CHAPTER 1

An Introduction to Computer Science

1.1 Introduction

This text is an invitation to learn about one of the youngest and most exciting of the scientific disciplines—**computer science**. Almost every day our newspapers, magazines, and televisions carry reports of advances in computing, such as high-speed supercomputers that perform one quadrillion (10^{15}) mathematical operations per second; networks that transmit high-definition images and movies anywhere in the world in fractions of a second; and minute computers that can be embedded into our books, watches, clothing, and even our bodies. The next few years will see technological breakthroughs that, until a few years ago, existed only in the minds of dreamers and science fiction writers. These are exciting times in computing, and our goal in this text is to provide you with an understanding of computer science and an appreciation for the diverse areas of research and study within this important field.

While the average person can produce a reasonably accurate description of most scientific fields, even if he or she did not study the subject in school, many people do not have an intuitive understanding of the types of problems studied by computer science professionals. For example, you probably know that biology is the study of living organisms and that chemistry deals with the structure and composition of matter. However, you might not have the same understanding of the work that goes on in computer science. In fact, many people harbor one or more of the following common misconceptions about this field.

MISCONCEPTION 1: *Computer science is the study of computers*.

This apparently obvious definition is actually incorrect or, to put it more precisely, incomplete. For example, some of the earliest and most fundamental theoretical work in computer science took place from 1920 to 1940, years before the development of the first computer system. (This pioneering work was initially considered a branch of logic and applied mathematics. Computer science did not come to be recognized as a separate and independent field of scientific study until the late 1950s to early 1960s.) Even today, there are branches of computer science quite distinct from the study of "real" machines. In *theoretical computer science*, for example, researchers study the logical and mathematical properties of problems and their solutions. Frequently, these researchers investigate problems not with actual computers but rather with *formal models* of computation, which are easier to study and analyze mathematically. Their work involves pencil and paper, not circuit boards and disks.

This distinction between computers and computer science is beautifully expressed by computer scientists Michael R. Fellows and Ian Parberry in an article in the journal *Computing Research News*:

> Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes, or chemistry is about beakers and test tubes. Science is not about tools. It is about how we use them and what we find out when we do.[1]

MISCONCEPTION 2: *Computer science is the study of how to write computer programs*.

Many people are first introduced to computer science when learning to write programs in a language such as C++, Python, or Java. This almost universal use of programming as the entry to the discipline can create the misunderstanding that computer science is equivalent to computer programming.

Programming is extremely important to the discipline—researchers use it to study new ideas and build and test new solutions—but like the computer itself it is a tool. When computer scientists design and analyze a new approach to solving a problem, or create new ways to represent information, they implement their ideas as programs in order to test them on an actual computer system. This enables researchers to see how well these new ideas work and whether they perform better than previous methods.

For example, searching a list is one of the most common applications of computers, and it is frequently applied to huge problems, such as finding one name among the approximately 20,000,000 listings in the New York City telephone directory. (We will solve this problem in Chapter 2.) A more efficient lookup method could significantly reduce the time that customers must wait for directory assistance. Assume that we have designed what we believe to be a "new and improved" search technique. After analyzing it theoretically, we would study it empirically by writing a program to implement our new method, executing it on our computer, and measuring its performance. These tests would demonstrate under what conditions our new method is or is not faster than the directory search procedures currently in use.

In computer science, it is not simply the construction of a high-quality program that is important but also the methods it embodies, the services it provides, and the results it produces. It is possible to become so enmeshed in writing code and getting it to run that we forget that a program is only a means to an end, not an end in itself.

MISCONCEPTION 3: *Computer science is the study of the uses and applications of computers and software*.

If one's introduction to computer science is not programming, then it may be a course on the application of computers and software. Such a course typically teaches the use of a number of popular packages, such as word processors, presentation software, database systems, imaging software, electronic mail, and a Web browser.

---

1 Fellows, M. R., and Parberry, I. "Getting Children Excited About Computer Science," *Computing Research News*, vol. 5, no. 1 (January 1993).

These packages are widely used by professionals in all fields. However, learning to use a software package is no more a part of computer science than driver's education is a branch of automotive engineering. A wide range of people *use* computer software, but the computer scientist is responsible for *specifying*, *designing*, *building*, and *testing* software packages as well as the computer systems on which they run.

These three misconceptions about computer science are not entirely wrong; they are just woefully incomplete. Computers, programming languages, software, and applications *are* part of the discipline of computer science, but neither individually nor combined do they capture the richness and diversity of this field.

We have spent a good deal of time saying what computer science is *not*. What, then, is it? What are its basic concepts? What are the fundamental questions studied by professionals in this field? Is it possible to capture the breadth and scope of the discipline in a single definition? We answer these fundamental questions in the next section and, indeed, in the remainder of the text.

In the Beginning . . .

There is no single date that marks the beginning of computer science. Indeed, there are many "firsts" that could be used to mark this event. For example, some of the earliest theoretical work on the logical foundations of computer science occurred in the 1930s. The first general-purpose, electronic computers appeared during the period 1940–1946. (We will discuss the history of these early machines in Section 1.4.) These first computers were one-of-a-kind experimental systems that never moved outside the research laboratory. The first commercial machine, the UNIVAC I, did not make its appearance until March 1951, a date that marks the real beginning of the computer industry. The first high-level (i.e., based on natural language) programming language was FORTRAN. Some people mark its debut in 1957 as the beginning of the "software" industry. The appearance of these new machines and languages created new occupations, such as programmer, numerical analyst, and computer engineer. To address the intellectual needs of these workers, the first professional society for people in the field of computing, the Association for Computing Machinery (ACM), was established in 1947. (The ACM is still the largest professional computer science society in the world. Its Web page is located at *www.acm.org*.) To help meet the rapidly growing need for computer professionals, the first Department of Computer Science was established at Purdue University in October 1962. It awarded its first M.Sc. degree in 1964 and its first Ph.D. in computer science in 1966. An undergraduate program was established in 1968.

Thus, depending on what you consider the most important "first," the field of computer science is somewhere between 40 and 70 years old. Compared to such classic scientific disciplines as mathematics, physics, chemistry, and biology, computer science is the new kid on the block.

1.2 The Definition of Computer Science

There are many definitions of computer science, but the one that best captures the richness and breadth of ideas embodied in this branch of science was first proposed by professors Norman Gibbs and Allen Tucker.[2] According to their definition, the central concept in computer science is the **algorithm**. It is not possible to understand the field without a thorough understanding of this critically important idea.

2 Gibbs, N. E., and Tucker, A. B. "A Model Curriculum for a Liberal Arts Degree in Computer Science," *Comm. of the ACM*, vol. 29, no. 3 (March 1986).

DEFINITION

Computer science the study of algorithms, including

1. Their formal and mathematical properties
2. Their hardware realizations
3. Their linguistic realizations
4. Their applications

The Gibbs and Tucker definition says that it is the task of the computer scientist to design and develop algorithms to solve a range of important problems. This design process includes the following operations:

- Studying the behavior of algorithms to determine if they are correct and efficient (their formal and mathematical properties)
- Designing and building computer systems that are able to execute algorithms (their hardware realizations)
- Designing programming languages and translating algorithms into these languages so that they can be executed by the hardware (their linguistic realizations)
- Identifying important problems and designing correct and efficient software packages to solve these problems (their applications)

Because it is impossible to appreciate this definition fully without knowing what an algorithm is, let's look more closely at this term.

The dictionary defines the word *algorithm* as follows:

> ***al • go • rithm*** n. *A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step method for accomplishing some task.*

Informally, an algorithm is an ordered sequence of instructions that is guaranteed to solve a specific problem. It is a list that looks something like this:

STEP 1: Do something
STEP 2: Do something
STEP 3: Do something
. .
. .
. .
STEP *N*: Stop, you are finished

If you are handed this list and carefully follow its instructions in the order specified, when you reach the end you will have solved the task at hand.

All the operations used to construct algorithms belong to one of only three categories:

1. *Sequential operations* A sequential instruction carries out a single well-defined task. When that task is finished, the algorithm moves on

to the next operation. Sequential operations are usually expressed as simple declarative sentences.

- Add 1 cup of butter to the mixture in the bowl.
- Subtract the amount of the check from the current account balance.
- Set the value of x to 1.

2. *Conditional operations* These are the "question-asking" instructions of an algorithm. They ask a question, and the next operation is selected on the basis of the answer to that question.
   - If the mixture is too dry, then add one-half cup of water to the bowl.
   - If the amount of the check is less than or equal to the current account balance, then cash the check; otherwise, tell the person that the account is overdrawn.
   - If x is not equal to 0, then set y equal to $1/x$; otherwise, print an error message that says you cannot divide by 0.
3. *Iterative operations* These are the "looping" instructions of an algorithm. They tell us not to go on to the next instruction but, instead, to go back and repeat the execution of a previous block of instructions.
   - Repeat the previous two operations until the mixture has thickened.
   - While there are still more checks to be processed, do the following five steps.
   - Repeat steps 1, 2, and 3 until the value of y is equal to +1.

We use algorithms (although we don't call them that) all the time—whenever we follow a set of instructions to assemble a child's toy, bake a cake, balance a checkbook, or go through the college registration process. A good example of an algorithm used in everyday life is the set of instructions shown in Figure 1.1 for programming a DVR to record a sequence of television shows. Note the three types of instructions in this algorithm: sequential (steps 2, 4, 5, 6, and 8), conditional (steps 1 and 7), and iterative (step 3).

Mathematicians use algorithms all the time, and much of the work done by early Greek, Roman, Persian, and Indian mathematicians involved the discovery of algorithms for important problems in geometry and arithmetic; an example is *Euclid's algorithm* for finding the greatest common divisor of two positive integers. (Exercise 7 at the end of the chapter presents this 2,300-year-old algorithm.) We also studied algorithms in elementary school, even if we didn't know it. For example, in the first grade we learned an algorithm for adding two numbers such as

$$\begin{array}{r} 47 \\ +\ 25 \\ \hline 72 \end{array}$$

The instructions our teacher gave were as follows: First add the rightmost column of numbers (7 + 5), getting the value 12. Write down the 2 under the line and carry the 1 to the next column. Now move left to the next column, adding (4 + 2) and the previous carry value of 1 to get 7. Write this value under the line, producing the correct answer 72.

FIGURE 1.1

Programming Your DVR. An Example of an Algorithm

Algorithm for Programming Your DVR

Step 1 If the clock and calendar are not correctly set, then go to page 9 of the instruction manual and follow the instructions there before proceeding to step 2.
Step 2 Place a blank tape into the DVR disc slot.
Step 3 Repeat steps 4 through 7 for each program that you wish to record.
Step 4 Enter the channel number that you wish to record and press the button labeled CHAN.
Step 5 Enter the time that you wish recording to start and press the button labeled TIME-START.
Step 6 Enter the time that you wish recording to stop and press the button labeled TIME-FINISH. This completes the programming of one show.
Step 7 If you do not wish to record anything else, press the button labeled END-PROG.
Step 8 Turn off your DVR. Your DVR is now in TIMER mode, ready to record.

Although as children we learned this algorithm informally, it can, like the DVR instructions in Figure 1.1, be written formally as an explicit sequence of instructions. Figure 1.2 shows an algorithm for adding two positive m-digit numbers. It expresses formally the operations informally described previously. Again, note the three types of instructions used to construct the algorithm: sequential (steps 1, 2, 4, 6, 7, 8, and 9), conditional (step 5), and iterative (step 3).

Even though it may not appear so, this is the same "decimal addition algorithm" that you learned in grade school; if you follow it rigorously, it is guaranteed to produce the correct result. Let's watch it work.

FIGURE 1.2

Algorithm for Adding Two m-digit Numbers

Algorithm for Adding Two *m*-Digit Numbers

Given: $m \geq 1$ and two positive numbers each containing m digits, $a_{m-1}\ a_{m-2,} \ldots a_0$ and $b_{m-1}\ b_{m-2,} \ldots b_0$
Wanted: $c_m c_{m-1}\ c_{m-2} \ldots c_0$, where $c_m c_{m-1}\ c_{m-2} \ldots c_0 = (a_{m-1}\ a_{m-2} \ldots a_0) + (b_{m-1}\ b_{m-2} \ldots b_0)$

Algorithm:
Step 1 Set the value of *carry* to 0.
Step 2 Set the value of i to 0.
Step 3 While the value of i is less than or equal to $m - 1$, repeat the instructions in steps 4 through 6.
Step 4 Add the two digits a_i and b_i to the current value of *carry* to get c_i.
Step 5 If $c_i \geq 10$, then reset c_i to $(c_i - 10)$ and reset the value of *carry* to 1; otherwise, set the new value of *carry* to 0.
Step 6 Add 1 to i, effectively moving one column to the left.
Step 7 Set c_m to the value of *carry*.
Step 8 Print out the final answer, $c_m\ c_{m-1}\ c_{m-2} \ldots c_0$.
Step 9 Stop.

Add $(47 + 25)$
$m = 2$
$a_1 = 4$ $a_0 = 7$
$b_1 = 2$ $b_0 = 5$
} The input

STEP 1: *carry* = 0.

STEP 2: $i = 0$.

STEP 3: We now repeat steps 4 through 6 while i is less than or equal to 1.

First repetition of the loop (i has the value 0)

STEP 4: Add $(a_0 + b_0 + carry)$, which is 7 + 5 + 0, so $c_0 = 12$.

STEP 5: Because $c_0 \geq 10$, we reset $\boldsymbol{c_0}$ **to 2** and reset *carry* to 1.

STEP 6: Reset i to (0 + 1) = 1. Since i is less than or equal to 1, go back to step 4.

Second repetition of the loop (i has the value 1)

STEP 4: Add $(a_1 + b_1 + carry)$, which is 4 + 2 + 1, so $\boldsymbol{c_1 = 7}$.

STEP 5: Because $c_1 < 10$, we reset *carry* to 0.

STEP 6: Reset i to (1 + 1) = 2. Because i is greater than 1, do not repeat the loop but instead go to step 7.

STEP 7: Set $\boldsymbol{c_2 = 0}$.

STEP 8: Print out the answer $c_2\ c_1\ c_0 = 072$ (see the **boldface** values).

STEP 9: Stop.

We have reached the end of the algorithm, and it has correctly produced the sum of the two numbers 47 and 25, the three-digit result 072. (A more clever algorithm would omit the unnecessary leading zero at the beginning of the number if the last carry value is a zero. That modification is an exercise at the end of the chapter.) Try working through the algorithm shown in Figure 1.2 with another pair of numbers to be sure that you understand exactly how it functions.

Abu Ja' far Muhammad ibn Musa Al-Khowarizmi (a.d. 780–850?)

The word *algorithm* is derived from the last name of Muhammad ibn Musa Al-Khowarizmi, a famous Persian mathematician and author from the eighth and ninth centuries. Al-Khowarizmi was a teacher at the Mathematical Institute in Baghdad and the author of the book *Kitab al jabr w'al muqabala,* which in English means "Rules of Restoration and Reduction." It is one of the earliest mathematical textbooks, and its title gives us the word *algebra* (the Arabic word *al jabr* means "reduction").

In 825 A.D., Al-Khowarizmi wrote another book about the base-10 positional numbering system that had recently been developed in India. In this book he described formalized, step-by-step procedures for doing arithmetic operations, such as addition, subtraction, and multiplication, on numbers represented in this new decimal system. In the twelfth century this book was translated into Latin, introducing the base-10 Hindu-Arabic numbering system to Europe, and Al-Khowarizmi's name became closely associated with these formal numerical techniques. His last name was rendered as Algorismus in Latin characters, and eventually the formalized procedures that he pioneered and developed became known as *algorithms* in his honor.

The addition algorithm shown in Figure 1.2 is a highly formalized representation of a technique that most people learned in the first or second grade and that virtually everyone knows how to do informally. Why would we take such a simple task as adding two numbers and express it in so complicated a fashion? Why are formal algorithms so important in computer science? Because:

> *If we can specify an algorithm to solve a problem, then we can automate its solution.*

Once we have formally specified an algorithm, we can build a machine (or write a program or hire a person) to carry out the steps contained in the algorithm. The machine (or program or person) does not need to understand the concepts or ideas underlying the solution. It merely has to do step 1, step 2, step 3, . . . exactly as written. In computer science terminology, the machine, robot, person, or thing carrying out the steps of the algorithm is called a **computing agent**.

Thus computer science can also be viewed as the science of algorithmic problem solving. Much of the research and development work in computer science involves discovering correct and efficient algorithms for a wide range of interesting problems, studying their properties, designing programming languages into which those algorithms can be encoded, and designing and building computer systems that can automatically execute these algorithms in an efficient manner.

At first glance, it may seem that every problem can be solved algorithmically. However, you will learn in Chapter 12 the startling fact (first proved by the German logician Kurt Gödel in the early 1930s) that there are problems for which no generalized algorithmic solution can possibly exist. These problems are, in a sense, *unsolvable*. No matter how much time and effort is put into obtaining a solution, none will ever be found. Gödel's discovery, which staggered the mathematical world, effectively places a limit on the capabilities of computers and computer scientists.

There are also problems for which it is possible to specify an algorithm but a computing agent would take so long to execute it that the solution is essentially useless. For example, to get a computer to play winning chess, we could use a *brute force* approach. Given a board position as input, the computer would examine every legal move it could possibly make, then every legal response an opponent could make to each initial move, then every response it could select to that move, and so on. This analysis would continue until the game reached a win, lose, or draw position. With that information the computer would be able to optimally choose its next move. If, for simplicity's sake, we assume that there are 40 legal moves from any given position on a chessboard, and it takes about 30 moves to reach a final conclusion, then the total number of board positions that our brute force program would need to evaluate in deciding its first move is

$$\underbrace{40 \times 40 \times 40 \times \ldots \times 40}_{30 \text{ times}} = 40^{30}, \text{ which is roughly } 10^{48}$$

If we could build a supercomputer that evaluates 1 trillion (10^{12}) board positions per second (which is too high at current levels of technology), it would take about 30,000,000,000,000,000,000,000,000,000 years for the computer to make its first move! Obviously, a computer could not use a brute force technique to play a real chess game.

There also exist problems that we do not yet know *how* to solve algorithmically. Many of these involve tasks that require a degree of what we term "intelligence." For example, after only a few days a baby recognizes the face of its mother from among the many faces it sees. In a few months it begins to develop coordinated sensory and motor control skills and can efficiently plan how to use them—how to get from the playpen to the toy on the floor without bumping into either the chair or the desk that are in the way. After a few years the child begins to develop powerful language skills and abstract reasoning capabilities.

We take these abilities for granted, but the operations just mentioned—sophisticated visual discrimination, high-level problem solving, abstract reasoning, natural language understanding—cannot be done well (or even at all) using the computer systems and software packages currently available. The primary reason is that researchers do not yet know how to specify these operations algorithmically. That is, they do not yet know how to specify a solution formally in a detailed step-by-step fashion. As humans, we are able to do them simply by using the "algorithms" in our heads. To appreciate this problem, imagine trying to describe algorithmically exactly what steps you follow when you are painting a picture, composing a poem, or formulating a business plan.

Thus, algorithmic problem solving has many variations. Sometimes solutions do not exist; sometimes a solution is too inefficient to be of any use; sometimes a solution is not yet known. However, discovering an algorithmic solution has enormously important consequences. As we noted earlier, if we can create a correct and efficient algorithm to solve a problem, and if we encode it into a programming language, then we can take advantage of the speed and power of a computer system to automate the solution and produce the desired result. This is what computer science is all about.

1.3 Algorithms

1.3.1 *The Formal Definition of an Algorithm*

DEFINITION

Algorithm a well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time.

The formal definition of an algorithm is rather imposing and contains a number of important ideas. Let's take it apart, piece by piece, and analyze each of its separate points.

. . . a well-ordered collection . . .

An algorithm is a collection of operations, and there must be a clear and unambiguous *ordering* to these operations. Ordering means that we know which operation to do first and precisely which operation follows each completed operation. After all, we cannot expect a computing agent to carry out our instructions correctly if it is confused about which instruction it should be carrying out.

Consider the following "algorithm" that was taken from the back of a shampoo bottle and is intended to be instructions on how to use the product.

STEP 1: Wet hair
STEP 2: Lather
STEP 3: Rinse
STEP 4: Repeat

At step 4, what operations should be repeated? If we go back to step 1, we will be unnecessarily wetting our hair. (It is presumably still wet from the previous operations.) If we go back to step 3 instead, we will not be getting our hair any cleaner because we have not reused the shampoo. The Repeat instruction in step 4 is ambiguous in that it does not clearly specify what to do next. Therefore, it violates the well-ordered requirement of an algorithm. (It also has a second and even more serious problem—it never stops! We will have more to say about this second problem shortly.) Statements such as

- Go back and do it again. (Do *what* again?)
- Start over. (From *where*?)
- If you understand this material, you may skip ahead. (How *far*?)
- Do either part 1 or part 2. (How do I decide *which* one to do?)

are ambiguous and can leave us confused and unsure about what operation to do next. We must be extremely precise in specifying the order in which operations are to be carried out. One possible way is to number the steps of the algorithm and use these numbers to specify the proper order of execution. For example, the ambiguous operations shown above could be made more precise as follows:

- Go back to step 3 and continue execution from that point.
- Start over from step 1.
- If you understand this material, skip ahead to line 21.
- If you are 18 years of age or older, do part 1 beginning with step 9; otherwise, do part 2 beginning with step 40.

. . . *of unambiguous and effectively computable operations* . . .

Algorithms are composed of things called "operations," but what do those operations look like? What types of building blocks can be used to construct an algorithm? The answer to these questions is that the operations used in an algorithm must meet two criteria—they must be *unambiguous,* and they must be *effectively computable*.

Here is a possible "algorithm" for making a cherry pie:

STEP 1: Make the crust
STEP 2: Make the cherry filling
STEP 3: Pour the filling into the crust
STEP 4: Bake at 350°F for 45 minutes

For a professional baker, this algorithm would be fine. He or she would understand how to carry out each of the operations listed above. Novice cooks, like most of us, would probably understand the meaning of steps 3 and 4. However, we would probably look at steps 1 and 2, throw up our hands in confusion, and ask for clarification. We might then be given more detailed instructions.

STEP 1: Make the crust
- 1.1 Take one and one-third cups flour
- 1.2 Sift the flour
- 1.3 Mix the sifted flour with one-half cup butter and one-fourth cup water
- 1.4 Roll into two 9-inch pie crusts

STEP 2: Make the cherry filling
- 2.1 Open a 16-ounce can of cherry pie filling and pour into bowl
- 2.2 Add a dash of cinnamon and nutmeg, and stir

With this additional information most people, even inexperienced cooks, would understand what to do, and they could successfully carry out this baking algorithm. However, there may be some people, perhaps young children, who still do not fully understand each and every line. For those people, we must go through the simplification process again and describe the ambiguous steps in even more elementary terms.

For example, the computing agent executing the algorithm might not know the meaning of the instruction "Sift the flour" in step 1.2, and we would have to explain it further.

1.2 Sift the flour
- 1.2.1 Get out the sifter, which is the device shown on page A-9 of your cookbook, and place it directly on top of a 2-quart bowl
- 1.2.2 Pour the flour into the top of the sifter and turn the crank in a counterclockwise direction
- 1.2.3 Let all the flour fall through the sifter into the bowl

Now, even a child should be able to carry out these operations. But if that were not the case, then we would go through the simplification process yet one more time, until every operation, every sentence, every word was clearly understood.

An **unambiguous** operation is one that can be understood and carried out directly by the computing agent without further simplification or explanation.

When an operation is unambiguous, we call it a **primitive operation**, or simply a **primitive** of the computing agent carrying out the algorithm. An algorithm must be composed entirely of primitives. Naturally, the primitive operations of different individuals (or machines) vary depending on their sophistication, experience, and intelligence, as is the case with the cherry pie recipe, which varies with the baking experience of the person following the instructions. Hence, an algorithm for one computing agent may not be an algorithm for another.

One of the most important questions we will answer in this text is, What are the primitive operations of a typical modern computer system? What operations can a hardware processor "understand" in the sense of being able to carry out directly, and what operations must be further refined and simplified?

