (b) after last recursive call, (c) after return

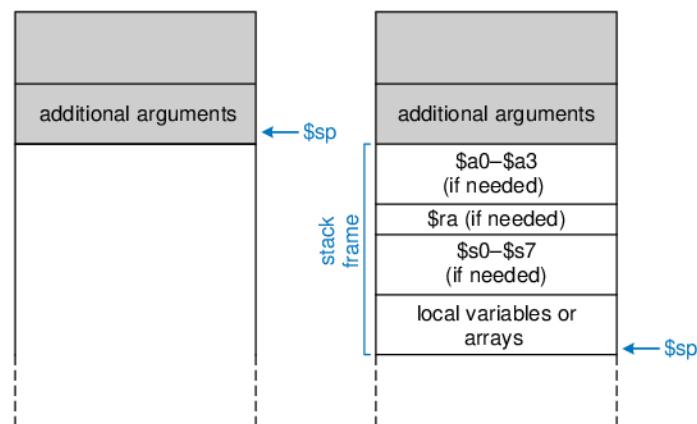
factorial returns the value 1 in \$v0 and deallocates the stack frame before returning to the second invocation. The second invocation restores n to 2, restores \$ra to 0xBC (it happened to already have this value), deallocates the stack frame, and returns \$v0 = 2 × 1 = 2 to the first invocation. The first invocation restores n to 3, restores \$ra to the return address of the caller, deallocates the stack frame, and returns \$v0 = 3 × 2 = 6. Figure 6.26(c) shows the stack as the recursively called procedures return. When factorial returns to the caller, the stack pointer is in its original position (0xFC), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. \$v0 holds the return value, 6.

#### **Additional Arguments and Local Variables\***

Procedures may have more than four input arguments and local variables. The stack is used to store these temporary values. By MIPS convention, if a procedure has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above \$sp. The *caller* must expand its stack to make room for the additional arguments. Figure 6.27(a) shows the caller's stack for calling a procedure with more than four arguments.

A procedure can also declare local variables or arrays. *Local* variables are declared within a procedure and can be accessed only within that procedure. Local variables are stored in \$s0–\$s7; if there are too many local variables, they can also be stored in the procedure's stack frame. In particular, local arrays are stored on the stack.

Figure 6.27(b) shows the organization of a callee's stack frame. The frame holds the procedure's own arguments (if it calls other procedures), the return address, and any of the saved registers that the procedure will



**Figure 6.27** Stack usage:  
(left) before call,  
(right) after call

modify. It also holds local arrays and any excess local variables. If the callee has more than four arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a procedure can access stack data not in its own stack frame.

## 6.5 ADDRESSING MODES

MIPS uses five *addressing modes*: register-only, immediate, base, PC-relative, and pseudo-direct. The first three modes (register-only, immediate, and base addressing) define modes of reading and writing operands. The last two (PC-relative and pseudo-direct addressing) define modes of writing the program counter, PC.

### Register-Only Addressing

*Register-only addressing* uses registers for all source and destination operands. All R-type instructions use register-only addressing.

### Immediate Addressing

*Immediate addressing* uses the 16-bit immediate along with registers as operands. Some I-type instructions, such as add immediate (`addi`) and load upper immediate (`lui`), use immediate addressing.

### Base Addressing

Memory access instructions, such as load word (`lw`) and store word (`sw`), use *base addressing*. The effective address of the memory operand is found by adding the base address in register `rs` to the sign-extended 16-bit offset found in the immediate field.

### PC-relative Addressing

Conditional branch instructions use *PC-relative addressing* to specify the new value of the PC if the branch is taken. The signed offset in the immediate field is added to the PC to obtain the new PC; hence, the branch destination address is said to be *relative* to the current PC.

Code Example 6.28 shows part of the factorial procedure from Code Example 6.27. Figure 6.28 shows the machine code for the `beq` instruction. The *branch target address* (BTA) is the address of the next instruction to execute if the branch is taken. The `beq` instruction in Figure 6.28 has a BTA of 0xB4, the instruction address of the `else` label.

The 16-bit immediate field gives the number of instructions between the BTA and the instruction *after* the branch instruction (the instruction at  $PC+4$ ). In this case, the value in the immediate field of `beq` is 3 because the BTA (0xB4) is 3 instructions past  $PC+4$  (0xA8).

The processor calculates the BTA from the instruction by sign-extending the 16-bit immediate, multiplying it by 4 (to convert words to bytes), and adding it to  $PC+4$ .