However, it is not enough for an operation to be understandable. It must also be *doable* by the computing agent. If an algorithm tells me to flap my arms really quickly and fly, I understand perfectly well what it is asking me to do. However, I am incapable of doing it. "Doable" means there exists a computational process that allows the computing agent to complete that operation successfully. The formal term for "doable" is **effectively computable**.

For example, the following is an incorrect technique for finding and printing the 100th prime number. (A prime number is a whole number not evenly divisible by any numbers other than 1 and itself, such as 2, 3, 5, 7, 11, 13, ...)

STEP 1: Generate a list L of all the prime numbers: L_1, L_2, L_3, . . .
STEP 2: Sort the list L in ascending order
STEP 3: Print out the 100th element in the list, L_{100}
STEP 4: Stop

The problem with these instructions is in step 1, "Generate a list L of *all* the prime numbers. . . ." That operation cannot be completed. There are an infinite number of prime numbers, and it is not possible in a finite amount of time to generate the desired list L. No such computational process exists, and the operation described in step 1 is not effectively computable. Here are some other examples of operations that may not be effectively computable:

Write out the exact decimal value of π. (π cannot be represented exactly.)
Set *average* to (*sum* ÷ *number*). (If *number* = 0, division is undefined.)
Set the value of *result* to $\sqrt{N}$. (If $N < 0$, then *result* is undefined if you are using real numbers.)
Add 1 to the current value of x. (What if x currently has no value?)

This last example explains why we had to initialize the value of the variable called *carry* to 0 in step 1 of Figure 1.2. In step 4 the algorithm says, "Add the two digits a_i and b_i to the current value of *carry* to get c_i." If *carry* has no current value, then when the computing agent tries to perform the instruction in step 4, it will not know what to do, and this operation is not effectively computable.

. . . that produces a result . . .

Algorithms solve problems. In order to know whether a solution is correct, an algorithm must produce a result that is observable to a user, such as a numerical answer, a new object, or a change to its environment. Without some

observable result, we would not be able to say whether the algorithm is right or wrong. In the case of the DVR algorithm (Figure 1.1), the result will be a disc containing recorded TV programs. The addition algorithm (Figure 1.2) produces an *m*-digit sum.

Note that we use the word *result* rather than *answer*. Sometimes it is not possible for an algorithm to produce the correct answer because for a given set of input, a correct answer does not exist. In those cases the algorithm may produce something else, such as an error message, a red warning light, or an approximation to the correct answer. Error messages, lights, and approximations, though not necessarily what we wanted, are all observable results.

. . . and halts in a finite amount of time.

Another important characteristic of algorithms is that the result must be produced after the execution of a finite number of operations, and we must guarantee that the algorithm eventually reaches a statement that says, "Stop, you are done" or something equivalent. We have already pointed out that the shampooing algorithm was not well ordered because we did not know which statements to repeat in step 4. However, even if we knew which block of statements to repeat, the algorithm would still be incorrect because it makes no provision to terminate. It will essentially run forever, or until we run out of hot water, soap, or patience. This is called an **infinite loop**, and it is a common error in the design of algorithms.

Figure 1.3(a) shows an algorithmic solution to the shampooing problem that meets all the criteria discussed in this section if we assume that you want to wash your hair twice. The algorithm of Figure 1.3(a) is well ordered. Each step is numbered, and the execution of the algorithm unfolds sequentially, beginning at step 1 and proceeding from instruction i to instruction $i + 1$, unless the operation specifies otherwise. (For example, the iterative instruction in step 3 says that after completing step 6, you should go back and start again at step 4 until the value of *WashCount* equals 2.) The intent of each operation is (we assume) clear, unambiguous, and doable by the person washing his or her hair. Finally, the algorithm will halt. This is confirmed by observing that *WashCount* is initially set to 0 in step 2. Step 6 says to add 1 to *WashCount* each time we lather and rinse our hair, so it will take on the values 0, 1, 2, ... However, the iterative statement in step 3 says stop lathering and rinsing when the value of *WashCount* reaches 2. At that point, the algorithm goes to step 7 and terminates execution with the desired result: clean

FIGURE 1.3(a)
A Correct Solution to the Shampooing Problem

Algorithm for Shampooing Your Hair

Step	Operation
1	Wet your hair
2	Set the value of *WashCount* to 0
3	Repeat steps 4 through 6 until the value of *WashCount* equals 2
4	Lather your hair
5	Rinse your hair
6	Add 1 to the value of *WashCount*
7	Stop, you have finished shampooing your hair

FIGURE 1.3(b)

Another Correct Solution to the Shampooing Problem

Another Algorithm for Shampooing Your Hair

Step	Operation
1	Wet your hair
2	Lather your hair
3	Rinse your hair
4	Lather your hair
5	Rinse your hair
6	Stop, you have finished shampooing your hair

hair. (Although it is correct, do not expect to see this algorithm on the back of a shampoo bottle in the near future.)

As is true for any recipe or set of instructions, there is always more than a single way to write a correct solution. For example, the algorithm of Figure 1.3(a) could also be written as shown in Figure 1.3(b). Both of these are correct solutions to the shampooing problem. (Although they are both correct, they are not necessarily equally elegant. This point is addressed in Exercise 6 at the end of the chapter.)

1.3.2 *The Importance of Algorithmic Problem Solving*

The instruction sequences in Figures 1.1, 1.2, 1.3(a), and 1.3(b) are examples of the types of algorithmic solutions designed, analyzed, implemented, and tested by computer scientists, although they are much shorter and simpler. The operations shown in these figures could be encoded into some appropriate language and given to a computing agent (such as a personal computer or a robot) to execute. The device would mechanically follow these instructions and successfully complete the task. This device could do this without having to understand the creative processes that went into the discovery of the solution and without knowing the principles and concepts that underlie the problem. The robot simply follows the steps in the specified order (a required characteristic of algorithms), successfully completing each operation (another required characteristic), and ultimately producing the desired result after a finite amount of time (also required).

Just as the Industrial Revolution of the nineteenth century allowed machines to take over the drudgery of repetitive physical tasks, the "computer revolution" of the twentieth and twenty-first centuries has enabled us to implement algorithms that mechanize and automate the drudgery of repetitive mental tasks, such as adding long columns of numbers, finding names in a telephone book, sorting student records by course number, and retrieving hotel or airline reservations from a file containing hundreds of thousands of pieces of data. This mechanization process offers the prospect of enormous increases in productivity. It also frees people to do those things that humans do much better than computers, such as creating new ideas, setting policy, doing high-level planning, and determining the significance of the results produced by a computer. Certainly, these operations are a much more effective use of that unique computing agent called the human brain.

PRACTICE PROBLEMS

Get a copy of the instructions that describe how to

1. register for classes at the beginning of the semester.
2. use the online computer catalog to see what is available in the college library on a given subject.
3. use the copying machine in your building.
4. log on to the World Wide Web.

Look over the instructions and decide whether they meet the definition of an algorithm given in this section. If not, explain why, and rewrite each set of instructions so that it constitutes a valid algorithm. Also state whether each instruction is a sequential, conditional, or iterative operation.

1.4 A Brief History of Computing

Although computer science is not simply a study of computers, there is no doubt that the field was formed and grew in popularity as a direct response to their creation and widespread use. This section takes a brief look at the historical development of computer systems.

The appearance of some technologies, such as the telephone, the light bulb, and the first heavier-than-air flight, can be traced directly to a single place, a specific individual, and an exact instant in time. Examples include the flight of Orville and Wilbur Wright on December 17, 1903, in Kitty Hawk, North Carolina; and the famous phrase "Mr. Watson, come here, I want to see you." uttered by Alexander Graham Bell over the first telephone on March 12, 1876.

Computers are not like that. They did not appear in a specific room on a given day as the creation of some individual genius. The ideas that led to the design of the first computers evolved over hundreds of years, with contributions coming from many people, each building on and extending the work of earlier discoverers.

1.4.1 *The Early Period: Up to 1940*

If this were a discussion of the history of mathematics and arithmetic instead of computer science, it would begin 3,000 years ago with the early work of the Greeks, Egyptians, Babylonians, Indians, Chinese, and Persians. All these cultures were interested in and made important contributions to the fields of mathematics, logic, and numerical computation. For example, the Greeks developed the fields of geometry and logic; the Babylonians and Egyptians developed numerical methods for generating square roots, multiplication tables, and trigonometric tables used by early sailors; Indian mathematicians developed both the base-10 decimal numbering system and the concept of zero; and in the ninth century the Persians developed algorithmic problem solving.

The first half of the seventeenth century saw a number of important developments related to automating and simplifying the drudgery of

arithmetic computation. (The motivation for this work appears to be the sudden increase in scientific research during the sixteenth and seventeenth centuries in the areas of astronomy, chemistry, and medicine. This work required the solution of larger and more complex mathematical problems.) In 1614, the Scotsman John Napier invented **logarithms** as a way to simplify difficult mathematical computations. The early seventeenth century also witnessed the development of a number of new and quite powerful mechanical devices designed to help reduce the burden of arithmetic. The first **slide rule** appeared around 1622. In 1672, the French philosopher and mathematician Blaise Pascal designed and built one of the first **mechanical calculators** (named the **Pascaline**) that could do addition and subtraction. A model of this early calculating device is shown in Figure 1.4.

The famous German mathematician Gottfried Leibnitz (who, along with Isaac Newton, was one of the inventors of the calculus) was also excited by the idea of automatic computation. He studied the work of Pascal and others, and in 1674, he constructed a mechanical calculator called **Leibnitz's Wheel** that could do not only addition and subtraction but multiplication and division as well. Both Pascal's and Leibnitz's machines used interlocking mechanical cogs and gears to store numbers and perform basic arithmetic operations. Considering the state of technology available to Pascal, Leibnitz, and others in the seventeenth century, these first calculating machines were truly mechanical wonders.

These early developments in mathematics and arithmetic were important milestones because they demonstrated how mechanization could simplify and speed up numerical computation. For example, Leibnitz's Wheel enabled seventeenth-century mathematicians to generate tables of mathematical functions many times faster than was possible by hand. (It is hard to believe in our modern high-tech society, but in the seventeenth century the generation of a table of logarithms could represent a *lifetime's* effort of one person!) However, the slide rule and mechanical calculators of Pascal and Leibnitz, though certainly impressive devices, were not computers. Specifically, they lacked two fundamental characteristics:

- They did not have a *memory* where information could be stored in machine-readable form.
- They were not *programmable*. A person could not provide *in advance* a sequence of instructions that could be executed by the device without manual intervention.

FIGURE 1.4
The Pascaline. One of the Earliest Mechanical Calculators

Computer History Museum

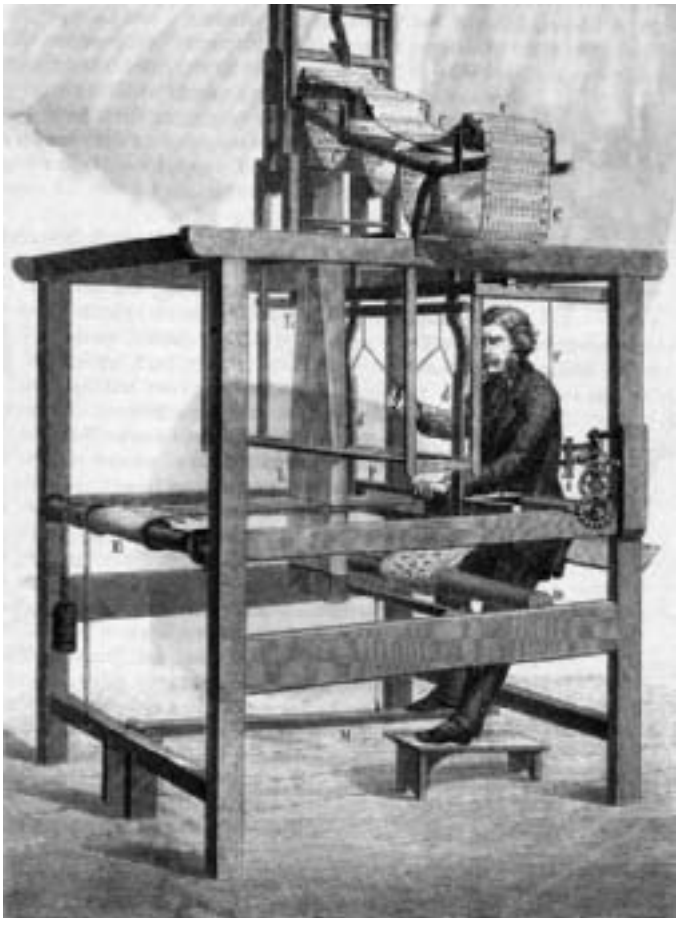
Surprisingly, the first actual "computing device" to include both of these features was not created for the purposes of mathematical computations. Rather, it was a loom used for the manufacture of rugs and clothing. It was developed in 1801 by the Frenchman Joseph Jacquard. Jacquard wanted to automate the weaving process, at the time a painfully slow and cumbersome task in which each separate row of the pattern had to be set up by the weaver and an apprentice. Because of this, anything but the most basic style of clothing was beyond the means of most people.

Jacquard designed an automated loom that used **punched cards** to create the desired pattern. If there was a hole in the card in a particular location, then a hook could pass through the card, grasp a warp thread, and raise it to allow a second thread to pass underneath. If there was no hole in the card, then the hook could not pass through, and the thread would pass over the warp. Depending on whether the thread passed above or below the warp, a specific design was created. Each punched card described one row of the pattern. Jacquard connected the cards and fed them through his loom, and it automatically sequenced from card to card, weaving the desired pattern. A drawing of the **Jacquard loom** is shown in Figure 1.5. The rows of connected punched cards can be seen at the top of the device.

Jacquard's loom represented an enormously important stage in the development of computers. Not only was it the first programmable device, but it also showed how the knowledge of a human expert (in this case, a master weaver) could be captured in machine-readable form and used to control a

FIGURE 1.5

Drawing of the Jacquard Loom



machine that accomplished the same task automatically. Once the program was created, the expert was no longer needed. The lowliest apprentice could load the cards into the loom, turn it on, and produce a finished, high-quality product over and over again.

The Original "Technophobia"

The development of the automated Jacquard loom and other technological advances in the weaving industry was so frightening to the craft guilds of the early nineteenth century that in 1811 it led to the formation of a group called the **Luddites**. The Luddites, named after their leader Ned Ludd of Nottingham, England, were violently opposed to this new manufacturing technology, and they burned down factories that attempted to use it. The movement lasted only a few years and its leaders were all jailed, but their name lives on today as a pejorative term for any group that is frightened and angered by the latest developments in any branch of science and technology, including computers.

These pioneers had enormous influence on the designers and inventors who came after them, among them a mathematics professor at Cambridge University named Charles Babbage. Babbage was interested in automatic computation. In 1823, he extended the ideas of Pascal and Leibnitz and constructed a working model of the largest and most sophisticated mechanical calculator of its time. This machine, called the **Difference Engine**, could do addition, subtraction, multiplication, and division to 6 significant digits, and it could solve polynomial equations and other complex mathematical problems as well. Babbage tried to construct a larger model of the Difference Engine that would be capable of working to an accuracy of 20 significant digits, but after 12 years of work he had to give up his quest. The technology available in the 1820s and 1830s was not sufficiently advanced to manufacture cogs and gears to the precise tolerances his design required. Like Galileo's helicopter or Jules Verne's atomic submarine, Babbage's ideas were fundamentally sound but years ahead of their time. (In 1991 the London Museum of Science, using Babbage's original plans, built an actual working model of the Difference Engine. It was 7 feet high, 11 feet wide, weighed 3 tons, and had 4,000 moving parts. It worked exactly as Babbage had planned.)

Babbage did not stop his investigations with the Difference Engine. In the 1830s, he designed a more powerful and general-purpose computational machine that could be configured to solve a much wider range of numerical problems. His machine had four basic components: a **mill** to perform the arithmetic manipulation of data, a **store** to hold the data, an **operator** to process the instructions contained on punched cards, and an **output unit** to put the results onto separate punched cards. Although it would be about 110 years before a "real" computer would be built, Babbage's proposed machine, called the **Analytic Engine**, is amazingly similar in design to a modern computer. The four components of the Analytic Engine are virtually identical in function to the four major components of today's computer systems:

Babbage's Term	*Modern Terminology*
mill	arithmetic/logic unit
store	memory
operator	processor
output	input/output

Babbage died before a working steam-powered model of his Analytic Engine could be completed, but his ideas lived on to influence others, and many computer scientists consider the Analytic Engine the first "true" computer system, even if it existed only on paper and in Babbage's dreams.

Another person influenced by the work of Pascal, Jacquard, and Babbage was a young statistician at the U.S. Census Bureau named Herman Hollerith. Because of the rapid increase in immigration to America at the end of the nineteenth century, officials estimated that doing the 1890 enumeration manually would take from 10 to 12 years. The 1900 census would begin before the previous one was finished. Something had to be done.

Hollerith designed and built programmable card-processing machines that could automatically read, tally, and sort data entered on punched cards. Census data were coded onto cards using a machine called a **keypunch**. The cards were taken either to a **tabulator** for counting and tallying or to a **sorter** for ordering alphabetically or numerically. Both of these machines were programmable (via wires and plugs) so that the user could specify such things as which card columns should be tallied and in what order the cards should be sorted. In addition, the machines had a small amount of memory to store results. Thus, they had all four components of Babbage's Analytic Engine.

Hollerith's machines were enormously successful, and they were one of the first examples of the use of automated information processing to solve large-scale "real-world" problems. Whereas the 1880 census required 8 years to be completed, the 1890 census was finished in about 2 years, even though there was a 30% increase in the U.S. population during that decade.

Although they were not really general-purpose computers, Hollerith's card machines were a very clear and very successful demonstration of the enormous advantages of automated information processing. This fact was not lost on

Charles Babbage (1791–1871)
Ada Augusta Byron, Countess of Lovelace (1815–1852)

Charles Babbage, the son of a banker, was born into a life of wealth and comfort in eighteenth-century England. He attended Cambridge University and displayed an aptitude for mathematics and science. He was also an inventor and "tinkerer" who loved to build all sorts of devices. Among the devices he constructed were unpickable locks, skeleton keys, speedometers, and even the first cow catcher for trains. His first and greatest love, though, was mathematics, and he spent the better part of his life creating machines to do automatic computation. Babbage was enormously impressed by the work of Jacquard in France. (In fact, Babbage had on the wall of his home a woven portrait of Jacquard that was woven using 24,000 punched cards.) He spent the last 30 to 40 years of his life trying to build a computing device, the Analytic Engine, based on Jacquard's ideas.

In that quest, he was helped by Countess Ada Augusta Byron, daughter of the famous English poet, Lord Byron. The countess was introduced to Babbage and was enormously impressed by his ideas about the Analytic Engine. As she put it, "We may say most aptly that the Analytic Engine weaves algebraic patterns just as the Jacquard Loom weaves flowers and leaves." Lady Lovelace worked closely with Babbage to specify how to organize instructions for the Analytic Engine to solve a particular mathematical problem. Because of that pioneering work, she is generally regarded as history's first computer programmer.

Babbage died in 1871 without realizing his dream. He also died quite poor because the Analytic Engine ate up virtually all of his personal fortune. His work was generally forgotten until the twentieth century when it became instrumental in moving the world into the computer age.

Hollerith, who left the Census Bureau in 1902 to found the Computer Tabulating Recording Company to build and sell these machines. He planned to market his new product to a country that was just entering the Industrial Revolution and that, like the Census Bureau, would be generating and processing enormous volumes of inventory, production, accounting, and sales data. His punched card machines became the dominant form of data processing equipment during the first half of the twentieth century, well into the 1950s and 1960s. During this period, virtually every major U.S. corporation had data processing rooms filled with keypunches, sorters, and tabulators, as well as drawer upon drawer of punched cards. In 1924, Hollerith's tabulating machine company changed its name to IBM, and it eventually evolved into the largest computing company in the world.

We have come a long way from the 1640s and the Pascaline, the early adding machine constructed by Pascal. We have seen the development of more powerful mechanical calculators (Leibnitz), automated programmable manufacturing devices (Jacquard), a design for the first computing device (Babbage), and the initial applications of information processing on a massive scale (Hollerith). However, we still have not yet entered the "computer age." That did not happen until about 1940, and it was motivated by an event that, unfortunately, has fueled many of the important technological advances in human history—the outbreak of war.

1.4.2 *The Birth of Computers: 1940–1950*

World War II created another, quite different set of information-based problems. Instead of inventory, sales, and payroll, the concerns became ballistics tables, troop deployment data, and secret codes. A number of research projects were started, funded largely by the military, to build automatic computing machines to perform these tasks and assist the Allies in the war effort.

Beginning in 1931, the U.S. Navy and IBM jointly funded a project at Harvard University under Professor Howard Aiken to build a computing device called Mark I. This was a general-purpose, electromechanical programmable computer that used a mix of relays, magnets, and gears to process and store data. The Mark I was the first computing device to use the base-2 binary numbering system, which we will discuss in Chapter 4. It used vacuum tubes and electric current to represent the two binary values, off for 0, on for 1. Until then computing machines had used decimal representation, typically using a 10-toothed gear, each tooth representing a digit from 0 to 9. The Mark I was completed in 1944, about 110 years after Babbage's dream of the Analytic Engine, and is generally considered one of the first working general-purpose computers. The Mark I had a memory capacity of 72 numbers, and it could be programmed to perform a 23-digit multiplication in the lightning-like time of 4 seconds. Although laughably slow by modern standards, the Mark I was operational for almost 15 years, and it carried out a good deal of important and useful mathematical work for the U.S. Navy during the war.

At about the same time, a much more powerful machine was taking shape at the University of Pennsylvania in conjunction with the U.S. Army. During the early days of World War II, the Army was producing many new artillery pieces, but it found that it could not produce the firing tables equally as fast. These tables told the gunner how to aim the gun on the basis of such input as distance to the target and current temperature, wind, and elevation. Because

of the enormous number of variables and the complexity of the computations (which use both trigonometry and calculus), these firing tables were taking more time to construct than the gun itself.

To help solve this problem, in 1943 the Army initiated a research project with J. Presper Eckert and John Mauchly of the University of Pennsylvania to build a completely electronic computing device. The machine, dubbed the ENIAC (Electronic Numerical Integrator and Calculator), was completed in 1946 and was the first fully electronic general-purpose programmable computer. This pioneering machine is shown in Figure 1.6.

ENIAC contained 18,000 vacuum tubes and nearly filled a building; it was 100 feet long, 10 feet high, and weighed 30 tons. Because it was fully electronic, it did not contain any of the slow mechanical components found in Mark I, and it executed instructions much more rapidly. The ENIAC could add two 10-digit numbers in about 1/5,000 of a second and could multiply two numbers in 1/300 of a second, a thousand times faster than the Mark I.

The Mark I and ENIAC are two well-known examples of early computers, but they are by no means the only ones of that era. For example, the ABC system (Atanasoff-Berry Computer), designed and built by Professor John Atanasoff and his graduate student Clifford Berry at Iowa State University, was actually the first electronic computer, constructed during the period 1939–1942. However, it never received equal recognition because it was useful for only one task, solving systems of simultaneous linear equations. In England, a computer called Colossus was built in 1943 under the direction of Alan Turing, a famous mathematician and computer scientist whom we will meet again in Chapter 12. This machine, one of the first computers built outside the United States, was used to crack the famous German Enigma code that the Nazis believed to be unbreakable. Colossus has also not received as much recognition as ENIAC because of the secrecy that shrouded the Enigma project. Its very existence was not widely known until many years after the end of the war.

At about the same time that Colossus was taking form in England, a German engineer named Konrad Zuse was working on a computing device for the German

FIGURE 1.6

Photograph of the ENIAC Computer

From the Collections of the University of Pennsylvania Archives

army. The machine, code named Z1, was similar in design to the ENIAC—a programmable, general-purpose, fully electronic computing device. Fortunately for the allied forces, the Z1 project was not completed before the end of World War II.

Although the machines just described—ABC, Mark I, ENIAC, Colossus, and Z1—were computers in the fullest sense of the word (they had memory and were programmable), they did not yet look like modern computer systems. One more step was necessary, and that step was taken in 1946 by the individual who was most instrumental in creating the computer as we know it today, John Von Neumann.

Von Neumann was not only one of the most brilliant mathematicians who ever lived, but was a genius in many other areas as well, including experimental physics, chemistry, economics, and computer science. Von Neumann, who taught at Princeton University, had worked with Eckert and Mauchly on the ENIAC project at the University of Pennsylvania. Even though that project was successful, he recognized a number of fundamental shortcomings in ENIAC. In 1946, he proposed a radically different computer design based on a model called the **stored program computer**. Until then, all computers were programmed *externally* using wires, connectors, and plugboards. The memory unit stored only data, not instructions. For each different problem, users had to rewire virtually the entire computer. For example, the plugboards on the ENIAC contained 6,000 separate switches, and reprogramming the ENIAC involved specifying the new settings for all these switches—not a trivial task.

Von Neumann proposed that the instructions that control the operation of the computer be encoded as binary values and stored internally in the

John Von Neumann (1903–1957)

John Von Neumann was born in Budapest, Hungary. He was a child prodigy who could divide 8-digit numbers in his head by the age of 6. He was a genius in virtually every field that he studied, including physics, economics, engineering, and mathematics. At 18 he received an award as the best mathematician in Hungary, a country known for excellence in the field, and he received his Ph.D., summa cum laude, at 21. He came to the United States in 1930 as a guest lecturer at Princeton University and taught there for three years. Then, in 1933 he became one of the founding members (along with Albert Einstein) of the Institute for Advanced Studies, where he worked for 20 years.

He was one of the most brilliant minds of the twentieth century, a true genius in every sense, both good and bad. He could do prodigious mental feats in his head, and his thought processes usually raced way ahead of "ordinary" mortals who found him quite difficult to work with. One of his colleagues joked that "Johnny wasn't really human, but after living among them for so long, he learned to do a remarkably good imitation of one."

Von Neumann was a brilliant theoretician who did pioneering work in pure mathematics, operations research, game theory, and theoretical physics. He was also an engineer, concerned about practicalities and real-world problems, and it was this interest in applied issues that led Von Neumann to design and construct the first stored program computer. One of the early computers built by the RAND Corp. in 1953 was affectionately called "Johnniac" in his honor, although Von Neumann detested that name. Like the UNIVAC I, it has a place of honor at the Smithsonian Institution.

memory unit along with the data. To solve a new problem, instead of rewiring the machine, you would rewrite the sequence of instructions—that is, create a new program. Von Neumann invented programming as it is known today.

The model of computing proposed by Von Neumann included many other important features found on all modern computing systems, and to honor him this model of computation has come to be known as the **Von Neumann architecture**. We will study this architecture in great detail in Chapters 4 and 5.

Von Neumann's research group at the University of Pennsylvania implemented his ideas, and they built one of the first stored program computers, called EDVAC (with a V), in 1951. At about the same time, a stored program computer called EDSAC (with an S) was built at Cambridge University in England under the direction of Professor Maurice Wilkes. The appearance of these machines and others like them ushered in the modern computer age. Even though they were much slower, bulkier, and less powerful than our current machines, EDVAC and EDSAC executed programs in a fashion surprisingly similar to the miniaturized and immensely more powerful computers of the twenty-first century. A commercial model of the EDVAC, called UNIVAC I—the first computer actually sold—was built by Eckert and Mauchly and delivered to the U.S. Bureau of the Census on March 31, 1951. (It ran for 12 years before it was retired, shut off for the last time, and moved to the Smithsonian Institution.) This date marks the true beginning of the "computer age."

The importance of Von Neumann's contributions to computer systems development cannot be overstated. Although his original proposals are at least 60 years old, virtually every computer built today is a Von Neumann machine in its basic design. A lot has changed in computing, and a powerful high-resolution graphics workstation and the EDVAC would appear to have little in common. However, the basic principles on which these machines are

And the Verdict Is . . .

Our discussion of what was happening in computing from 1939 to 1946 showed that many groups were involved in designing and building the first computers. Therefore, it would seem that no single individual can be credited with the title Inventor of the Electronic Digital Computer.

Surprisingly, that is not true. In February 1964, the Sperry Rand Corp. (now UNISYS) was granted a U.S. patent on the ENIAC computer as the first fully electronic computing device, J. Presper Eckert and John Mauchly being its designers and builders. However, in 1967 a suit was filed in U.S. District Court in Minneapolis, Minnesota, to overturn that patent. The suit, *Honeywell v. Sperry Rand*, was heard before U.S. Federal Judge Earl Larson, and on October 19, 1973, his verdict was handed down. (This enormously important verdict was never given the media coverage it deserved because it happened in the middle of the Watergate hearings and on the very day that Vice President Spiro Agnew resigned in disgrace for tax fraud.) Judge Larson overturned the ENIAC patent on the basis that Eckert and Mauchly had been significantly influenced in their 1943–1944 work on ENIAC by earlier research and development work by John Atanasoff at Iowa State University. During the period 1939–1943, Mauchly had communicated extensively with Atanasoff and had even traveled to Iowa to see the ABC machine in person. In a sense, the verdict declared that Atanasoff is really the inventor of the first computer. This decision was never appealed. Therefore, the official honor of having designed and built the first electronic computer, at least in U.S. District Court, goes to Professor John Vincent Atanasoff.

On November 13, 1990, in a formal ceremony at the White House, Professor Atanasoff was awarded the National Medal of Technology by President George H.W. Bush for his pioneering contributions to the development of the computer.

constructed are virtually identical, and the same theoretical model underlies their operation. There is an old saying in computer science that "There is nothing new since Von Neumann!" This saying is certainly not true (much *has* happened), but it demonstrates the importance and amazing staying power of Von Neumann's original design.

1.4.3 *The Modern Era: 1950 to the Present*

The last 60 or so years of computer development have involved taking the Von Neumann architecture and improving it in terms of hardware and software. Since 1950, computer systems development has been primarily an *evolutionary* process, not a revolutionary one. The enormous number of changes in computers in recent decades have made them faster, smaller, cheaper, more reliable, and easier to use, but have not drastically altered their basic underlying structure.

The period 1950–1957 (these dates are rough approximations) is often called the **first generation** of computing. This era saw the appearance of UNIVAC I, the first computer built for sale, and the IBM 701, the first computer built by the company that would soon become a leader in this new field. These early systems were similar in design to EDVAC, and they were bulky, expensive, slow, and unreliable. They used vacuum tubes for processing and storage, and they were extremely difficult to maintain. The act of turning the machine on alone could blow out a dozen tubes! For this reason, first-generation machines were used only by trained personnel and only in specialized locations such as large corporations, government and university research labs, and military installations, which could provide this expensive support environment.

The **second generation** of computing, roughly 1957–1965, heralded a major change in the size and complexity of computers. In the late 1950s, the bulky vacuum tube was replaced by a single transistor only a few millimeters in size, and memory was now constructed using tiny magnetic cores only 1/50 of an inch in diameter. (We will introduce and describe both of these devices in Chapter 4.) These technologies not only dramatically reduced the size of computers but also increased their reliability and reduced costs. Suddenly, buying and using a computer became a real possibility for some small and medium-sized businesses, colleges, and government agencies. This was also the era of the appearance of FORTRAN and COBOL, the first **high-level** (English-like) **programming languages**. (We will study this type of programming language in Chapters 9 and 10.) Now it was no longer necessary to be an electrical engineer to solve a problem on a computer. One simply needed to learn how to write commands in a high-level language. The occupation called **programmer** was born.

This miniaturization process continued into the **third generation** of computing, which lasted from about 1965 to 1975. This was the era of the **integrated circuit**. Rather than using discrete electronic components, integrated circuits with transistors, resistors, and capacitors were photographically etched onto a piece of silicon, which further reduced the size and cost of computers. From building-sized to room-sized, computers now became desk-sized, and this period saw the birth of the first **minicomputer**—the PDP-1 manufactured by the Digital Equipment Corp. It also saw the birth of the **software industry**, as companies sprang up to provide programs such as

accounting packages and statistical programs to the ever-increasing numbers of computer users. By the mid–1970s, computers were no longer a rarity. They were being widely used throughout industry, government, the military, and education.

The **fourth generation**, 1975–1985, saw the appearance of the first **microcomputer**. Integrated circuit technology had advanced to the point that a complete computer system could be contained on a single circuit board that you could hold in your hand. The desk-sized machine of the early 1970s now became a desktop machine, shrinking to the size of a typewriter. Figure 1.7 shows the Altair 8800, the world's first microcomputer, which appeared in January 1975.

It soon became unusual *not* to see a computer on someone's desk. The software industry poured forth all types of new packages—spreadsheets, databases, and drawing programs—to meet the needs of the burgeoning user population. This era saw the appearance of the first **computer networks**, as users realized that much of the power of computers lies in their facilitation of communication with other users. (We will look at networking in great detail in Chapter 7.) **Electronic mail** became an important application. Because so many users were computer novices, the concept of **user-friendly systems** emerged. This included new **graphical user interfaces** with pull-down menus, icons, and other visual aids to make computing easier and more fun. **Embedded systems**—devices that contain a computer to control their internal operation—first appeared during this generation. Computers were becoming small enough to be placed inside cars, thermostats, microwave ovens, and wristwatches.

The **fifth generation**, 1985–?, is where we are today. However, so much is changing so fast that most computer scientists believe that the concept of distinct generations has outlived its usefulness. In computer science, change

Good Evening, This Is Walter Cronkite

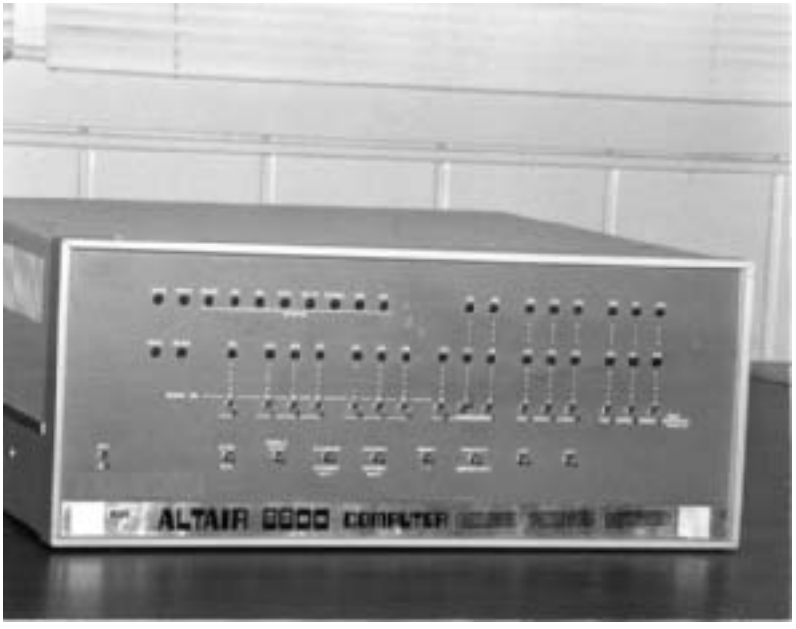
In the earliest days of computing (1951–1952), few people knew what a computer was, and even fewer had seen or worked with one. Computers were the tool of a very small group of highly trained technical specialists in such fields as mathematics, physics, and engineering. In those days, the general public's knowledge of computer science was limited to the robots and alien computers of science fiction movies.

This all changed in November 1952, when millions of Americans turned on their television sets (also a relatively new technology) to watch returns from the 1952 presidential election between Dwight D. Eisenhower and Adlai Stevenson. In addition to seeing Walter Cronkite and TV reporters and analysts, viewers were treated to an unexpected member of the news staff—a UNIVAC I. CBS executives had rented a computer and installed it in the very center of their set, where it sat, lights blinking and tape drives spinning. They planned to use UNIVAC to produce election predictions quickly and scoop rival stations that did their analyses by hand. Ironically, UNIVAC correctly predicted early that evening, on the basis of well-known statistical sampling techniques, that Eisenhower would win the election, but nervous CBS executives were so skeptical about this new technology that they did not go on the air with the computer's prediction until it had been confirmed by old-fashioned manual methods.

It was the first time that millions of TV viewers had actually seen this thing called an electronic digital computer. The CBS staff, who were also quite inexperienced in computer technology, treated the computer as though it were human. They would turn toward the computer console and utter phrases like "UNIVAC, can you tell me who is currently ahead in Ohio?" or "UNIVAC, do you have any prediction on the final electoral vote total?" In actuality, the statistical algorithms had been programmed in, days earlier, by the Remington Rand staff, but it looked great on TV! This first public appearance of a computer was so well received that computers were used many more times in the early days of TV, primarily on quiz shows, where they reinforced the public's image of the computer as a "giant electronic brain."

FIGURE 1.7

The Altair 8800, the World's First Microcomputer



is now a constant companion. Some of the recent developments in computer systems include:

- Massively parallel processors capable of quadrillions (10^{15}) of computations per second
- Handheld digital devices and other types of personal digital assistants (PDAs)
- High-resolution graphics for imaging, animation, movie making, and virtual reality

The World's First Microcomputer

The Altair 8800, shown in Figure 1.7, was the first microcomputer and made its debut on the cover of *Popular Electronics* in January 1975. Its developer, Ed Roberts, owned a tiny electronics store in Albuquerque, New Mexico. His company was in desperate financial shape when he read about a new microprocessor from Intel, the Intel 8080. Roberts reasoned that this new chip could be used to sell a complete personal computer in kit form. He bought these new chips from Intel at the bargain basement price of $75 each and packaged them in a kit called the Altair 8800 (named after a location in the TV series Star Trek), which he offered to hobbyists for $397. Roberts figured he might sell a few hundred kits a year, enough to keep his company afloat temporarily. He ended up selling hundreds of them a day! The Altair microcomputer kits were so popular that he could not keep them in stock, and legend has it that people even drove to New Mexico and camped out in the parking lot to buy their computers.

This is particularly amazing in view of the fact that the original Altair was difficult to assemble and had only 256 memory cells, no I/O devices, and no software support. To program it, the user had to enter binary machine language instructions directly from the console switches. But even though it could do very little, people loved it because it was a real computer, and it was theirs.

The Intel 8080 chip did have the capability of running programs written in the language called BASIC that had been developed at Dartmouth in the early 1960s. A small software company located in Washington state wrote Ed Roberts a letter telling him that it had a BASIC compiler that could run on his Altair, making it much easier to use. That company was called Microsoft—and the rest, as they say, is history.

- Powerful multimedia user interfaces incorporating sound, voice recognition, touch, photography, video, and television
- Integrated digital communication devices incorporating data, television, telephone, fax, the Internet, and the World Wide Web
- Wireless data communications
- Massive storage devices capable of holding one hundred terabytes (10^{14}) of data
- Ubiquitous computing, in which miniature computers are embedded into our cars, cameras, kitchen appliances, home heating systems, and even our clothing

In only a few decades, computers have progressed from the UNIVAC I, which cost millions of dollars, had a few thousand memory locations, and was capable of only a few thousand operations per second, to today's top-of-the-line workstation with a high-resolution flat panel monitor, billions of memory cells, massive amounts of external storage, and enough processing power to execute billions of instructions per second, all for about $1,000. Changes of this magnitude have never occurred so quickly in any other technology. If the same rate of change had occurred in the auto industry, beginning with the 1909 Model-T, today's cars would be capable of traveling at a speed of 20,000 miles per hour, would get about a million miles per gallon, and would cost about $1.00!

Figure 1.8 summarizes the major developments that occurred during each of the five generations of computer development discussed in this section. And underlying all of these amazing improvements, the theoretical model describing the design and construction of computers has not changed significantly in the last 60 years.

However, many people feel that significant and important structural changes are on the way. At the end of Chapter 5 we will introduce models of computing that are fundamentally quite different from the Von Neumann architecture in use today. These totally new approaches (e.g., quantum computing) may be the models used in the twenty-second century and beyond.

1.5 Organization of the Text

This book is divided into six separate sections, called **levels**, each of which addresses one aspect of the definition of computer science that appears at the beginning of this chapter. Let's repeat the definition and see how it maps into the sequence of topics to be presented.

DEFINITION

Computer science the study of algorithms, including

1. Their formal and mathematical properties
2. Their hardware realizations
3. Their linguistic realizations
4. Their applications

FIGURE 1.8

Some of the Major Advancements in Computing

Generation	Approximate Dates	Major Advances
First	1950–1957	First commercial computers First symbolic programming languages Use of binary arithmetic, vacuum tubes for storage Punched card input/output
Second	1957–1965	Transistors and core memories First disks for mass storage Size reduction, increased reliability, lower costs First high-level programming languages First operating systems
Third	1965–1975	Integrated circuits Further reduction in size and cost, increased reliability First minicomputers Time-shared operating systems Appearance of the software industry First set of computing standards for compatibility between systems
Fourth	1975–1985	Large-scale and very-large-scale integrated circuits Further reduction in size and cost, increased reliability First microcomputers Growth of new types of software and of the software industry Computer networks Graphical user interfaces
Fifth	1985–?	Ultra-large-scale integrated circuits Supercomputers and parallel processors Laptops and handheld computers Wireless computing Massive external data storage devices Ubiquitous computing High-resolution graphics, visualization, virtual reality Worldwide networks Multimedia user interfaces Widespread use of digitized sound, images, and movies

Computer science is the study of algorithms, including

1. *Their formal and mathematical properties* Level 1 of the text (Chapters 2 and 3) is titled "The Algorithmic Foundations of Computer Science." It continues the discussion of algorithmic problem solving begun in Sections 1.2 and 1.3 by introducing important mathematical and logical properties of algorithms. Chapter 2 presents the development of a number of algorithms that solve important technical problems—certainly more "technical" than shampooing your hair. It also looks at concepts related to the problem-solving process, such as how we discover and create good algorithms, what notation we can use to express our solutions, and how we can check to see whether our proposed algorithm correctly solves the desired problem.

Our brute force chess example illustrates that it is not enough simply to develop a correct algorithm; we also want a solution that is efficient and that produces the desired result in a reasonable amount of time. (Would you want to market a chess-playing program that takes 10^{48} years to make its first move?) Chapter 3 describes ways to compare the efficiency of different algorithms and select the best one to solve a given problem. The material in Level 1 provides the necessary foundation for a study of the discipline of computer science.

2. *Their hardware realizations* Although our initial look at computer science investigated how an algorithm behaved when executed by some abstract "computing agent," we ultimately want to execute our algorithms on "real" machines to get "real" answers. Level 2 of the text (Chapters 4 and 5) is titled "The Hardware World," and it looks at how to design and construct computer systems. It approaches this topic from two quite different viewpoints.

Chapter 4 presents a detailed discussion of the underlying hardware. It introduces the basic building blocks of computers—binary numbers, transistors, logic gates, and circuits—and shows how these elementary electronic devices can be used to construct components to perform arithmetic and logic functions such as addition, subtraction, comparison, and sequencing. Although it is both interesting and important, this perspective produces a rather low-level view of a computer system. It is difficult to understand how a computer works by studying only these elementary components, just as it would be difficult to understand human behavior by investigating the behavior of individual cells. Therefore, Chapter 5 takes a higher-level view of computer hardware. It looks at computers not as a bunch of wires and circuits but as an integrated collection of subsystems called memory, processor, storage, input/output, and communications. It will explain in great detail the principles of the Von Neumann architecture introduced in Section 1.4.

A study of computer systems can be done at an even higher level. To understand how a computer works, we do not need to examine the functioning of every one of the thousands of components inside a machine. Instead, we need only be aware of a few critical pieces that are essential to our work. From the user's perspective, everything else is superfluous. This "user-oriented" view of a computer system and its resources is called a **virtual machine** or a **virtual environment**. A virtual machine is composed only of the resources that the user perceives rather than of all the hardware resources that actually exist.

This viewpoint is analogous to our level of understanding of what happens under the hood of our car. There may be thousands of mechanical components inside an automobile engine, but most of us concern ourselves only with the items reported on the dashboard—oil pressure, fuel level, engine temperature. This is our "virtual engine," and that is all we need or want to know. We are all too happy to leave the remaining details about engine design to our friendly neighborhood mechanic.

Level 3 (Chapters 6, 7, and 8), titled "The Virtual Machine," describes how a virtual environment is created using a component called **system software**. Chapter 6 takes a look at the most important and widely used piece of system software on a modern computer system, the **operating system**, which controls the overall operation of a computer and makes it easier for users to access. Chapter 7 then goes on to describe how this virtual environment can extend beyond the boundaries of a single system as it examines how to interconnect individual machines into **computer networks** and **distributed systems** that provide users with access to a huge collection of computer systems and information as well as an enormous number of other users. It is the system software,

and the virtual machine it creates, that makes computer hardware manageable and usable. Finally, Chapter 8 discusses a critically important component of a virtual machine—the **security system** that validates who you are and ensures that you are not attempting to carry out an improper, illegal, or unsafe operation. As computers become central to the management of such sensitive data as medical records, military information, and financial data, this aspect of system software is taking on even greater importance.

3. *Their linguistic realizations* After studying hardware design, computer organization, and virtual machines, you will have a good idea of the techniques used to design and build computers. In the next section of the text, we ask the question, How can this hardware be used to solve important and interesting problems? Level 4, titled "The Software World" (Chapters 9–12), takes a look at what is involved in designing and implementing computer software. It investigates the programs and instruction sequences executed by the hardware, rather than the hardware itself.

Chapter 9 compares several high-level programming languages and introduces fundamental concepts related to the topic of computer programming regardless of the particular language being studied. This single chapter is certainly not intended to make you a proficient programmer. Instead, its purpose is to illustrate some basic features of modern programming languages and give you an appreciation for the interesting and challenging task of the computer programmer. Rather than print a separate version of this text for each programming language, the textual material specific to each language can be found on the Web site for this text, and you can download the pages for the language specified by your instructor and used in your class. See the Preface of this text for instructions on accessing these Web pages.

There are many programming languages such as C++, Python, Java, and Perl that can be used to encode algorithms. Chapter 10 provides an overview of a number of different languages and language models in current use, including the functional and parallel models. Chapter 11 describes how a program written in a high-level programming language can be translated into the low-level machine language codes first described in Chapter 5. Finally, Chapter 12 shows that, even when we marshal all the powerful hardware and software ideas described in the first 11 chapters, problems exist that cannot be solved algorithmically. Chapter 12 demonstrates that there are, indeed, limits to computing.

4. *Their applications* Most people are concerned not with creating programs but with using programs, just as there are few automotive engineers but many, many drivers. Level 5, titled "Applications" (Chapters 13–16), moves on from *how* to write a program to *what* these programs can do.

Chapters 13 through 16 explore just a few of the many important and rapidly growing applications of computers, such as simulation, visualization, e-commerce, databases, artificial intelligence, and computer graphics and entertainment. This section cannot possibly survey all the ways in which computers are being used today or will be used in the future. Indeed, there is hardly an area in our modern, complex society that is not affected in some important way by information technology. Readers interested in applications not discussed should seek readings specific to their own areas of interest.

Some computer science professionals are not concerned with building computers, creating programs, or using any of the applications just described. Instead, they are interested in the social and cultural impact—both positive and negative—of this ever-changing technology. The sixth level of this text addresses this important perspective on computer science. This is not part of the original

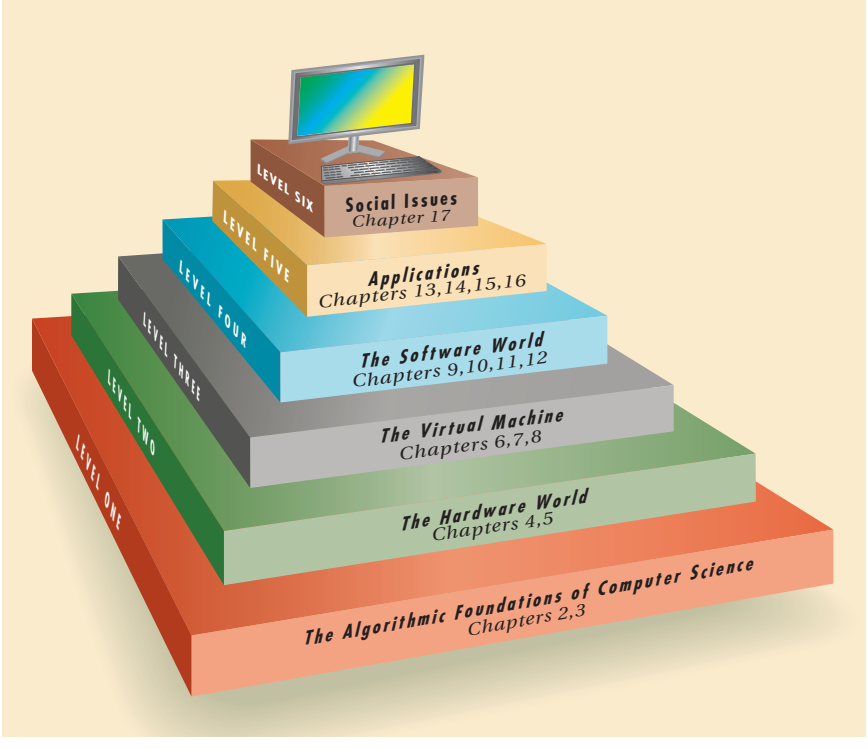
definition of computer science but has become an important area of study. In Level 6, titled "Social Issues" (Chapter 17), we move to the highest level of abstraction—the view furthest removed from the computer itself—to discuss social, ethical, legal, and professional issues related to computer and information technology. These issues are critically important, because even individuals not directly involved in developing or using computers are deeply affected by them, just as society has been drastically and permanently altered by such technological developments as the telephone, television, and automobile. This last chapter takes a look at such thorny and difficult topics as computer crime, information privacy, and intellectual property. Because it's impossible to resolve the complex questions that arise in these areas, our intent is simply to raise your awareness and provide some decision-making tools to help you reach your own conclusions.

The overall six-layer hierarchy of this text is summarized in Figure 1.9. The organizational structure diagrammed in Figure 1.9 is one of the most important aspects of this text. To describe a field of study, it is not enough to present a mass of facts and explanations. For learners to absorb, understand, and integrate this information, there must be a theme, a relationship, a thread that ties together the various parts of the narrative—in essence, a "big picture." Our big picture is Figure 1.9.

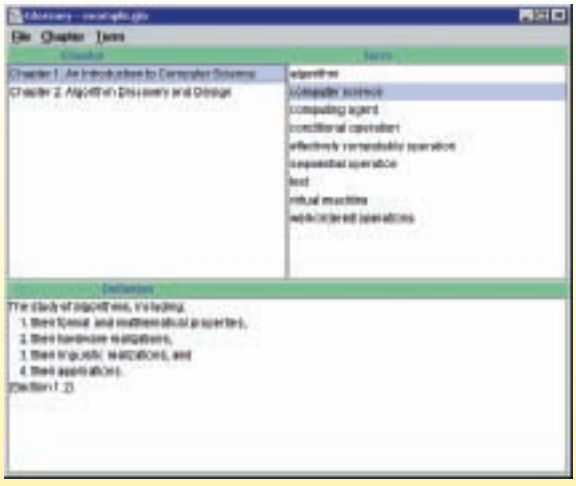
We first lay out the basic foundations of computer science (Level 1). We then proceed upward through five distinct layers of abstraction, from extremely low-level machine details such as electronic circuits and computer hardware (Level 2), through intermediate levels that address virtual machines (Level 3), programming languages and software development (Level 4), to higher levels that investigate computer applications (Level 5), and address the use and misuse of information technology (Level 6). The material in each level provides a foundation to reveal the beauty and complexity of a higher and more abstract view of the discipline of computer science.

FIGURE 1.9

Organization of the Text into a Six-Layer Hierarchy



LABORATORY EXPERIENCE 1



Associated with this text is a laboratory manual that includes software packages and a collection of formal laboratory exercises. These laboratory experiences are designed to give you a chance to build on, modify, and experiment with the ideas discussed in the text. You are strongly encouraged to carry out these laboratories to gain a deeper understanding of the concepts presented in the chapters. Learning computer science involves not just reading and listening but also doing and trying. Our laboratory exercises will give you that chance. (In addition, we hope that you will find them fun.)

Laboratory Experience 1, titled "A Glossary and Web Browsing," introduces the fundamental operations that you will need in all future labs—operations such as using menus, buttons, and windows and accessing pages on the Web. (In the text, you will find a number of pointers to Web pages containing a wealth of information that complements our discussions.) In addition, the lab provides a useful tool that you may use during your study of computer science and in other courses as well. You will learn how to use a computer to build a *glossary* of important technical terms along with their definitions and locations in the text. Not only will the lab introduce you to some essential skills, but it will also allow you to create your own glossary of important terms and definitions as shown in this screen shot from the lab.

Please turn to Laboratory Experience 1 in the laboratory manual and try it now.

EXERCISES

1. Identify some algorithms, apart from DVR instructions and cooking recipes, that you encounter in your everyday life. Write them out in any convenient notation, and explain how they meet all of the criteria for algorithms presented in this chapter.
2. In the DVR instructions in Figure 1.1, step 4 says, "Enter the channel number that you wish to record and press the button labeled CHAN." Is that an unambiguous and well-defined operation? Explain why or why not.
3. Trace through the decimal addition algorithm of Figure 1.2 using the following input values:

   $m = 3 \quad a_2 = 1\ a_1 = 4\ a_0 = 9$

   $b_2 = 0\ b_1 = 2\ b_0 = 9$

   At each step, show the values for c_3, c_2, c_1, c_0, and *carry*.
4. Modify the decimal addition algorithm of Figure 1.2 so that it does not print out nonsignificant leading zeroes; that is, the answer to question 3 would appear as 178 rather than 0178.
5. Under what conditions would the well-known quadratic formula

   $$Roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   not be effectively computable? (Assume that you are working with real numbers.)
6. Compare the two solutions to the shampooing algorithm shown in Figures 1.3(a) and 1.3(b). Which do you think is a better general-purpose solution? Why? (*Hint:* What if you wanted to wash your hair 1,000 times?)
7. The following is Euclid's 2,300-year-old algorithm for finding the greatest common divisor of two positive integers *I* and *J*.

Step	***Operation***
1	Get two positive integers as input. Call the larger value *I* and the smaller value *J*.
2	Divide *I* by *J,* and call the remainder *R*.
3	If *R* is *not* 0, then reset *I* to the value of *J,* reset *J* to the value of *R,* and go back to step 2.
4	Print out the answer, which is the value of *J*.
5	Stop.

   a. Go through this algorithm using the input values 20 and 32. After each step of the algorithm is completed, give the values of *I, J,* and *R*. Determine the final output of the algorithm.

   b. Does the algorithm work correctly when the two inputs are 0 and 32? Describe exactly what happens, and modify the algorithm so that it gives an appropriate error message.
8. A salesperson wants to visit 25 cities while minimizing the total number of miles she has to drive. Because she has studied computer science, she decides to design an algorithm to determine the optimal order in which to visit the cities to (1) keep her driving distance to a minimum, and (2) visit each city exactly once. The algorithm that she has devised is the following:

   > The computer first lists *all* possible ways to visit the 25 cities and then, for each one, determines the total mileage associated with that particular ordering. (Assume that the computer has access to a road map that provides the distances between all cities.) After determining the total mileage for each possible trip, the computer searches for the ordering with the minimum mileage and prints out the list of cities on that optimal route, that is, the order in which the salesperson should visit her destinations.

   If a computer could analyze 10,000,000 separate paths per second, how long would it take to determine the optimal route for visiting these 25 cities? On the basis of your answer, do you think this is a feasible algorithm? If it is not, can you think of a way to obtain a reasonable solution to this problem?
9. One way to do multiplication is by repeated addition. For example, 47 × 25 can be evaluated as 47 + 47 + 47 + . . . + 47 (25 times). Sketch out an algorithm for multiplying two positive numbers *a* and *b* using this technique.
10. Read about one of the early pioneers mentioned in this chapter—Pascal, Liebnitz, Jacquard, Babbage, Lovelace, Hollerith, Eckert, Mauchly, Aiken, Zuse, Atanasoff, Turing, or Von Neumann. Write a paper describing in detail that person's contribution to computing and computer science.
11. Get the technical specifications of the computer on which you are working (either from a technical manual or from your computer center staff). Determine its cost, its processing speed (in MIPS, millions of instructions per second), its computational speed (in MFlops, millions of floating point operations per second), and the size of its primary memory.

    Compare those values with what was typically available on first-, second-, and third-generation computer systems, and calculate the percentage improvement between your computer and the first commercial machines of the early 1950s.
12. A new and growing area of computer science is *ubiquitous computing,* in which a number of computers automatically provide services for a user without that user's knowledge or awareness. For example, a computer located in your car contacts the garage door opener and tells it to open the garage door when the car is close to home. Read about this new model of computing and write a paper describing some of its applications. What are some of the possible problems that could be created?
13. A standard computer CD holds approximately 700 million characters. Estimate how many linear feet of shelf space are required to store 700 million characters encoded as text (i.e., printed, bound books) rather than as electronic media. Assume there are 5 characters per word, 300 words per page, and 300 pages/inch of shelf.

CHALLENGE WORK

1. Assume we have a "computing agent" that knows how to do one-digit subtraction where the first digit is at least as large as the second (i.e., we do not end up with a negative number). Thus, our computing agent can do such operations as 7 – 3 = 4, 9 – 1 = 8, and 5 – 5 = 0. It can also subtract a one-digit value from a two-digit value in the range 10–18 as long as the final result has only a single digit. This capability enables it to do such operations as 13 – 7 = 6, 10 – 2 = 8, and 18 – 9 = 9.

   Using these primitive capabilities, design an algorithm to do *decimal subtraction* on two m-digit numbers, where $m \geq 1$. You will be given two unsigned whole numbers $a_{m-1}\ a_{m-2}\ldots a_0$ and $b_{m-1}\ b_{m-2}\ldots b_0$. Your algorithm must compute the value $c_{m-1}\ c_{m-2}\ldots c_0$, the difference of these two values.

$$\begin{array}{r} a_{m-1}\ a_{m-2}\ldots a_0 \\ -b_{m-1}\ b_{m-2}\ldots b_0 \\ \hline c_{m-1}\ c_{m-2}\ldots c_0 \end{array}$$

   You may assume that the top number ($a_{m-1}\ a_{m-2}\ldots a_0$) is greater than or equal to the bottom number ($b_{m-1}\ b_{m-2}\ldots b_0$) so that the result is not a negative value. However, do not assume that each individual digit a_i is greater than or equal to b_i. If the digit on the bottom is larger than the digit on the top, then you must implement a *borrowing scheme* to allow the subtraction to continue. (*Caution*: It may have been easy to learn subtraction as a first grader, but it is devilishly difficult to tell a computer how to do it!)

2. Our definition of the field of computer science is only one of many that have been proposed. Because it is so young, people working in the field are still debating how best to define exactly what they do. Review the literature of computer science (perhaps some of the books listed in the next section) and browse the Web to locate other definitions of computer science. Compare these definitions with the one presented in this chapter and discuss the differences among them. Discuss how different definitions may give you a vastly different perspective on the field and what people in this field do. [*Note:* A very well-known and widely used definition of computer science was presented in "Report of the ACM Task Force on the Core of Computer Science," reprinted in the journal *Communications of the ACM,* vol. 32, no. 1 (January 1989).]

FOR FURTHER READING

The following books provide a good introduction to and overview of the field of computer science. Like this text, they survey many different aspects of the discipline.

Biermann, A. W. *Great Ideas in Computer Science,* 2nd ed. Cambridge, MA: MIT Press, 1997.

Brookshear, J. G. *Computer Science: An Overview,* 10th ed. Reading, MA: Addison Wesley, 2008.

Decker, R., and Hirshfield, S. *The Analytical Engine: An Introduction to Computer Science Using the Internet,* Boston, MA: Course Technology, 2004.

Dewdney, A. K. *The New Turing Omnibus*. New York: Freeman, 2001.

Dewdney, A. K. *Introductory Computer Science: Bits of Theory, Bytes of Practice*. Boston, MA: W.H. Freeman & Company, 1996.

Snyder, Lawrence, *Fluency with Information Technology,* 3rd ed. Prentice Hall, 2008.

The following books provide an excellent overview of the historical development of both computers and software.

Broy, M., and Denert, E. *Software Pioneers*. Amsterdam: Springer-Verlag, 2002.

Cambell-Kelly, M., and Asprey, W. *Computers: A History of the Information Machine*. New York: Basic Books, 1997.

Ceruzzi, P. *A History of Modern Computing*. 2nd Edition, Cambridge, MA: MIT Press, 2003.

Ifrah, George. *The Universal History of Computing: From the Abacus to Quantum Computer*. New York: Wiley, 2002.

Rojas, Ral, Hashagen, Ulf, Rojas, Raul, The First Computers—Their History and Architecture, Cambridge, MA: MIT Press, 2002.

Wurster, C. *The Computer: An Illustrated History*. Cologne, Germany: Taschen, 2002

In addition, the Charles Babbage Institute at the University of Minnesota is an outstanding resource for information about the history of information technology and its impact on society. Its Web site is at www.cbi.umn.edu.

This page intentionally left blank

CHAPTER 4

The Building Blocks: Binary Numbers, Boolean Logic, and Gates

4.1 Introduction

Level 1 of the text investigated the algorithmic foundations of computer science. It developed algorithms for searching tables, finding largest and smallest values, locating patterns, sorting lists, and cleaning up bad data. It also showed how to analyze and evaluate algorithms to demonstrate that they are not only correct but efficient and useful as well.

Our discussion assumed that these algorithms would be executed by something called a **computing agent**, an abstract concept representing any object capable of understanding and executing our instructions. We didn't care what that computing agent was—person, mathematical model, computer, or robot. However, in this section of the text we *do* care what our computing agent looks like and how it is able to execute instructions and produce results.

In this chapter we introduce the fundamental building blocks of all computer systems—binary representation, Boolean logic, gates, and circuits.

4.2 The Binary Numbering System

Our first concern with learning how to build computers is understanding how computers represent information. Their internal storage techniques are quite different from the way you and I represent information in our notebooks, desks, and filing cabinets.

4.2.1 *Binary Representation of Numeric and Textual Information*

People generally represent numeric and textual information (language differences aside) by using the following notational conventions:

a. The 10 decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 for numeric values such as 459.

b. *Sign/magnitude notation* for signed numbers—that is, a + or – sign placed immediately to the left of the digits; +31 and –789 are examples.

c. *Decimal notation* for real numbers, with a decimal point separating the whole number part from the fractional part; an example is 12.34.

d. The 26 letters A, B, C, . . . , Z for textual information (as well as lowercase letters and a few special symbols for punctuation).

You might suppose that these well-known schemes are the same conventions that computers use to store information in memory. Surprisingly, this is not true.

There are two types of information representation: The **external representation** of information is the way information is represented by humans and the way it is entered at a keyboard or displayed on a printer or screen. The **internal representation** of information is the way it is stored in the memory of a computer. This difference is diagrammed in Figure 4.1.

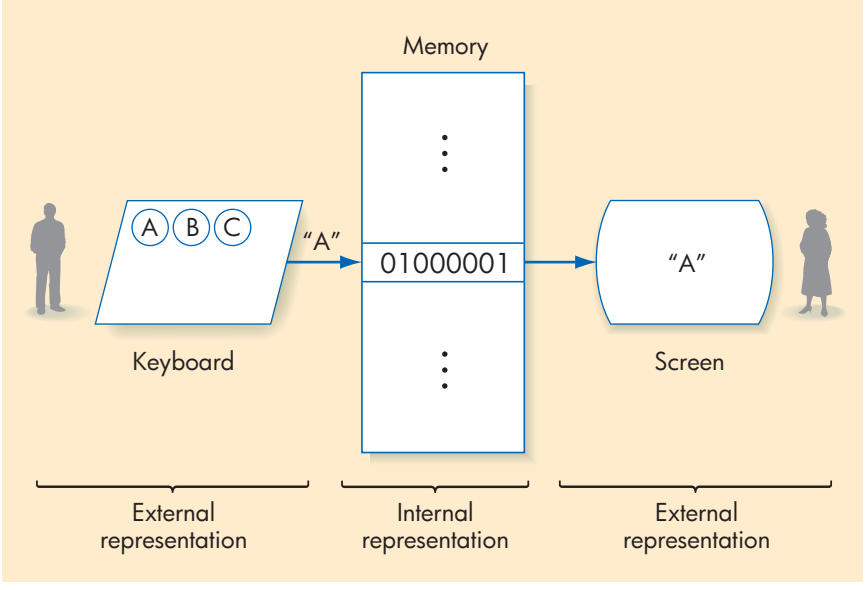
Externally, computers do use decimal digits, sign/magnitude notation, and the 26-character alphabet. However, virtually every computer ever built stores data—numbers, letters, graphics, images, sound—internally using the **binary numbering system**.

Binary is a **base-2 positional numbering system** not unlike the more familiar decimal, or base-10, system used in everyday life. In these systems, the value or "worth" of a digit depends not only on its absolute value but also on its specific position within a number. In the decimal system there are 10 unique digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9), and the value of the positions in a decimal number is based on powers of 10. Moving from right to left in a number, the positions represent ones (10^0), tens (10^1), hundreds (10^2), thousands (10^3), and so on. Therefore, the decimal number 2,359 is evaluated as follows:

$$(2 \times 10^3) + (3 \times 10^2) + (5 \times 10^1) + (9 \times 10^0)$$
$$= 2{,}000 + 300 + 50 + 9$$
$$= 2{,}359$$

The same concepts apply to binary numbers except that there are only two digits, 0 and 1, and the value of the positions in a binary number is based on powers of 2. Moving from right to left, the positions represent ones (2^0), twos (2^1), fours (2^2), eights (2^3), sixteens (2^4), and so on. The two digits, 0 and 1, are frequently referred to as **bits**, a contraction of the two words *b*inary dig*its*.

FIGURE 4.1
Distinction Between External and Internal Representation of Information



For example, the 6-digit binary number 111001 is evaluated as follows:

$$\begin{aligned} 111001 &= (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= 32 + 16 + 8 + 0 + 0 + 1 \\ &= 57 \end{aligned}$$

The 5-digit binary quantity 10111 is evaluated in the following manner:

$$\begin{aligned} 10111 &= (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &= 16 + 0 + 4 + 2 + 1 \\ &= 23 \end{aligned}$$

Evaluating a binary number is quite easy, because 1 times any value is simply that value, and 0 times any value is always 0. Thus, when evaluating a binary number, use the following **binary-to-decimal algorithm**: Whenever there is a 1 in a column, add the positional value of that column to a running sum, and whenever there is a 0 in a column, add nothing. The final sum is the decimal value of this binary number. This is the procedure we followed in the previous two examples.

A binary-to-decimal conversion table for the values 0–31 is shown in Figure 4.2. You may want to evaluate a few of the binary values using this algorithm to confirm their decimal equivalents.

Any whole number that can be represented in base 10 can also be represented in base 2, although it may take more digits because a single decimal digit contains more information than a single binary digit. Note that in the first example shown above it takes only 2 decimal digits (5 and 7) to represent the quantity 57 in base 10, but it takes 6 binary digits (1, 1, 1, 0, 0, and 1) to express the same value in base 2.

To go in the reverse direction—that is, to convert a decimal value into its binary equivalent—we use the **decimal-to-binary algorithm**, which is based on successive divisions by 2. Dividing the original decimal value by 2 produces a quotient and a remainder, which must be either a 0 or a 1. Record the remainder digit and then divide the quotient by 2, getting a new quotient and a second remainder digit. The process of dividing by 2, saving the quotient, and

FIGURE 4.2

Binary-to-Decimal Conversion Table

Binary	Decimal	Binary	Decimal
0	0	10000	16
1	1	10001	17
10	2	10010	18
11	3	10011	19
100	4	10100	20
101	5	10101	21
110	6	10110	22
111	7	10111	23
1000	8	11000	24
1001	9	11001	25
1010	10	11010	26
1011	11	11011	27
1100	12	11100	28
1101	13	11101	29
1110	14	11110	30
1111	15	11111	31

writing down the remainder is repeated until the quotient equals 0. The sequence of remainder digits, when written left to right from the last remainder digit to the first, is the binary representation of the original decimal value. For example, here is the conversion of the decimal value 19 into binary:

	Convert the value 19 to binary:		
19 ÷ 2	quotient = 9	remainder = 1	
9 ÷ 2	quotient = 4	remainder = 1	order for
4 ÷ 2	quotient = 2	remainder = 0	reading the
2 ÷ 2	quotient = 1	remainder = 0	remainder
1 ÷ 2	quotient = 0	remainder = 1	digits
	Stop, since the quotient is now 0.		

In this example, the remainder digits, when written left-to-right from the last one to the first, are 10011. This is the binary form of the decimal value 19. To confirm this, we can convert this value back to decimal form using the binary-to-decimal algorithm.

$$\begin{aligned} 10011 &= (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\ &= 16 + 0 + 0 + 2 + 1 \\ &= 19 \end{aligned}$$

In every computer there is a maximum number of binary digits that can be used to store an integer. Typically, this value is 16, 32, or 64 bits. Once we have fixed this maximum number of bits (as part of the design of the computer), we also have fixed the largest unsigned whole number that can be represented in this computer. For example, Figure 4.2 used at most 5 bits to represent binary numbers. The largest value that could be represented is 11111, not unlike the number 99999, which is the maximum mileage value that can be represented on a 5-digit decimal odometer. 11111 is the binary representation for the decimal integer 31. If there were 16 bits available, rather than 5, then the largest integer that could be represented is

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

A Not So Basic Base

The decimal system has been in use for so long that most people cannot imagine using a number system other than base 10. Tradition says it was chosen because we have 10 fingers and 10 toes. However, the discussion of the past few pages should convince you that there is nothing unique or special about decimal numbering, and the basic operations of arithmetic (addition, subtraction, multiplication, and division) work just fine in other bases, such as base 2. In addition to binary, computer science makes frequent use of *octal* (base 8) and *hexadecimal* (base 16). Furthermore, it is not only computers that utilize non-decimal bases. For example, the Native American Yuki tribe of Northern California reportedly used base 4, or *quaternary* numbers, counting using the spaces between fingers rather than on the fingers themselves. The pre-Columbian Mayans of Mexico and Central America used a *vigesimal* system, or base 20, while ancient Babylonians employed *sexagesimal,* or base 60 (and we are quite sure that members of both cultures had the same number of fingers and toes as twenty-first-century human beings!).

This quantity is $2^{15} + 2^{14} + \ldots + 2^2 + 2^1 + 2^0 = 65{,}535$. Unsigned integers larger than this cannot be represented with 16 binary digits. Any operation on this computer that produces an unsigned value greater than 65,535 results in the error condition called **arithmetic overflow**. This is an attempt to represent an integer that exceeds the maximum allowable value. The computer could be designed to use more than 16 bits to represent integers, but no matter how many bits are ultimately used, there is always a maximum value beyond which the computer cannot correctly represent any integer. This characteristic is one of the major differences between the disciplines of mathematics and computer science. In mathematics a quantity may usually take on any value, no matter how large. Computer science must deal with a finite—and sometimes quite limited—set of possible representations, and it must handle the errors that occur when those limits are exceeded.

Arithmetic in binary is quite easy because we have only 2 digits to deal with rather than 10. Therefore, the rules that define arithmetic operations such as addition and subtraction have only $2 \times 2 = 4$ entries, rather than the $10 \times 10 = 100$ entries for decimal digits. For example, here are the four rules that define binary addition:

```
  0      0      1      1
+ 0    + 1    + 0    + 1
  0      1      1     10   (that is, 0 with a carry of 1.)
```

The last rule says that 1 + 1 = 10, which has the decimal value 2.

To add two binary numbers you use the same technique first learned in grade school. Add each column one at a time from right to left, using the binary addition rules shown above. In the column being added you write the sum digit under the line and any carry digit produced is written above the next column to the left. For example, addition of the two binary values 7 (00111) and 14 (01110) proceeds as follows:

```
  00111   (the binary value 7)
+ 01110   (the binary value 14)
```

Start by adding the two digits in the rightmost column—the 1 and 0. This produces a sum of 1 and a carry digit of 0; the carry digit gets "carried" to the second column.

```
    0       ← carry digit
  00111
+ 01110
    1
```

Now add the carry digit from the previous column to the two digits in the second column, which gives 0 + 1 + 1. From the rules above, we see that the (0 + 1) produces a 1. When this is added to the value 1 it produces a sum of 0 and a new carry digit of 1.

```
    1       ← carry digit
  00111
+ 01110
   01
```

Adding the two digits in the third column plus the carry digit from the second column produces 1 + 1 + 1, which is 11, or a sum of 1 and a new carry digit of 1.

```
  1      ← carry digit
 00111
+01110
  101
```

Continuing in this right-to-left manner until we reach the leftmost column produces the final result, 10101 in binary, or 21 in decimal.

```
 00111
+01110
 10101   (the value 21 = 16 + 4 + 1)
```

SIGNED NUMBERS. Binary digits can represent not only whole numbers but also other forms of data, including signed integers, decimal numbers, and characters. For example, to represent signed integers, we can use the leftmost bit of a number to represent the sign, with 0 meaning positive (+) and 1 meaning negative (–). The remaining bits are used to represent the magnitude of the value. This form of signed integer representation is termed **sign/magnitude notation**, and it is one of a number of different techniques for representing positive and negative whole numbers. For example, to represent the quantity –49 in sign/magnitude, we could use seven binary digits with one bit for the sign and six bits for the magnitude:

$$\underbrace{1}_{-}\ \underbrace{1\,1\,0\,0\,0\,1}_{49} \qquad (2^5 + 2^4 + 2^0 = 32 + 16 + 1 = 49)$$

The value +3 would be stored like this:

$$\underbrace{0}_{+}\ \underbrace{0\,0\,0\,0\,1\,1}_{3} \qquad (2^1 + 2^0 = 2 + 1 = 3)$$

You may wonder how a computer knows that the 7-digit binary number 1110001 in the first example above represents the signed integer value –49 rather than the unsigned whole number 113.

$$\begin{aligned} 1110001 &= (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^0) \\ &= 64 + 32 + 16 + 1 \\ &= 113 \end{aligned}$$

The answer to this question is that a computer does *not* know. A sequence of binary digits can have many different interpretations, and there is no fixed, predetermined interpretation given to any binary value. A binary number stored in the memory of a computer takes on meaning only because it is used in a certain way. If we use the value 1110001 as though it were a signed integer, then it will be interpreted that way and will take on the value –49. If it is used, instead, as an unsigned whole number, then that is what it will become, and it will be interpreted as the value 113. The meaning of a binary number stored in memory is based solely on the context in which it is used.

Initially this may seem strange, but we deal with this type of ambiguity all the time in natural languages. For example, in the Hebrew language, letters of the alphabet are also used as numbers. Thus the Hebrew character aleph (א) can stand for either the letter A or the number 1. The only way to tell which meaning is appropriate is to consider the context in which the character is used. Similarly, in English the word *ball* can mean either a round object used to play games or an elegant formal party. Which interpretation is correct? We cannot say without knowing the context in which the word is used. The same is true for values stored in the memory of a computer system. It is the context that determines the meaning of a binary string.

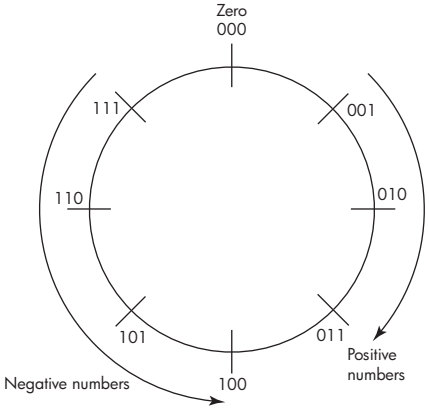
Sign/magnitude notation is quite easy for people to work with and understand, but, surprisingly, it is used rather infrequently in real computer systems. The reason is the existence of the very "messy" and unwanted signed number: 10000 . . . 0000. Because the leftmost bit is a 1, this value is treated as negative. The magnitude is 0000 . . . 0000. Thus this bit pattern represents the numerical quantity "negative zero," a value that has no real mathematical meaning and should not be distinguished from the other representation for zero, 00000 . . . 0000. The existence of two distinct bit patterns for a single numerical quantity causes some significant problems for computer designers.

For example, assume we are executing the following algorithmic operation on two signed numbers a and b

```
if (a = b)
     do operation 1
else
     do operation 2
```

when a has the value 0000 . . . 0 and b has the value 1000 . . . 0. Should they be considered equal to each other? Numerically the value –0 does equal +0, so maybe we should do operation 1. However, the two bit patterns are not identical, so maybe these two values are not equal, and we should do operation 2. This situation can result in programs that execute in different ways on different machines.

Therefore, computer designers tend to favor signed integer representations that do not suffer from the problem of two zeros. One of the most widely used is called **two's complement representation**. To understand how this method works you need to write down, in circular form, all binary patterns from 000 . . . 0 to 111 . . . 1 in increasing order. Here is what that circle might look like using 3-digit numbers:



In this diagram the positive numbers begin at 000 and proceed in order around the circle to the right. Negative numbers begin at 111 and proceed in order around the circle to the left. The leftmost digit specifies whether the number is to be given a positive interpretation (leftmost bit = 0) or a negative interpretation (leftmost bit = 1).

Bit pattern	*Decimal value*
000	0
001	+1
010	+2
011	+3
100	-4
101	-3
110	-2
111	-1

In this representation if we add, for example, 3 + (–3), we get 0, as expected:

11 ← carry digits
011
101
000 (*Note:* The 1 that carries into column 4 can be discarded.)

Note that in the two's complement representation there is only a single zero, the binary number 000 . . . 0. However, the existence of a single pattern for zero leads to another unusual situation. The total number of values that can be represented with n bits is 2^n, which is always an even number. In the previous example $n = 3$, so there were $2^3 = 8$ possible values. One of these is used for 0, leaving seven remaining values, which is an odd number. It is impossible to divide these seven patterns equally between the positive and negative numbers, and in this example we ended up with four negative values but only three positive ones. The pattern that was previously "negative zero" (100) now represents the value –4, but there is no equivalent number on the positive side, that is, there is no binary pattern that represents +4. In the two's complement representation of signed integers you can always represent one more negative number than positive. This is not as severe a problem as having two zeros, though, and two's complement is widely used for representing signed numbers inside a computer.

This has been only a brief introduction to the two's complement representation. A Challenge Work problem at the end of this chapter invites you to investigate further the underlying mathematical foundations of this interesting representational technique.

FRACTIONAL NUMBERS. Fractional numbers, such as 12.34 and –0.001275, can also be represented in binary by using the signed-integer techniques we have just described. To do that, however, we must first convert the number to **scientific notation**:

$$\pm M \times B^{\pm E}$$

where M is the **mantissa,** B is the **exponent base** (usually 2), and E is the **exponent**. For example, assume we want to represent the decimal quantity +5.75. In addition, assume that we will use 16 bits to represent the number, with 10 bits allocated for representing the mantissa and 6 bits for the exponent. (The exponent base B is assumed to be 2 and is not explicitly stored.) Both the mantissa and the exponent are signed integer numbers, so we can use either the sign/magnitude or two's complement notations that we just learned to represent each of these two fields. (In all the following examples we have chosen to use sign/magnitude notation.)

In binary, the value 5 is 101. To represent the fractional quantity 0.75, we need to remember that the bits to the right of the decimal point (or binary point in our case) have the positional values r^{-1}, r^{-2}, r^{-3}, and so on, where r is the base of the numbering system used to represent the number. When using decimal these position values are the tenths (10^{-1}), hundredths (10^{-2}), thousandths (10^{-3}), and so on. Because r is 2 in our case, the positional values of the digits to the right of the binary point are halves (2^{-1}), quarters (2^{-2}), eighths (2^{-3}), sixteenths (2^{-4}), and so on. Thus,

$$0.75 = 1/2 + 1/4 = 2^{-1} + 2^{-2} \text{ (which in binary is 0.11)}$$

Therefore, in binary 5.75 = 101.11. Using scientific notation, and an exponent base B = 2, we can write this value as

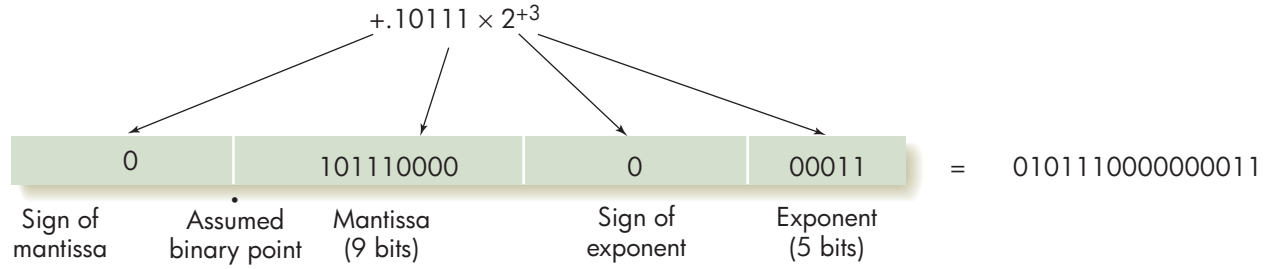
$$5.75 = 101.11 \times 2^0$$

Next, we must **normalize** the number so that its first significant digit is immediately to the right of the binary point. As we move the binary point, we adjust the value of the exponent so that the overall value of the number remains unchanged. If we move the binary point to the left one place (which makes the value smaller by a factor of 2), then we add 1 to the exponent (which makes it larger by a factor of 2). We do the reverse when we move the binary point to the right.

$$\begin{aligned} 5.75 &= 101.11 \times 2^0 \\ &= 10.111 \times 2^1 \\ &= 1.0111 \times 2^2 \\ &= .10111 \times 2^3 \end{aligned}$$

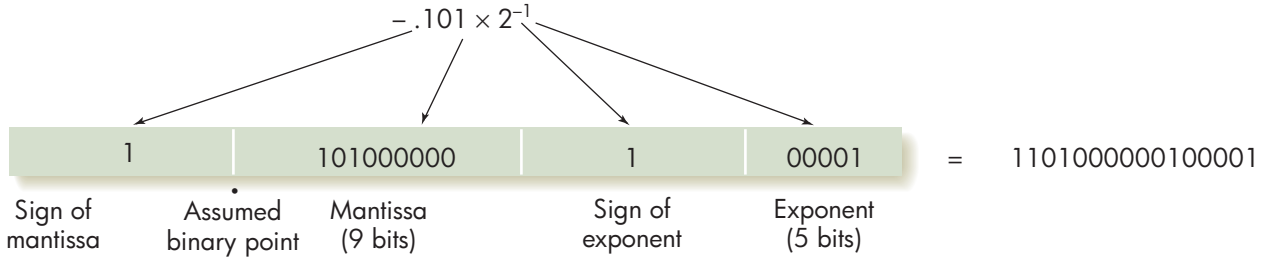
(which is $(1/2 + 1/8 + 1/16 + 1/32) \times 8 = 5.75$)

We now have the number in the desired format and can put all the pieces together. We separately store the mantissa (excluding the binary point, which is assumed to be to the left of the first significant digit) and the exponent, both of which are signed integers and can be represented in sign/magnitude notation. The mantissa is stored with its sign—namely, 0, because it is a positive quantity—followed by the assumed binary point, followed by the magnitude of the mantissa, which in this case is 10111. Next we store the exponent, which is +3, or 000011 in sign/magnitude. The overall representation, using 16 bits, is



For another example, let's determine the internal representation of the fraction –5/16.

$$
\begin{aligned}
-5/16 &= -(1/4 + 1/16) \\
&= -.0101 \times 2^{0} \quad \text{(this is the value } -5/16 \text{ in scientific notation)} \\
&= -.101 \times 2^{-1} \quad \text{(after normalization)}
\end{aligned}
$$



TEXTUAL INFORMATION. To represent textual material in binary, the system assigns to each printable letter or symbol in our alphabet a unique number (this assignment is called a **code mapping**), and then it stores that symbol internally using the binary equivalent of that number. For example, here is one possible mapping of characters to numbers, which uses 8 bits to represent each character.

Symbol	*Decimal Value*	*Binary (Using 8 Binary Digits)*
A	1	00000001
B	2	00000010
C	3	00000011
D	4	00000100
⋮	⋮	⋮
Z	26	00011010
⋮	⋮	⋮
@	128	10000000
!	129	10000001
⋮	⋮	⋮

To store the 4-character string "BAD!" in memory, the computer would store the binary representation of each individual character using the above 8-bit code.

$$\text{BAD!} = \underbrace{00000010}_{\text{B}}\ \underbrace{00000001}_{\text{A}}\ \underbrace{00000100}_{\text{D}}\ \underbrace{10000001}_{\text{!}}$$

We have indicated above that the 8-bit numeric quantity 10000001 is interpreted as the character "!". However, as we mentioned earlier, the only way a computer knows that the 8-bit value 10000001 represents the symbol "!" and not the unsigned integer value 129 (128 + 1) or the signed integer value –1 (sign bit = negative, magnitude is 1) is by the context in which it is used. If these 8 bits are sent to a display device that expects to be given characters, then this value will be interpreted as an "!". If, on the other hand, this 8-bit value is sent to an arithmetic unit that adds unsigned numbers, then it will be interpreted as a 129 in order to make the addition operation meaningful.

To facilitate the exchange of textual information, such as word processing documents and electronic mail, between computer systems, it would be most helpful if everyone used the same code mapping. Fortunately, this is pretty much the case. Currently the most widely used code for representing characters internally in a computer system is called **ASCII**, an acronym for the *A*merican *S*tandard *C*ode for *I*nformation *I*nterchange. ASCII is an international standard for representing textual information in the majority of computers. It uses 8 bits to represent each character, so it is able to encode a total of $2^8 = 256$ different characters. These are assigned the integer values 0 to 255. However, only the numbers 32 to 126 have been assigned so far to printable characters. The remainder either are unassigned or are used for nonprinting control characters such as tab, form feed, and return. Figure 4.3 shows the ASCII conversion table for the numerical values 32–126.

However, a new code set called **UNICODE** is rapidly gaining popularity because it uses a 16-bit representation for characters rather than the 8-bit format of ASCII. This means that it is able to represent $2^{16} = 65{,}536$ unique characters instead of the $2^8 = 256$ of ASCII. It may seem like 256 characters are more than enough to represent all the textual symbols that we would ever need—for example, 26 uppercase letters, 26 lowercase letters, 10 digits, and a few dozen special symbols, such as +=–{}][\:"?><.,;%$#@. Add that all together and it still totals only about 100 symbols, far less than the 256 that can be represented in ASCII. However, that is true only if we limit our work to Arabic numerals and the Roman alphabet. The world grows more connected all the time—helped along by computers, networks, and the Web—and it is critically important that computers represent and exchange textual information using alphabets in addition to these 26 letters and 10 digits. When we start assigning codes to symbols drawn from alphabets such as Russian, Arabic, Chinese, Hebrew, Greek, Thai, Bengali, and Braille, as well as mathematical symbols and special linguistic marks such as tilde, umlaut, and accent grave, it becomes clear that ASCII does not have nearly enough room to represent them all. However, UNICODE, with space for over 65,000 symbols, is large enough to accommodate all these symbols and many more to come. In fact, UNICODE has defined standard code mappings for more than 50,000 symbols from literally hundreds of alphabets, and it is a way for users around the world to share textual information regardless of the language in which they are writing. The UNICODE home page, which gives all the current standard mappings, is located at *www.unicode.org*.

FIGURE 4.3

ASCII Conversion Table

Keyboard Character	Binary ASCII Code	Integer Equivalent	Keyboard Character	Binary ASCII Code	Integer Equivalent
(blank)	00100000	32	P	01010000	80
!	00100001	33	Q	01010001	81
“	00100010	34	R	01010010	82
#	00100011	35	S	01010011	83
$	00100100	36	T	01010100	84
%	00100101	37	U	01010101	85
&	00100110	38	V	01010110	86
‘	00100111	39	W	01010111	87
(	00101000	40	X	01011000	88
)	00101001	41	Y	01011001	89
*	00101010	42	Z	01011010	90
+	00101011	43	[	01011011	91
´	00101100	44	\	01011100	92
–	00101101	45	]	01011101	93
.	00101110	46	^	01011110	94
/	00101111	47	_	01011111	95
0	00110000	48	`	01100000	96
1	00110001	49	a	01100001	97
2	00110010	50	b	01100010	98
3	00110011	51	c	01100011	99
4	00110100	52	d	01100100	100
5	00110101	53	e	01100101	101
6	00110110	54	f	01100110	102
7	00110111	55	g	01100111	103
8	00111000	56	h	01101000	104
9	00111001	57	i	01101001	105
:	00111010	58	j	01101010	106
;	00111011	59	k	01101011	107
<	00111100	60	l	01101100	108
=	00111101	61	m	01101101	109
>	00111110	62	n	01101110	110
?	00111111	63	o	01101111	111
@	01000000	64	p	01110000	112
A	01000001	65	q	01110001	113
B	01000010	66	r	01110010	114
C	01000011	67	s	01110011	115
D	01000100	68	t	01110100	116
E	01000101	69	u	01110101	117
F	01000110	70	v	01110110	118
G	01000111	71	w	01110111	119
H	01001000	72	x	01111000	120
I	01001001	73	y	01111001	121
J	01001010	74	z	01111010	122
K	01001011	75	{	01111011	123
L	01001100	76	:	01111100	124
M	01001101	77	]	01111101	125
N	01001110	78	~	01111110	126
O	01001111	79			

PRACTICE PROBLEMS

1. What is the value of the 8-bit binary quantity 10101000 if it is interpreted (a) as an unsigned integer, and (b) as a signed integer represented in sign/magnitude notation?
2. What does the unsigned decimal value 99 look like in binary using 8 bits?
3. What do the signed integers –300 and +254 look like in binary using 10 bits and signed magnitude integer representation?
4. Using 4 bits and two's complement representation, what is the binary representation of the following signed decimal values:

   a. +6
   b. –3

5. Perform the following 5-bit binary addition showing the carry bit that propagates to each column. Assume the numbers are unsigned binary quantities:

   ```
     01110
   + 01011
   ```

6. What does the 3-character string "X+Y" look like internally using the 8-bit ASCII code given in Figure 4.3? What does it look like in 16-bit UNICODE? (Go to *www.unicode.org* to find the specific code mappings for these three characters.)
7. Using 10 bits to represent the mantissa (sign/magnitude) and 6 bits for the exponent (also sign/magnitude), show the internal representation of the following two values:

   a. +0.25
   b. –32 1/16

8. Explain exactly what happens when you add a 1 to the following 5-bit, two's complement value: 01111

4.2.2 *Binary Representation of Sound and Images*

During the first 30 to 40 years of computing, the overwhelming majority of applications, such as word processing and spreadsheets, were text-based and limited to the manipulation of characters, words, and numbers. However, sound and images are now as important a form of representation as text and numbers because of the rapid growth of the Web, the popularity of digitally encoded music, the emergence of digital photography, and the almost universal availability of digital CD and DVD movies. Most of us, whether computer specialists or not, have probably had the experience of playing MP3 files or e-mailing vacation pictures to friends and family. In this section we take a brief look at how sounds and images are represented in computers, using the same binary numbering system that we have been discussing.

Sound is analog information, unlike the digital format used to represent text and numbers discussed in the previous section. In a **digital** representation, the values for a given object are drawn from a finite set, such as letters {A, B, C, . . . , Z} or a subset of integers {0, 1, 2, 3, . . . , MAX }. In an **analog** representation, objects can take on any value. For example, in the case of sound, a tone is a continuous sinusoidal waveform that varies in a regular periodic fashion over time, as shown in Figure 4.4. (*Note:* This diagram shows only a single tone. Complex sounds, such as symphonic music, are composed of multiple overlapping waveforms. However, the basic ideas are the same.)

The **amplitude** (height) of the wave is a measure of its loudness—the greater the amplitude the louder the sound. The **period** of the wave, designated as T, is the time it takes for the wave to make one complete cycle. The **frequency** f is the total number of cycles per unit time measured in cycles/second, also called **hertz**, and defined as $f = 1/T$. The frequency is a measure of the **pitch**, the highness or lowness of a sound. The higher the frequency the higher the perceived tone. A human ear can generally detect sounds in the range of 20 to 20,000 hertz.

To store a waveform (such as the one in Figure 4.4) in a computer, the analog signal must first be **digitized**, that is, converted to a digital representation. This can be done using a technique known as **sampling**. At fixed time intervals, the amplitude of the signal is measured and stored as an integer value. The wave is thus represented in the computer in digital form as a sequence of sampled numerical amplitudes. For example, Figure 4.5(a) shows the sampling of the waveform of Figure 4.4.

This signal can now be stored inside the computer as the series of signed integer values 3, 7, 7, 5, 0, −3, −6, −6, . . . , where each numerical value is encoded in binary using the techniques described in the previous section. From these stored digitized values the computer can recreate an approximation to the original analog wave. It would first generate an amplitude level of 3, then an amplitude level of 7, then an amplitude level of 7, and so on, as shown in Figure 4.5(b). These values would be sent to a sound-generating device, like stereo speakers, which would produce the actual sounds based on the numerical values received.

The accuracy with which the original sound can be reproduced is dependent on two key parameters—the sampling rate and the bit depth. The **sampling rate** measures how many times per second we sample the amplitude of the sound wave. Obviously, the more often we sample, the more accurate the reproduction. Note, for example, that the sampling shown in Figure 4.5(a) appears to have missed the peak value of the wave because the peak occurred between two sampling intervals. Furthermore, the more often we sample, the greater the range of frequencies that can be captured; if the frequency of a wave is greater than or

FIGURE 4.4
Example of Sound Represented as a Waveform

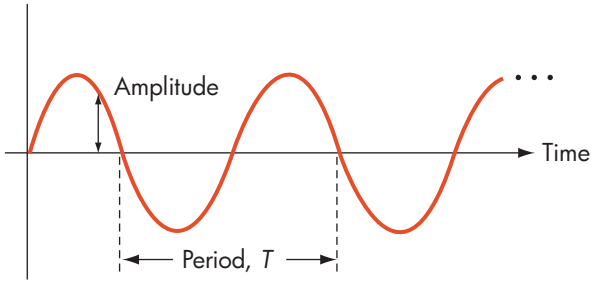
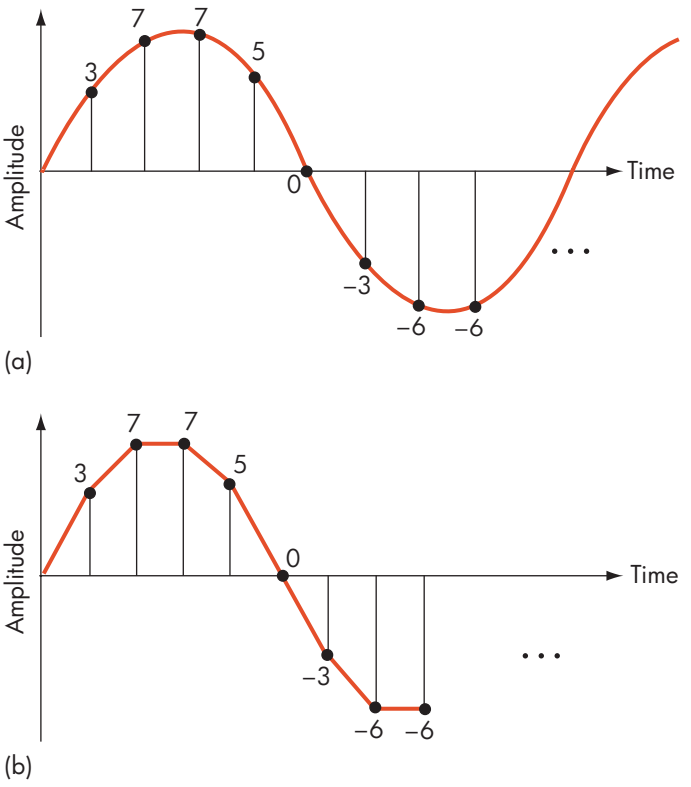


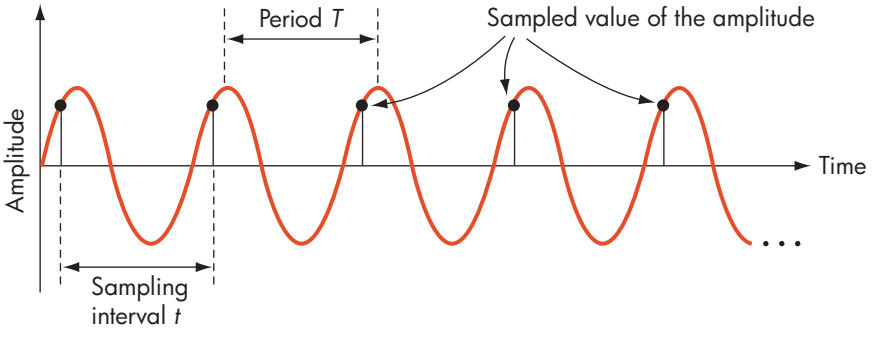
FIGURE 4.5

Digitization of an Analog Signal

(a) Sampling the Original Signal
(b) Re-creating the Signal from the Sampled Values



equal to the sampling rate, we may not sample any points whatsoever on an entire waveform. For example, look at the following sampling interval *t* which is exactly equal to the period *T* of the wave being measured:



This rate of sampling produces a constant amplitude value, totally distorting the original sound. In general, a sampling rate of R samples/second allows you to reproduce all frequencies up to about R/2 hertz. Because the human ear can normally detect sound up to about 20,000 hertz, a sampling rate of at least 40,000 samples per second is necessary to capture all audible frequencies.

The **bit depth** is the number of bits used to encode each sample. In the previous section you learned that ASCII is an 8-bit character code, allowing for 256 unique symbols. UNICODE uses 16 bits, allowing for more than 65,000 symbols and greatly increasing the number of symbols that can be represented. The same

trend can be seen in sound reproduction. Initially, 8 bits per sample was the standard, but the 256 levels of amplitude that could be represented turned out to be insufficient for the sophisticated high-end sound systems produced and marketed today. Most audio encoding schemes today use either 16 or 24 bits per sample level, allowing for either 65,000 or 16,000,000 distinct amplitude levels.

There are many audio-encoding formats in use today, including WAV, AU, Quicktime, and RealAudio. Probably the most popular and widely used digital audio format is **MP3**, an acronym for MPEG-1, Audio Level 3 Encoding. This is a digital audio encoding standard established by the Motion Picture Experts Group (MPEG), a committee of the International Organization for Standardization (ISO) of the United Nations. MP3 samples sound signals at the rate of 44,100 samples/second, using 16 bits per sample. This produces high-quality sound reproduction, which is why MP3 is the most widely used format for rock, opera, and classical music.

An image, such as a photograph, is also analog data but can also be stored using binary representation. An image is a continuous set of intensity and color values that can be digitized by sampling the analog information, just as is done for sound. The sampling process, often called **scanning**, consists of measuring the intensity values of distinct points located at regular intervals across the image's surface. These points are called **pixels**, for picture elements, and the more pixels used, the more accurate the encoding of the image. The average human eye cannot accurately discern components closer together than about 0.05–0.1 mm, so if the pixels, or dots, are sufficiently dense, they appear to the human eye as a single contiguous image. For example, a high-quality digital camera stores about 5–10 million pixels per photograph. For a 3 in. $\times$ 5 in. image, this is about 500,000 pixels/in.^{2}, or 700 pixels per linear inch. This means the individual pixels are separated by about 1/700th of an inch, or 0.03 mm—too close together to be individually visualized. Figure 4.6 enlarges a small section of a digitized photograph to better show how it is stored internally as a set of discrete picture elements.

One of the key questions we need to answer is how much information is stored for each pixel. Suppose we want to store a representation of a

FIGURE 4.6

Example of a Digitized Photograph

(a) Individual Pixels in the Photograph
(b) Photograph

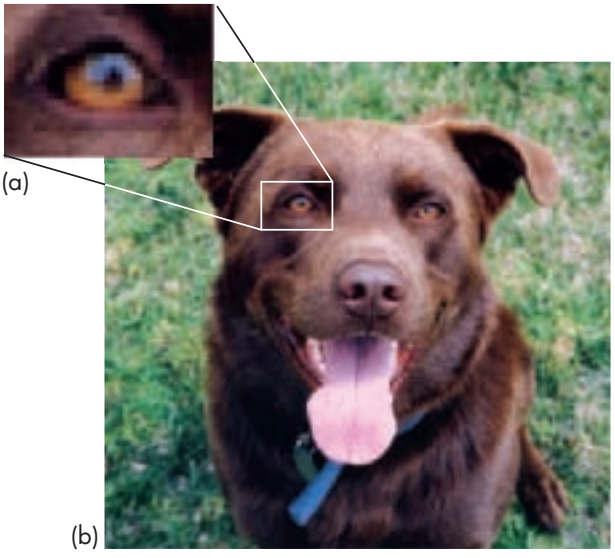


Photo by Maris Sidenstecker – Marine Biologist

black-and-white image. The easiest and most space-efficient approach is to mark each pixel as either white, stored as a binary 0, or black, stored as a binary 1. The only problem is that this produces a stark **black/white image**, with a highly sharp and unpleasant visual contrast. A much better way, though it takes more storage, is to represent black and white images using a **gray scale** of varying intensity. For example, if we use 3 bits per pixel, we can represent $2^3 = 8$ shades of intensity from level 0, pure white, to level 7, pure black. An example of this eight level gray scale is shown in Figure 4.7. If we wanted more detail than is shown there, we could use 8 bits per pixel, giving us $2^8 = 256$ distinct shades of gray.

We now can encode our image as a sequence of numerical pixel values, storing each row of pixels completely, from left to right, before moving down to store the next row. Each pixel is encoded as an unsigned binary value representing its gray scale intensity. This form of image representation is called **raster graphics**, and it is used by such well-known graphics standards as JPEG (Joint Photographer Experts Group), GIF (Graphics Interchange Format), and BMP (bitmap).

Today, most images are not black and white, but are in color. To digitize color images, we still measure the intensity value of the image at a discrete set of points, but we need to store more information about each pixel. The most common format for storing color images is the **RGB encoding scheme**, RGB being an acronym for Red-Green-Blue. This technique describes a specific color by capturing the individual contribution to a pixel's color of each of the three colors, red, green, and blue. It uses one **byte**, or 8 bits, for each color, allowing us to represent an intensity range of 0 to 255. The value 0 means that there is no contribution from this color, whereas the value 255 means a full contribution of this color.

For example, the color magenta is an equal mix of pure red and blue, which would be RGB encoded as (255, 0, 255):

Red	*Green*	*Blue*
255	0	255

The color "hot pink" is produced by setting the three RGB values to

Red	*Green*	*Blue*
255	105	180

and "harvest gold" is rendered as

Red	*Green*	*Blue*
218	165	32

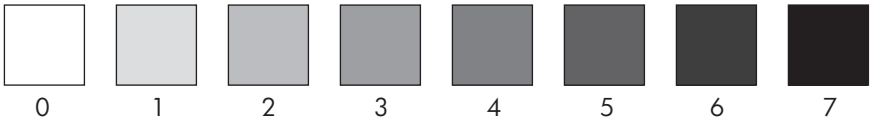


FIGURE 4.7
An Eight-Level Gray Scale

Using three bytes of information per pixel—24 bits—allows us to represent 2^{24} distinct colors, about 16.7 million. This 24-bit color-encoding scheme is often referred to as **True Color**, and it provides an enormous range of shades and an extremely accurate color image reproduction. That is why it is the encoding scheme used in the JPEG Color Imaging format. However, representing 16+ million colors requires a huge amount of memory space, and some image representation techniques reduce that value by using what is called a **color palette**. While theoretically supporting 16+ million different colors, they only allow you to use 256 (or some other small number) at any one time, just as a painter may have a lot of colors in his or her studio but puts only a few on the palette at a time. With a palette size of 256, we can encode each pixel using only 8 bits rather than 24, because $2^8 = 256$, thus reducing storage space demands by almost 67%. Each of these 256 values does not represent an explicit RGB color value but rather an index into a palette, or a color table. This index specifies which color on the palette is to be used to draw this pixel. This is the technique used, for example, in the Graphics Interchange Format (GIF), which uses a palette that can hold as few as 2 colors or as many as 256.

Sound and image data typically require huge amounts of storage, far more than is required for the numbers and text discussed in Section 4.2.1. For example, a 300-page novel contains about 100,000 words. Each word has on average about 5 characters and, as discussed in the previous section, each character can be encoded into the ASCII code set using 8 bits. Thus, the total number of bits needed to represent this book is roughly

$$100{,}000 \text{ words} \times 5 \text{ char/word} \times 8 \text{ bits/char} = \textit{4 million bits}$$

By comparison, 1 minute of sound recording encoded using the MP3 standard, which samples 44,100 times per second using a bit depth of 16 bits per sample, requires

$$44{,}100 \text{ samples/sec} \times 16 \text{ bits/sample} \times 60 \text{ sec/minute} = \textit{42 million bits}$$

It takes 10 times as much space to store the information in 1 minute of music as it does to store an entire 300-page book! Similarly, to store a single photograph taken using a digital camera with 5 million pixels using 24-bit TrueColor raster graphics requires:

$$5{,}000{,}000 \text{ pixels/photograph} \times 24 \text{ bits/pixel} = \textit{120 million bits}$$

A single photograph could require as much as 30 times more storage than an entire novel.

As these examples clearly show, the storage of analog information, such as sound, images, voice, and video, is enormously space-intensive, and an important area of computer science research—**data compression**—is directed at addressing just this issue. Data compression algorithms attempt to represent information in ways that preserve accuracy while using significantly less space.

For example, a simple compression technique that can be used on almost any form of data is **run-length encoding**. This method replaces a sequence of identical values $v_1, v_2, \ldots, v_n$ by a pair of values (v, n) which indicates that the value v is replicated n times. If both v and n require 1 byte of storage, then we have reduced the total number of bytes required to store this

sequence from *n* down to 2. Using this method, we could encode the following 5 × 3 image of the letter *E,* where 0 = white, 255 = black:

```
255     255     255
255     0       0
255     255     255
255     0       0
255     255     255
```

like this:

(255, 4) (0, 2) (255, 4) (0, 2) (255, 3)

Run-length encoding reduces the number of bytes needed to store this image from 15, using the raster graphics representation, to the 10 bytes shown above. Compression schemes are usually evaluated by their **compression ratio**, which measures how much they reduce the storage requirements of the data:

$$\text{compression ration} = \frac{\text{size of the uncompressed data}}{\text{size of the compressed data}}$$

For the example shown above, this ratio is

$$\text{ratio} = 15/10 = 1.5$$

meaning the scheme reduces the amount of space needed to store the image by 33%. Applied to a larger image, this might mean that a 4-million-bit representation could be reduced to about 2.7 million bits, a significant savings.

Another popular compression technique is **variable length code sets**, which are often used to compress text but can also be used with other forms of data. In Section 4.2.1 we showed that textual symbols, such as 'A', 'z', and '#' are represented internally by a code mapping that uses exactly the same number of bits for every symbol, either 8 (ASCII) or 16 (UNICODE). That is a wasteful approach as some symbols occur much more frequently than others. (For example, in English the letters *E* and *A* are much more common than *J, Q, X,* and *Z.*) If the codes representing commonly used symbols were shorter than the codes representing the less common symbols, this could result in a significant saving of space.

Assume that we want to encode the Hawaiian alphabet, which only contains the 5 vowels *A, E, I, O,* and *U,* and the 7 consonants *H, K, L, M, N, P,* and *W*. If we were to store these characters using a fixed length code set, we would need at least 4 bits/symbol, because $2^4 = 16$. Figure 4.8(a) shows one possible encoding of these 12 letters using a fixed length, 4-bit encoding. However, if we know that *A* and *I* are the most commonly used letters in the Hawaiian alphabet, with *H* and *W* next, we could represent *A* and *I* using two bits, *H* and *W* using 3 bits, and the remaining letters using either 4, 5, 6, or 7 bits, depending on their frequency. However, we must be sure that if the 2-bit sequence s_1s_2 is used to represent an *A,* for example, then no other symbol representation can start with the same 2-bit sequence. Otherwise, if we saw the sequence s_1s_2 we would not know if it was an *A* or the beginning of another character.

One possible variable-length encoding for the Hawaiian alphabet is shown in Figure 4.8(b).

FIGURE 4.8

Using Variable Length Code Sets

(a) Fixed Length
(b) Variable Length

Letter	4-bit Encoding	Variable Length Encoding
A	0000	00
I	0001	10
H	0010	010
W	0011	110
E	0100	0110
O	0101	0111
M	0110	11100
K	0111	11101
U	1000	11110
N	1001	111110
P	1010	1111110
L	1011	1111111
	(a)	(b)

Representing the 6-character word HAWAII using the fixed length 4-bit encoding scheme of Figure 4.8(a) requires $6 \times 4 = 24$ bits. Representing it with the variable length encoding shown in Figure 4.8(b) produces the following:

H	A	W	A	I	I
010	00	110	00	10	10

This is a total of 14 bits, producing a compression ratio of 24/14 = 1.71, a reduction in storage demands of about 42%.

These two techniques are examples of what are called **lossless compression** schemes. This means that no information is lost in the compression, and it is possible to exactly reproduce the original data. **Lossy compression** schemes compress data in a way that does not guarantee that all of the information in the original data can be fully and completely recreated. They trade a possible loss of accuracy for a higher compression ratio because the small inaccuracies in sounds or images are often undetectable to the human ear or eye. Many of the compression schemes in

PRACTICE PROBLEMS

1. Using MP3, how many bits are required to store a 3-minute song in uncompressed format? If the information is compressed with a ratio of 4:1, how many bits are required?
2. How many bits are needed to store a single uncompressed RGB image from a 2.1 megapixel digital camera? How many bytes of memory is this?
3. If we want the image in Exercise 2 to fit into 1 megabyte of memory, what compression ratio is needed? If we want it to fit into 256 kilobytes of memory, what compression ratio is needed?
4. How much space is saved by representing the Hawaiian word ALOHA in the variable length code of Figure 4.8(b) as compared to the fixed length representation of Figure 4.8(a)? What is the compression ratio?

widespread use today, including MP3 and JPEG, use lossy techniques, which permit significantly greater compression ratios than would otherwise be possible. Using lossy JPEG, for example, it is possible to achieve compression ratios of 10:1, 20:1, or more, depending on how much loss of detail we are willing to tolerate. This compares with the values of 1.5 and 1.7 in the above described lossless schemes. Using these lossy compression schemes, that 120-megabit, high-resolution image mentioned earlier could be reduced to only 6 or 12 megabits, certainly a much more manageable value. Data compression schemes are an essential component in allowing us to represent multimedia information in a concise and manageable way.

4.2.3 *The Reliability of Binary Representation*

At this point you might be wondering: Why are we bothering to use binary? Because we use a decimal numerical system for every day tasks, wouldn't it be more convenient to use a base-10 representation for both the external and the internal representation of information? Then there would be no need to go through the time-consuming conversions diagrammed in Figure 4.1, or to learn the binary representation techniques discussed in the previous two sections.

As we stated in the boxed text entitled, "The Not So Basic Base," there is absolutely no theoretical reason why one could not build a "decimal" computer or, indeed, a computer that stored numbers using base 3 (**ternary**), base 8 (**octal**), or base 16 (**hexadecimal**). The techniques described in the previous two sections apply to information represented in *any* base of a positional numbering system, including base 10.

Computers use binary representation not for any theoretical reasons but for reasons of **reliability**. As we shall see shortly, computers store information using electronic devices, and the internal representation of information must be implemented in terms of electronic quantities such as currents and voltage levels.

Building a base-10 "decimal computer" requires finding a device with 10 distinct and stable energy states that can be used to represent the 10 unique digits (0, 1, . . . , 9) of the decimal system. For example, assume there exists a device that can store electrical charges in the range 0 to +45 volts. We could use it to build a decimal computer by letting certain voltage levels correspond to specific decimal digits:

Voltage Level	*Corresponds to this Decimal Digit*
+0	0
+5	1
+10	2
+15	3
+20	4
+25	5
+30	6
+35	7
+40	8
+45	9

Storing the 2-digit decimal number 28 requires two of these devices, one for each of the digits in the number. The first device would be set to +10 volts to represent the digit 2, and the second would be set to +40 volts to represent the digit 8.

Although this is theoretically feasible, it is certainly not recommended. As electrical devices age they become unreliable, and they may *drift,* or change their energy state, over time. What if the device representing the value 8 (the one set to +40 volts) lost 6% of its voltage (not a huge amount for an old, well-used piece of equipment)? The voltage would drop from +40 volts to about +37.5 volts. The question is whether the value +37.5 represents the digit 7 (+35) or the digit 8 (+40). It is impossible to say. If that same device lost another 6% of its voltage, it would drop from +37.5 volts to about +35 volts. Our 8 has now become a 7, and the original value of 28 has unexpectedly changed to 27. Building a reliable decimal machine would be an engineering nightmare.

The problem with a base-10 representation is that it needs to store 10 unique symbols, and therefore it needs devices that have 10 stable states. Such devices are extremely rare. Electrical systems tend to operate best in a **bistable environment**, in which there are only two (rather than 10) stable states separated by a huge energy barrier. Examples of these bistable states include

- full on/full off
- fully charged/fully discharged
- charged positively/charged negatively
- magnetized/nonmagnetized
- magnetized clockwise/magnetized counterclockwise

In the binary numbering system there are only two symbols (0 and 1), so we can let one of the two stable states of our bistable device represent a 0 and the other a 1. This is a much more reliable way to represent information inside a computer.

For example, if we use binary rather than decimal to store data in our hypothetical electronic device that stores voltages in the range 0 to +45 volts, the representational scheme becomes much simpler:

0 volts = 0 (full off)
+45 volts = 1 (full on)

Now a 6% or even a 12% drift doesn't affect the interpretation of the value being represented. In fact, it takes an almost 50% change in voltage level to create a problem in interpreting a stored value. The use of binary for the internal representation of data significantly increases the inherent reliability of a computer. This single advantage is worth all the time it takes to convert from decimal to binary for internal storage and from binary to decimal for the external display of results.

4.2.4 *Binary Storage Devices*

As you learned in the previous section, binary computers can be built out of any bistable device. This idea can be expressed more formally by saying that it

is possible to construct a binary computer and its internal components using any hardware device that meets the following four criteria:

1. The device has two stable energy states (one for a 0, one for a 1).
2. These two states are separated by a large energy barrier (so that a 0 does not accidentally become a 1, or vice versa).
3. It is possible to sense which state the device is in (to see whether it is storing a 0 or a 1) without permanently destroying the stored value.
4. It is possible to switch the state from a 0 to a 1, or vice versa, by applying a sufficient amount of energy.

There are many devices that meet these conditions, including some surprising ones such as a light switch. A typical light switch has two stable states (ON and OFF). These two states are separated by a large energy barrier so that a switch that is in one state will not accidentally change to the other. We can determine what state the switch is in by looking to see whether the label says ON or OFF (or just by looking at the light), and we can change the state of the switch by applying a sufficient amount of energy via our fingertips. Thus it would be possible to build a reliable (albeit very slow and bulky) binary computing device out of ordinary light switches and fingertips!

As you might imagine, computer systems are not built from light switches, but they have been built using a wide range of devices. This section describes two of these devices.

Magnetic cores were used to construct computer memories for about 20 years. From roughly 1955 to 1975, this was by far the most popular storage technology—even today, the memory unit of a computer is sometimes referred to as **core memory** even though it has been decades since magnetic cores have been used.

A **core** is a small, magnetizable, iron oxide-coated "doughnut," about 1/50 of an inch in inner diameter, with wires strung through its center hole. The two states used to represent the binary values 0 and 1 are based on the *direction* of the magnetic field of the core. When electric current is sent through the wire in one specific direction, say left to right, the core is magnetized in a counterclockwise direction.[1] This state could represent the binary value 0. Current sent in the opposite direction produces a clockwise magnetic field that could represent the binary value 1. These scenarios are diagrammed in Figure 4.9. Because magnetic fields do not change much over time, these two states are highly stable, and they form the basis for the construction of memory devices that store binary numbers.

In the early 1970s, core memories were replaced by smaller, cheaper technologies that required less power and were easier to manufacture. One-fiftieth of an inch in diameter and a few grams of weight may not seem like much, but it can produce a bulky and unworkable structure when memory units must contain millions or billions of bits. For example, a typical core memory from the 1950s or 1960s had about 500 cores/in^2. The memory in a modern computer typically has at least 1 GB (1 **gigabyte** = 1 billion bytes), which is more than 8 billion bits. At the bit density of core memory, the memory unit would

1 The righthand rule of physics says that if the thumb of your right hand is pointing in the direction of the electric current, then the fingers will be curled in the direction of the magnetic field.

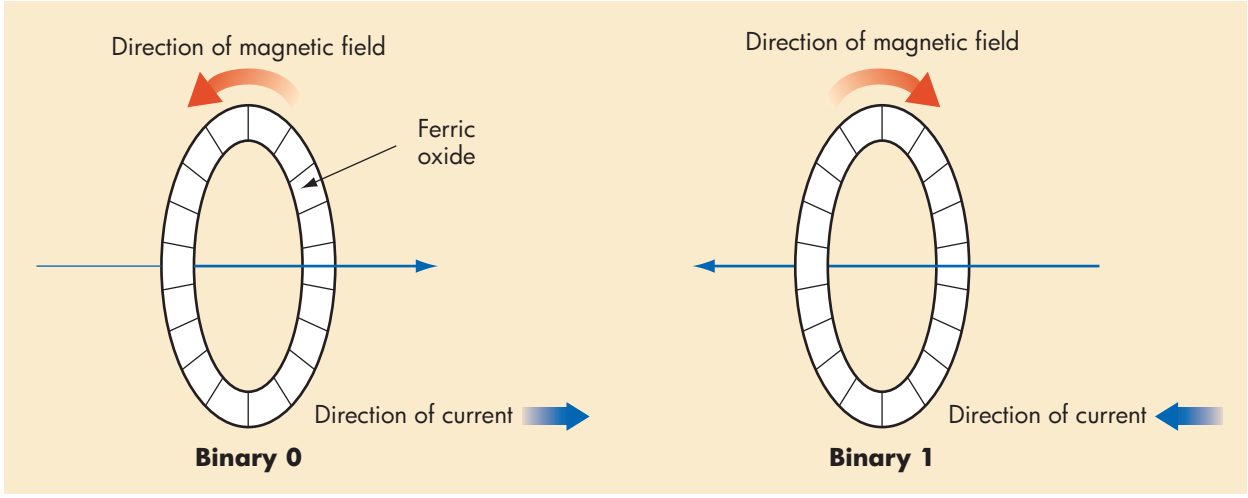


FIGURE 4.9

Using Magnetic Cores to Represent Binary Values

need about 16 million in^2, which is a square about 4,000 inches, or 330 feet, on a side. Built from cores, the memory unit would stand more than 30 stories high!

Today, the elementary building block for all modern computer systems is no longer the core but the transistor. A **transistor** is much like the light switch mentioned earlier. It can be in an OFF state, which does not allow electricity to flow, or in an ON state, in which electricity can pass unimpeded. However, unlike the light switch, a transistor is a solid-state device that has no mechanical or moving parts. The switching of a transistor from the OFF to the ON state, and vice versa, is done electronically rather than mechanically. This allows the transistor to be fast as well as extremely small. A typical transistor can switch states in a billionth of a second, and at current technology levels, 100 million to 1 billion transistors can fit into a space only 1 cm^2. Furthermore, hardware technology is changing so rapidly that both these numbers may be out of date by the time you read these words.

Transistors are constructed from special materials called **semiconductors**, such as silicon and gallium arsenide. A large number of transistors, as well as the electrical conducting paths that connect them, can be printed photographically on a wafer of silicon to produce a device known as an **integrated circuit** or, more commonly, a **chip.** The chip is mounted on a **circuit board**, which interconnects all the different chips (e.g., memory, processor, communications) needed to run a computer system. This circuit board is then plugged into the computer using a set of connectors located on the end of the board. The relationships among transistors, chips, and circuit boards is diagrammed in Figure 4.10. The use of photographic rather than mechanical production techniques has numerous advantages. Because light can be focused very sharply, these integrated circuits can be manufactured in very high densities—high numbers of transistors per square centimeter—and with a very high degree of accuracy. The more transistors that can be packed into a fixed amount of space, the greater the processing power of the computer and the greater the amount of information that can be stored in memory.

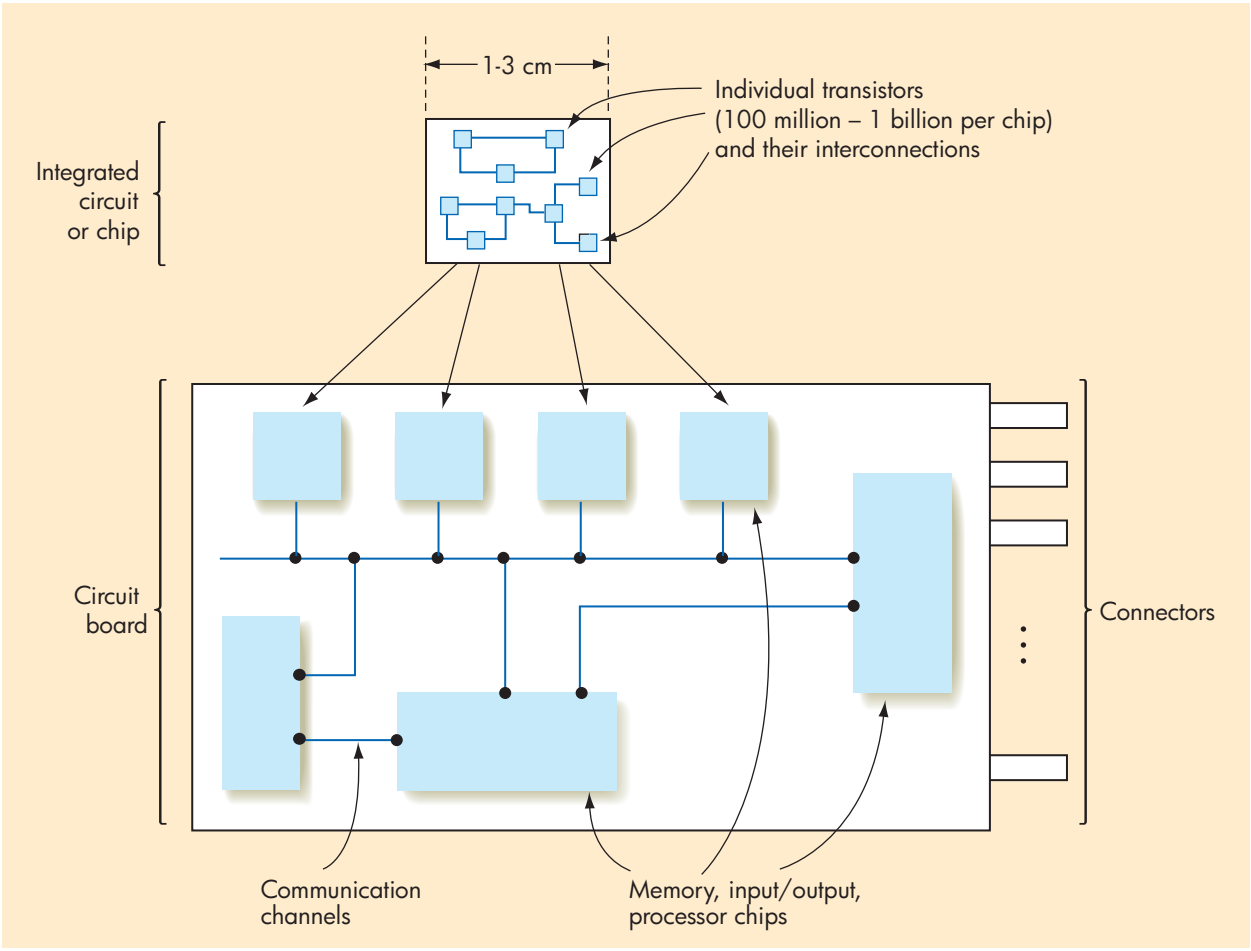


FIGURE 4.10

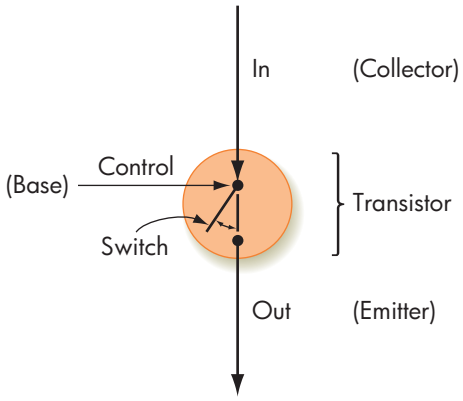
Relationships Among Transistors, Chips, and Circuit Boards

Another advantage of photographic production techniques is that it is possible to make a standard template, called a **mask**, which describes the circuit. This mask can be used to produce a virtually unlimited number of copies of that chip, much as a photographic negative can be used to produce an unlimited number of prints.

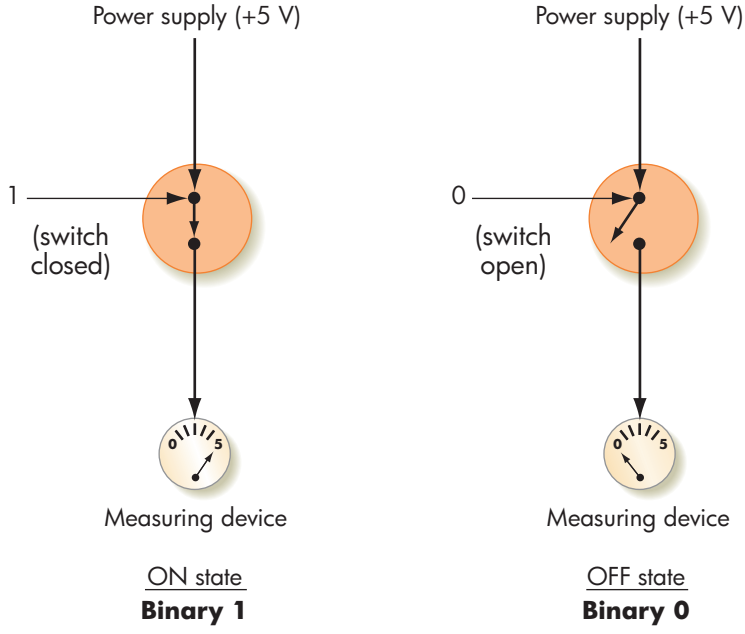
Together, these characteristics can result in very small and very inexpensive high-speed circuits. Whereas the first computers of the early 1940s (as seen in Figure 1.6) filled huge rooms and cost millions of dollars, the processor inside a modern workstation contains hundreds of millions of transistors on a tiny chip just a few centimeters square, is thousands of times more powerful than those early machines, and costs just a few hundred dollars.

The theoretical concepts underlying the physical behavior of semiconductors and transistors, as well as the details of chip manufacture, are well beyond the scope of this book. They are usually discussed in courses in physics or electrical engineering. Instead, we will examine a transistor in terms of the simplified model shown in Figure 4.11 and then use this model to explain its behavior. (Here is another example of the importance of abstraction in computer science.)

FIGURE 4.11
Simplified Model of a Transistor



In the model shown in Figure 4.11, each transistor contains three lines—two input lines and one output line, with each line either in the 1-state, with a high positive voltage, or in the 0-state, with a voltage close to 0. The first input line, called the **control** or the **base**, is used to open or close the switch inside the transistor. If we set the input on the control line to a 1 by applying a sufficiently high positive voltage, the switch closes and the transistor enters the ON state. In this state, current coming from the **In** line, called the **Collector**, can flow directly to the **Out** line, called the **Emitter**, and the associated voltage can be detected by a measuring device. This ON state could be used to represent the binary value 1. If instead we set the input value of the control line to a 0 by applying a voltage close to zero, the switch opens, and the transistor enters the OFF state. In this state the flow of current through the transistor is blocked and no voltage is detected on the Out line. The OFF state could be used to represent the binary value 0. This is diagrammed as follows:



This type of solid-state switching device forms the basis for the construction of virtually all computers built today, and it is the fundamental building block for all high-level components described in the upcoming chapters. Remember, however, that there is no theoretical reason why we must use transistors as our "elementary particles" when designing computer systems. Just as cores were replaced by transistors, transistors may ultimately be replaced by some newer (perhaps molecular or biological) technology that is faster, smaller, and cheaper. The only requirements for our building blocks are those given in the beginning of this section—that they be able to represent reliably the two binary values 0 and 1.

Moore's Law and the Limits of Chip Design

Since the development of the first integrated circuits in the 1950s, the number of transistors on a circuit board has been doubling roughly every 24 months. This observation was first reported in a 1965 paper by Gordon E. Moore, the co-founder of Intel, and is now universally referred to as "Moore's Law." This doubling has continued unabated for the last 40 years, and represents a rate of improvement unequaled in any other technical field. More transistors on a chip means more speed and more power, and is the reason for the enormous increase in performance (and decrease in size) of computers in the last 40–50 years. The following table details this growth from 1971, when chips held just a few thousand transistors, to today's microprocessors that hold well over a billion.

Processor	Transistor Count	Date
Intel 404	2,300	1971
Intel 8080	4,500	1974
Intel 8088	29,000	1979
Intel 80286	134,000	1982
Intel 80386	275,000	1985
Intel 80486	1,200,000	1989
Pentium	3,100,000	1993
Pentium II	7,500,000	1997
Pentium 4	42,000,000	2000
Itanium 2	220,000,000	2003
Dual-Core Itanium 2	1,400,000,000	2006
Quad-Core Itanium Tukwila	2,000,000,000	2008

It is impossible to maintain this type of exponential growth for an indefinitely extended period of time, and industry analysts have been predicting the demise of Moore's Law for the last 10–15 years. However, the ongoing development of new materials and new manufacturing technologies has allowed the industry to continue this phenomenal rate of improvement. But there is a physical limit looming on the horizon that will be the most difficult hurdle yet. As more and more transistors are packed onto a single chip, distances between them get smaller and smaller, and experts estimate that in about 10–20 years inter-transistor distances will approach the space between individual atoms. For example, transistors on today's chips are separated by 50–100 nanometers (1 nanometer = 10^{-9} meter), only about 500–1000 times greater than the diameter of a single atom of silicon, which is about 10^{-10} meters. In a few generations, these atomic distances will be reached, and a totally new approach to computer design will be required, perhaps one based on the emerging fields of nanotechnology and quantum computing.

4.3.1 *Boolean Logic*

The construction of computer circuits is based on the branch of mathematics and symbolic logic called **Boolean logic**. This area of mathematics deals with rules for manipulating the two logical values **true** and **false**. It is easy to see the relationship between Boolean logic and computer design: the truth value *true* could represent the binary value 1, and the truth value *false* could represent the binary value 0. Thus anything stored internally as a sequence of binary digits (which, as we saw in earlier sections, is everything stored inside a computer) can also be viewed as a sequence of the logical values true and false, and these values can be manipulated by the operations of Boolean logic.

Let us define a **Boolean expression** as any expression that evaluates to either true or false. For example, the expression ($x = 1$) is a Boolean expression because it is true if x is 1, and it is false if x has any other value. Similarly, both ($a \neq b$) and ($c > 5.23$) are Boolean expressions.

In "traditional" mathematics (the mathematics of real numbers), the operations used to construct arithmetic expressions are +, −, ×, ÷, and a^b, which map real numbers into real numbers. In Boolean logic, the operations used to construct Boolean expressions are AND, OR, and NOT, and they map a set of (true, false) values into a single (true, false) result.

The rule for performing the AND operation is as follows: If a and b are Boolean expressions, then the value of the expression (a AND b), also written as ($a \cdot b$), is *true* if and only if both a and b have the value *true*; otherwise, the expression (a AND b) has the value *false*. Informally, this rule says that the AND operation produces the value *true* if and only if both of its components are true. This idea can be expressed using a structure called a **truth table**, shown in Figure 4.12.

The two columns labeled Inputs in the truth table of Figure 4.12 list the four possible combinations of true/false values of a and b. The column labeled Output specifies the value of the expression (a AND b) for the corresponding values of a and b.

To illustrate the AND operation, imagine that we want to check whether a test score S is in the range 90 to 100 inclusive. We wish to develop a Boolean expression that is true if the score is in the desired range and false otherwise. We cannot do this with a single comparison. If we test only that ($S \geq 90$), then a score of 105, which is greater than or equal to 90, will produce the result *true,* even though it is out of range. Similarly, if we test only that ($S \leq 100$), then a score of 85, which is less than or equal to 100, will also produce a *true,* even though it too is not in the range 90 to 100.

FIGURE 4.12

Truth Table for the AND Operation

Inputs		Output
a	b	a AND b (also written a · b)
False	False	False
False	True	False
True	False	False
True	True	True

Instead, we need to determine whether the score S is greater than or equal to 90 *and* whether it is less than or equal to 100. Only if both conditions are true can we say that S is in the desired range. We can express this idea using the following Boolean expression:

$$(S \geq 90) \text{ AND } (S \leq 100)$$

Each of the two expressions in parentheses can be either true or false depending on the value of S. However, only if both conditions are true does the expression evaluate to *true*. For example, a score of $S = 70$ causes the first expression to be false (70 is not greater than or equal to 90), whereas the second expression is true (70 is less than or equal to 100). The truth table in Figure 4.12 shows that the result of evaluating (*false* AND *true*) is *false*. Thus, the overall expression is false, indicating (as expected) that 70 is not in the range 90 to 100.

The second Boolean operation is OR. The rule for performing the OR operation is as follows: If *a* and *b* are Boolean expressions, then the value of the Boolean expression (*a* OR *b*), also written as (*a* + *b*), is *true* if *a* is *true,* if *b* is *true,* or if both are *true*. Otherwise, (*a* OR *b*) has the value *false*. The truth table for OR is shown in Figure 4.13.

To see the OR operation at work, imagine that a variable called *major* specifies a student's college major. If we want to know whether a student is majoring in either math or computer science, we cannot accomplish this with a single comparison. The test (*major* = math) omits computer science majors, whereas the test (*major* = computer science) leaves out the mathematicians. Instead, we need to determine whether the student is majoring in *either* math or computer science (or perhaps in both). This can be expressed as follows:

$$(major = \text{math}) \text{ OR } (major = \text{computer science})$$

If the student is majoring in either one or both of the two disciplines, then one or both of the two terms in the expression is true. Referring to the truth table in Figure 4.13, we see that (*true* OR *false*), (*false* OR *true*), and (*true* OR *true*) all produce the value *true,* which indicates that the student is majoring in at least one of these two fields. However, if the student is majoring in English, both conditions are false. As Figure 4.13 illustrates, the value of the expression (*false* OR *false*) is *false,* meaning that the student is not majoring in either math or computer science.

The final Boolean operator that we examine here is NOT. Unlike AND and OR, which require two operands and are therefore called **binary operators**, NOT requires only one operand and is called a **unary operator**, like the square root operation in arithmetic. The rule for evaluating the NOT operation is as follows: If *a* is a Boolean expression, then the value of the expression (NOT *a*),

FIGURE 4.13

Truth Table for the OR Operation

Inputs		Output
a	b	a OR b (also written a + b)
False	False	False
False	True	True
True	False	True
True	True	True

also written as $\bar{a}$, is *true* if *a* has the value *false,* and it is *false* if *a* has the value *true*. The truth table for NOT is shown in Figure 4.14.

Informally, we say that the NOT operation reverses, or **complements**, the value of a Boolean expression, making it true if currently false, and vice versa. For example, the expression (GPA > 3.5) is true if your grade point average is greater then 3.5, and the expression NOT (GPA > 3.5) is true only under the reverse conditions, that is when your grade point average is less than or equal to 3.5.

AND, OR, and NOT are the three operations of Boolean logic that we use in this chapter. (*Note:* We will briefly mention other Boolean operators such as XOR, NOR, and NAND.) Why have we introduced these Boolean operations in the first place? The previous section discussed hardware concepts such as energy states, electrical currents, transistors, and integrated circuits. Now it appears that we have changed directions and are discussing highly abstract ideas drawn from the discipline of symbolic logic. However, as we hinted earlier and will see in detail in the next section, there is a very close relationship between the hardware concepts of Section 4.2.4 and the operations of Boolean logic. In fact, the fundamental building blocks of a modern computer system (the objects with which engineers actually design) are not the transistors introduced in Section 4.2.4 but the gates that implement the Boolean operations AND, OR, and NOT. Surprisingly, it is the rules of logic—a discipline developed by the Greeks 2,300 years ago and expanded by George Boole (see the box feature on page 162) 150 years ago—that provide the theoretical foundation for constructing modern computer hardware.

PRACTICE PROBLEMS

1. Assuming that $x = 1$ and $y = 2$, determine the value of each of the following Boolean expressions:
   - **a.** $(x = 1)$ AND $(y = 3)$
   - **b.** $(x < y)$ OR $(x > 1)$
   - **c.** NOT $[(x = 1)$ AND $(y = 2)]$
2. What is the value of the following Boolean expression:

   $(x = 5)$ AND $(y = 11)$ OR $([x + y] = z)$

   if $x = 5$, $y = 10$, and $z = 15$? Did you have to make some assumptions when you evaluated this expression?
3. Write a Boolean expression that is true if and only if *x* and *y* are both in the range 0 to 100 but *x* is not equal to *y*.
4. Write a Boolean expression that is true if and only if the variable *score* is *not* in the range 200–800, inclusive.

FIGURE 4.14
Truth Table for the NOT Operation

INPUT a	OUTPUT NOT a (ALSO WRITTEN $\bar{a}$)
False	True
True	False

4.3.2 *Gates*

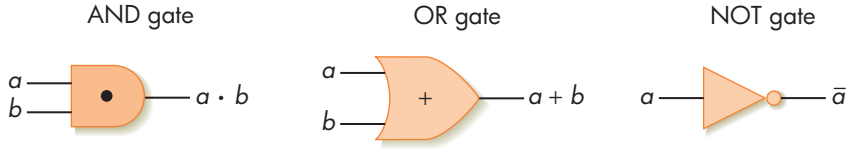
A **gate** is an electronic device that operates on a collection of binary inputs to produce a binary output. That is, it transforms a set of (0,1) input values into a single (0,1) output value according to a specific transformation rule. Although gates can implement a wide range of different transformation rules, the ones we are concerned with in this section are those that implement the Boolean operations AND, OR, and NOT introduced in the previous section. As shown in Figure 4.15, these gates can be represented symbolically, along with the truth tables that define their transformation rules.

Comparing Figures 4.12-4.14 with Figure 4.15 shows that if the value 1 is equivalent to *true* and the value 0 is equivalent to *false,* then these three electronic gates directly implement the corresponding Boolean operation. For example, an AND gate has its output line set to 1 (set to some level of voltage that represents a binary 1) if and only if both of its inputs are 1. Otherwise, the output line is set to 0 (set to some level of voltage that represents a binary 0). This is functionally identical to the rule that says the result of (*a* AND *b*) is *true* if and only if both *a* and *b* are *true*; otherwise, (*a* AND *b*) is *false*. Similar arguments hold for the OR and NOT.

A NOT gate can be constructed from a single transistor, as shown in Figure 4.16, in which the collector is connected to the power supply (logical-1) and the emitter is connected to the ground (logical-0). If the input to the transistor is set to 1, then the transistor is in the ON state, and it passes current through to the ground. In this case the output voltage of the gate is 0. However, if the input is set to 0, the transistor is in the OFF state, and it blocks passage of current to the ground. Instead, the current is transmitted to the output line, producing an output of 1. Thus, the value appearing on the output line of Figure 4.16 is the complement—the NOT—of the value appearing on the collector, or input line.

To construct an AND gate, we connect two transistors **in series**, as shown in Figure 4.17(a), with the collector line of transistor 1 connected to the power supply (logical-1) and the emitter line of transistor 2 connected to ground (logical-0). If both input lines, called Input-1 and Input-2 in Figure 4.17(a), are set to 1, then both transistors are in the ON state, and the output will be connected to ground, resulting in a value of 0 on the output line. If either (or both) Input-1 or Input-2 is 0, then the corresponding transistor is in the OFF state and does not allow current to pass, resulting in a 1 on the output line. Thus, the output of the gate in Figure 4.17(a) is a 0 if and only if both inputs are a 1; otherwise, it is a 1. This is the exact *opposite* of the definition of AND, and Figure 4.17(a) represents a gate called

FIGURE 4.15
The Three Basic Gates and Their Symbols

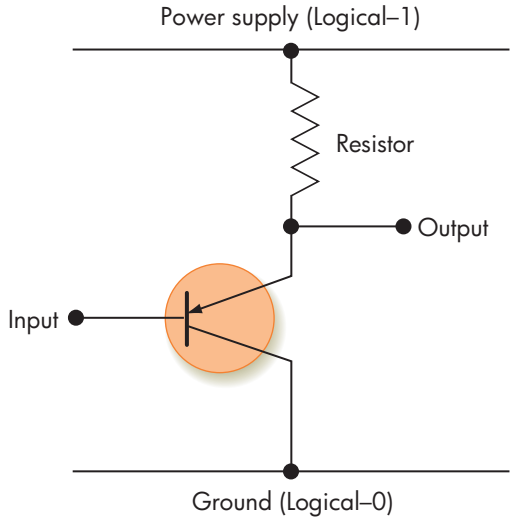


a	b	a · b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a + b
0	0	0
0	1	1
1	0	1
1	1	1

a	ā
0	1
1	0

FIGURE 4.16
Construction of a NOT Gate



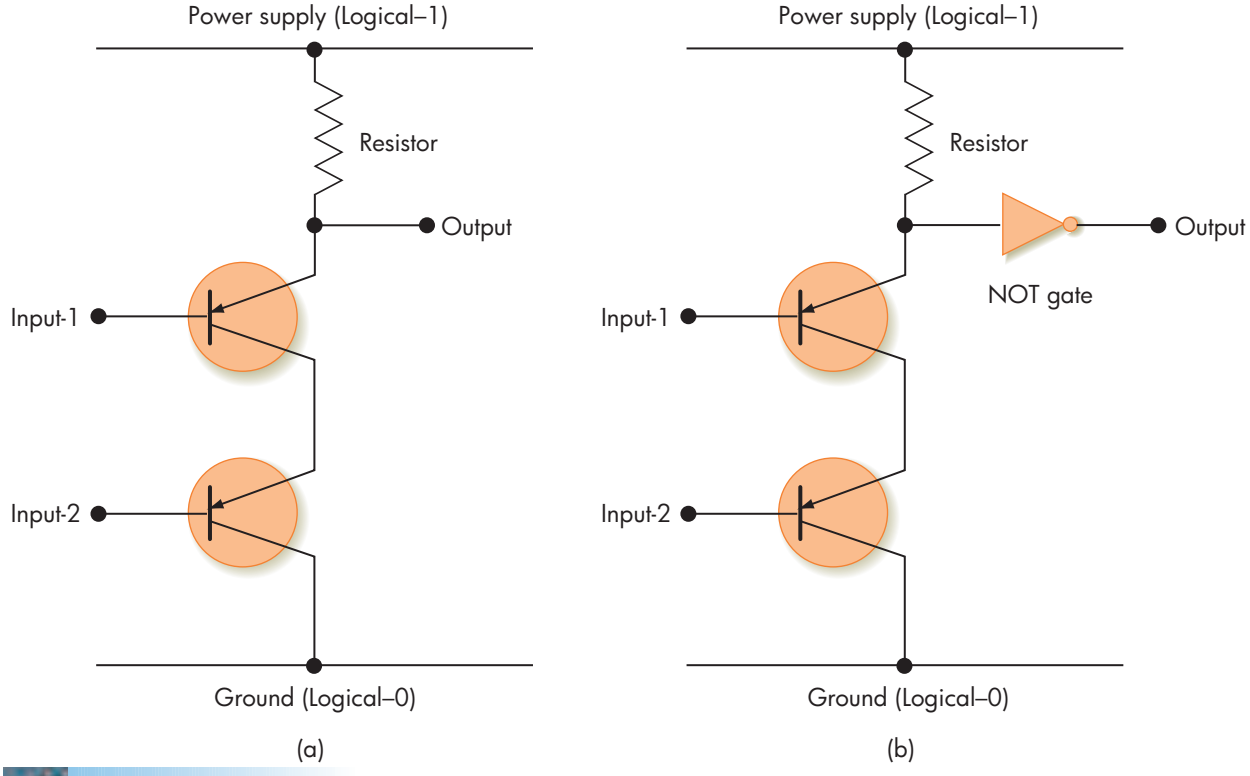


FIGURE 4.17
Construction of NAND and AND Gates

(a) A Two-transistor NAND Gate
(b) A Three-transistor AND Gate

NAND, an acronym for *NOT AND*. It produces the complement of the AND operation, and it is an important and widely used gate in hardware design.

If, however, we want to build an AND gate, then all we have to do is add a NOT gate (of the type shown in Figure 4.16) to the output line. This complements the NAND output and produces the AND truth table of Figure 4.12. This gate is shown in Figure 4.17(b). Note that the NAND of Figure 4.17(a) requires two transistors, whereas the AND of Figure 4.17(b) requires three. This is one reason why NAND gates are widely used to build computer circuits.

To construct an OR gate, we again start with two transistors. However, this time they are connected **in parallel** rather than in series, as shown in Figure 4.18(a).

In Figure 4.18(a) if either, or both, of the lines Input-1 and Input-2 are set to 1, then the corresponding transistor is in the ON state, and the output is connected to the ground, producing an output value of 0. Only if both input lines are 0, effectively shutting off both transistors, will the output line contain a 1. Again, this is the exact opposite to the definition of OR given in Figure 4.13. Figure 4.18(a) is an implementation of a NOR gate, an acronym for *NOT OR*. To convert this to an OR gate, we do the same thing we did earlier—add a NOT gate to the output line. This gate is diagrammed in Figure 4.18(b).

Gates of the type shown in Figures 4.16 through 4.18 are not abstract entities that exist only in textbooks and classroom discussions. They are actual electronic devices that serve as the building blocks in the design and construction of modern computer systems. The reason for using gates rather than transistors is that a transistor is too elementary a device to act as the fundamental design component. It requires a designer to deal with such low-level issues as currents, voltages, and the laws of physics. Transistors, grouped together to form more powerful building blocks called gates, allow us to think and design at a higher level. Instead of dealing with the complex physical rules associated with discrete electrical devices, we can use the power and expressiveness of mathematics and logic to build computers.

This seemingly minor shift (from transistors to gates) has a profound effect on how computer hardware is designed and built. From this point on in our discussion of hardware design, we no longer need deal with anything electrical. Instead, our building blocks are AND, OR, and NOT gates, and our circuit construction rules are the rules of Boolean logic. This is another example of the importance of abstraction in computer science.

George Boole (1815–1864)

George Boole was a mid-nineteenth-century English mathematician and logician. He was the son of a shoemaker and had little formal education, having dropped out of school in the third grade. He taught himself mathematics and logic and mastered French, German, Italian, Latin, and Greek. He avidly studied the works of the great Greek and Roman philosophers such as Aristotle, Plato, and Euclid. He built on their work in logic, argumentation, and reasoning and, in 1854, produced a book titled *Introduction into the Laws of Thought*. This seminal work attempted to apply the formal laws of algebra and arithmetic to the principles of logic. That is, it treated reasoning as simply another branch of mathematics containing operators, variables, and transformation rules. He created a new form of logic containing the values *true* and *false* and the operators AND, OR, and NOT. He also developed a set of rules describing how to interpret and manipulate expressions that contain these values.

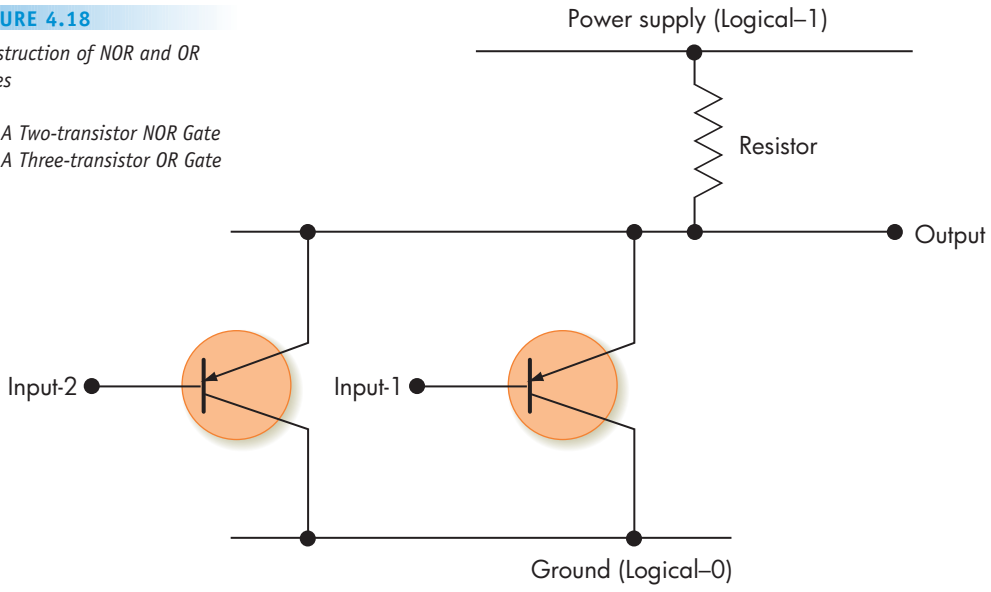
At the time of its development, the importance of this work was not apparent, and it languished in relative obscurity. However, 100 years later, Boole's ideas became the theoretical framework underlying the design of all computer systems. In his honor, these true/false expressions became known as **Boolean expressions**, and this branch of mathematics is called **Boolean logic** or **Boolean algebra**.

Even though he had very little formal schooling, Boole was eventually appointed Professor of Mathematics at Queens College in Cork, Ireland, and he received a gold medal from the Royal Mathematical Society. He is now universally recognized as one of the greatest mathematicians of the nineteenth century.

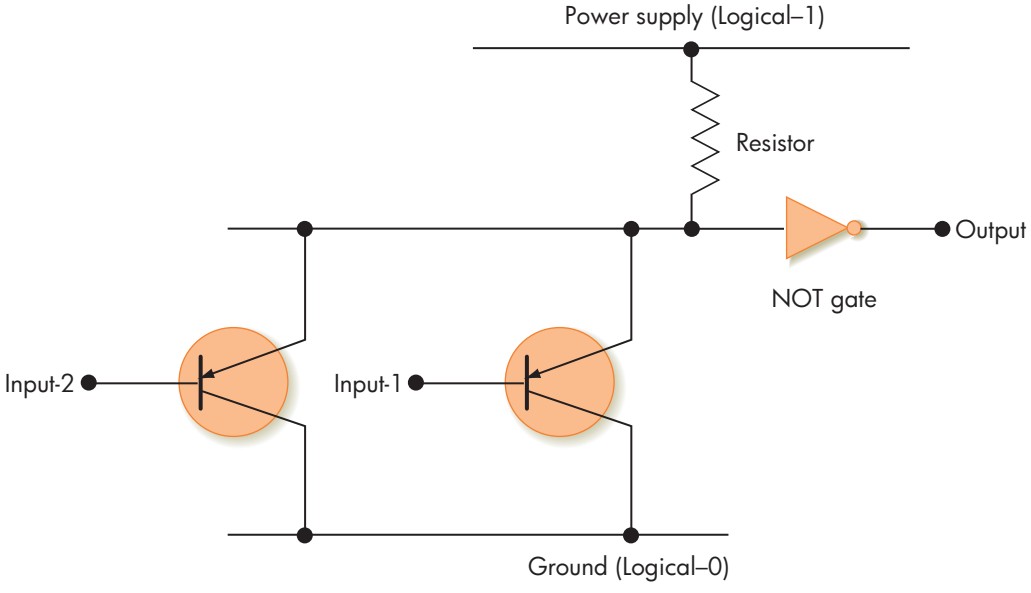
FIGURE 4.18

Construction of NOR and OR Gates

(a) A Two-transistor NOR Gate
(b) A Three-transistor OR Gate



(a)



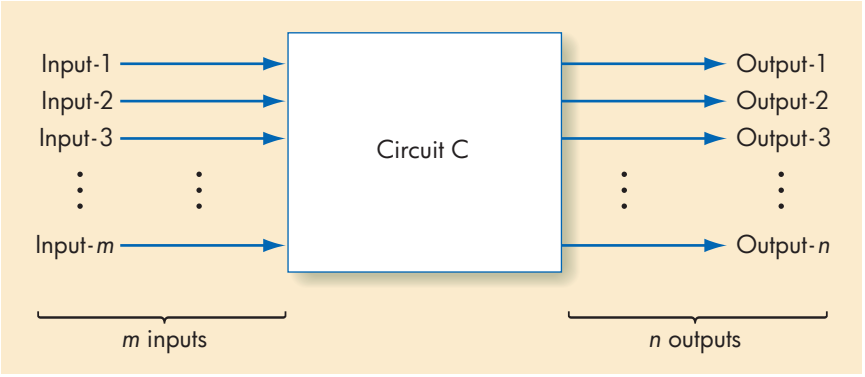
(b)

4.4 Building Computer Circuits

4.4.1 *Introduction*

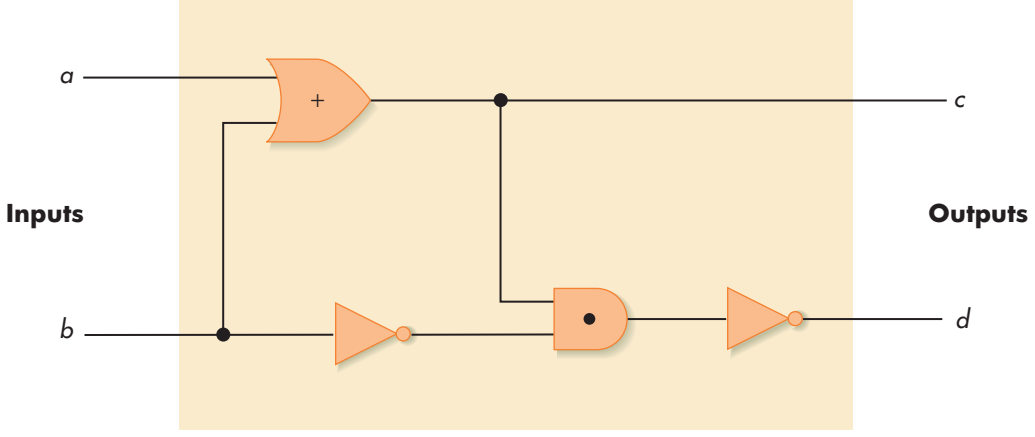
A **circuit** is a collection of logic gates that transforms a set of binary inputs into a set of binary outputs and in which the values of the outputs depend only on the current values of the inputs. (Actually, this type of circuit is more

FIGURE 4.19
Diagram of a Typical Computer Circuit



properly called a **combinational circuit**. We use the simpler term *circuit* in this discussion.) A circuit C with *m* binary inputs and *n* binary outputs is represented as shown in Figure 4.19.

Internally, the circuit shown in Figure 4.19 is constructed from the AND, OR, and NOT gates introduced in the previous section. (*Note:* We do not use the NAND and NOR gates diagrammed in Figure 4.17(a) and 4.18(a).) These gates can be interconnected in any way so long as the connections do not violate the constraints on the proper number of inputs and outputs for each gate. Each AND and OR gate must have exactly two inputs and one output. (Multiple-input AND and OR gates do exist, but we do not use them in our examples.) Each NOT gate must have exactly one input and one output. For example, the following is the diagram of a circuit with two inputs labeled *a* and *b* and two outputs labeled *c* and *d*. It contains one AND gate, one OR gate, and two NOT gates.



There is a direct relationship between Boolean expressions and **circuit diagrams** of this type. Every Boolean expression can be represented pictorially as a circuit diagram, and every output value in a circuit diagram can be written as a Boolean expression. For example, in the diagram shown, the two output values labeled *c* and *d* are equivalent to the following two Boolean expressions:

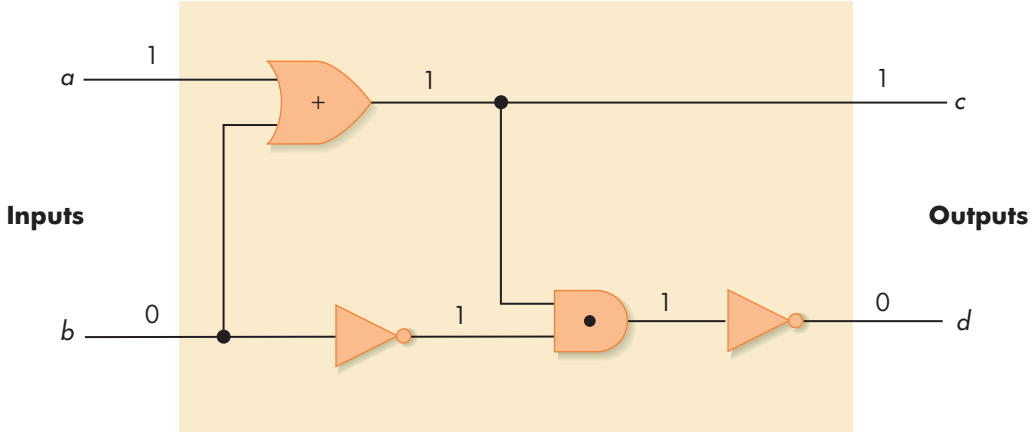
c = (*a* OR *b*)
d = NOT ((*a* OR *b*) AND (NOT *b*))

The choice of which representation to use depends on what we want to do. The pictorial view better allows us to visualize the overall structure of the circuit,

and is often used during the design stage. A Boolean expression may be better for performing mathematical or logical operations, such as verification and optimization, on the circuit. We use both representations in the following sections.

The value appearing on any output line of a circuit can be determined if we know the current input values and the transformations produced by each logic gate. (*Note:* There are circuits, called **sequential circuits**, that contain **feedback loops** in which the output of a gate is fed back as input to an earlier gate. The output of these circuits depends not only on the current input values but also on *previous* inputs. These circuits are typically used to build memory units because, in a sense, they can "remember" inputs. We do not discuss sequential circuits here.)

In the previous example, if $a = 1$ and $b = 0$, then the value on the c output line is 1, and the value on the d output line is 0. These values can be determined as follows:



Note that it is perfectly permissible to "split" or "tap" a line and send its value to two different gates. Here the input value b was split and sent to two separate gates.

The next section presents an algorithm for designing and building circuits from the three fundamental gate types AND, OR, and NOT. This enables us to move to yet a higher level of abstraction. Instead of thinking in terms of transistors and electrical voltages (as in Section 4.2.4) or in terms of logic gates and truth values (Section 4.3.2), we can think and design in terms of circuits for high-level operations such as addition and comparison. This makes understanding computer hardware much more manageable.

4.4.2 *A Circuit Construction Algorithm*

The circuit shown at the end of the previous section is simply an example, and is not meant to carry out any meaningful operation. To create circuits that perform useful arithmetic and logical functions, we need a way to convert a description of a circuit's desired behavior into a circuit diagram, composed of AND, OR, and NOT gates, that does exactly what we want it to do.

There are a number of **circuit construction algorithms** to accomplish this task, and the remainder of this section describes one such technique, called the **sum-of-products algorithm**, that allows us to design circuits. Section 4.4.3 demonstrates how this algorithm works by constructing actual circuits that all computer systems need.

STEP 1: TRUTH TABLE CONSTRUCTION. First, determine how the circuit should behave under all possible circumstances. That is, determine the binary value that should appear on each output line of the circuit for every possible combination of inputs. This information can be organized as a **truth table.** If a circuit has N input lines, and if each input line can be either a 0 or a 1, then there are 2^N combinations of input values, and the truth table has 2^N rows. For each output of the circuit, we must specify the desired output value for every row in the truth table.

For example, if a circuit has three inputs and two outputs, then a truth table for that circuit has $2^3 = 8$ input combinations and may look something like the following. (In this example, the output values are completely arbitrary.)

Inputs			Outputs	
a	b	c	Output-1	Output-2
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	0	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	0

$2^3 = 8$ input combinations

This circuit has two outputs labeled Output-1 and Output-2. The truth table specifies the value of each of these two output lines for every one of the eight possible combinations of inputs. We will use this example to illustrate the subsequent steps in the algorithm.

STEP 2: SUBEXPRESSION CONSTRUCTION USING AND AND NOT GATES. Choose any one output column of the truth table built in step 1 and scan down that column. Every place that you find a 1 in that output column, you build a Boolean *subexpression* that produces the value 1 (i.e., is true) for exactly that combination of input values and no other. To build this subexpression, you examine the value of each input for this specific case. If the input is a 1, use that input value directly in your subexpression. If the input is a 0, first take the NOT of that input, changing it from a 0 to a 1, and then use that **complemented** input value in your subexpression. You now have an input sequence of all 1s, and if all of these modified inputs are ANDed together (two at a time, of course), then the output value is a 1. For example, let's look at the output column labeled Output-1 from the previous truth table.

Inputs			Output-1	
a	b	c		
0	0	0	0	
0	0	1	0	
0	1	0	1	⟵ case 1
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	⟵ case 2
1	1	1	0	

There are two 1s in the column labeled Output-1; they are referred to as case 1 and case 2. We thus need to construct two subexpressions, one for each of these two cases.

In case 1, the inputs a and c have the value 0 and the input b has the value 1. Thus we apply the NOT operator to both a and c, changing them from 0 to 1. Because the value of b is 1, we can use b directly. We now have three modified input values, all of which have the value 1. ANDing these three values together yields the Boolean expression $(\bar{a} \cdot b \cdot \bar{c})$. This expression produces a 1 only when the input is exactly $a = 0$, $b = 1$, $c = 0$. In any other case, at least one of the three terms in the expression is 0, and when the AND operation is carried out, it produces a 0. (Check this yourself by trying some other input values and seeing what is produced.) Thus the desired subexpression for case 1 is

$$(\bar{a} \cdot b \cdot \bar{c})$$

The subexpression for case 2 is developed in an identical manner, and it results in

$$(a \cdot b \cdot \bar{c})$$

This subexpression produces a 1 only when the input is exactly $a = 1$, $b = 1$, $c = 0$.

STEP 3: SUBEXPRESSION COMBINATION USING OR GATES. Take each of the subexpressions produced in step 2 and combine them, two at a time, using OR gates. Each of the individual subexpressions produces a 1 for exactly one particular case where the truth table output is a 1, so the OR of the output of all of them produces a 1 in each case where the truth table has a 1 and in no other case. Consequently, the Boolean expression produced in step 3 implements exactly the function described in the output column of the truth table on which we are working. In the example above, the final Boolean expression produced during step 3 is

$$(\bar{a} \cdot b \cdot \bar{c}) + (a \cdot b \cdot \bar{c})$$

STEP 4: CIRCUIT DIAGRAM PRODUCTION. Construct the final circuit diagram. To do this, convert the Boolean expression produced at the end of step 3 into a circuit diagram, using AND, OR, and NOT gates to implement the AND, OR, and NOT operators appearing in the Boolean expression. This circuit diagram produces the output described in the corresponding column of the truth table created in step 1. The circuit diagram for the Boolean expression developed in step 3 is shown in Figure 4.20.

We have successfully built the part of the circuit that produces the output for the column labeled Output-1 in the truth table shown in step 1. We now repeat steps 2, 3, and 4 for any additional output columns contained in the truth table. (In this example there is a second column labeled Output-2. We leave the construction of that circuit as a practice exercise.) When we have constructed a circuit diagram for every output of the circuit, we are finished. The sum-of-products algorithm is summarized in Figure 4.21.

This has been a formal introduction to one particular circuit construction algorithm. The algorithm is not easy to comprehend in an abstract

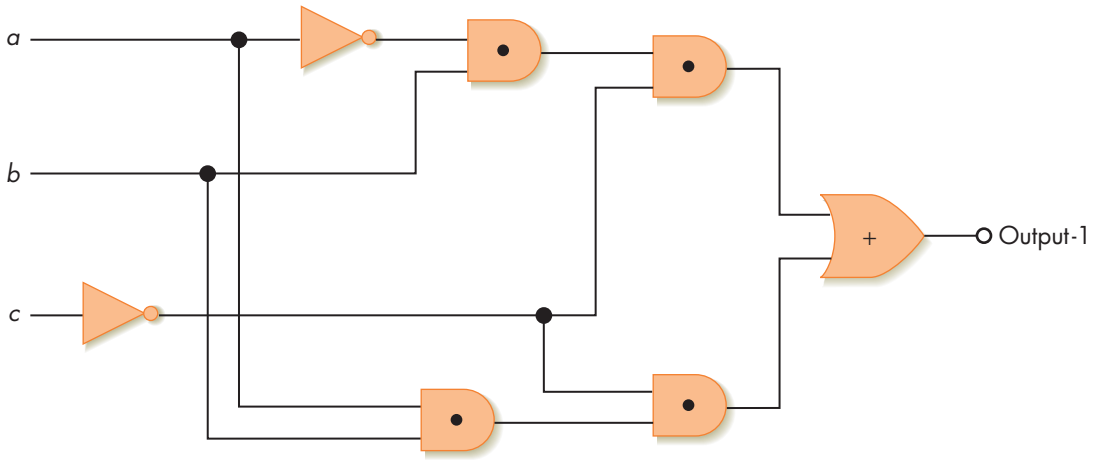


FIGURE 4.20
Circuit Diagram for the Output Labeled Output-1

sense. The next section clarifies this technique by using it to design two circuits that perform the operations of comparison and addition. Seeing it used to design actual circuits will make the steps of the algorithm easier to understand and follow.

We end this section by noting that the circuit construction algorithm just described does not always produce an **optimal** circuit, where *optimal* means that the circuit accomplishes its desired function using the smallest number of logic gates. For example, using the truth table shown on the bottom of page 166, our sum-of-products algorithm produced the seven-gate circuit shown in Figure 4.20. This is a correct answer in the sense that the circuit does produce the correct values for Output-1 for all combinations of inputs. However, it is possible to do much better.

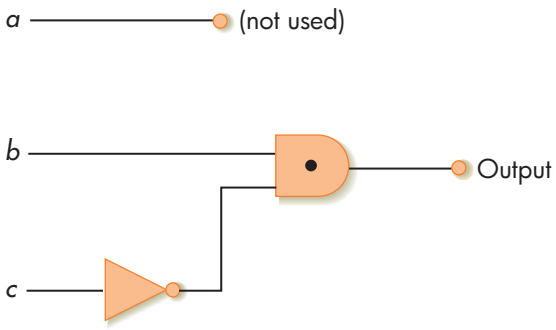


FIGURE 4.21
The Sum-of-Products Circuit Construction Algorithm

Sum-of-Products Algorithm for Constructing Circuits

1. Construct the truth table describing the behavior of the desired circuit
2. While there is still an output column in the truth table, do steps 3 through 6
3. Select an output column
4. Subexpression construction using AND and NOT gates
5. Subexpression combination using OR gates
6. Circuit diagram production
7. Done

The preceding circuit also produces the correct result using only two gates instead of seven. This difference is very important because each AND, OR, and NOT gate is a physical entity that costs real money, takes up space on the chip, requires power to operate, and generates heat that must be dissipated. Eliminating five unnecessary gates produces a real savings. The fewer gates we use, the cheaper, more efficient, and more compact are our circuits and hence the resulting computer. Algorithms for **circuit optimization**—that is, for reducing the number of gates needed to implement a circuit—are an important part of hardware design. Challenge Work problem 1 at the end of the chapter invites you to investigate this interesting topic in more detail.

PRACTICE PROBLEMS

1. Design the circuit to implement the output described in the column labeled Output-2 in the truth table on page 166.
2. Design a circuit using AND, OR, and NOT gates to implement the following truth table.

a	*b*	Output
0	0	0
0	1	1
1	0	1
1	1	0

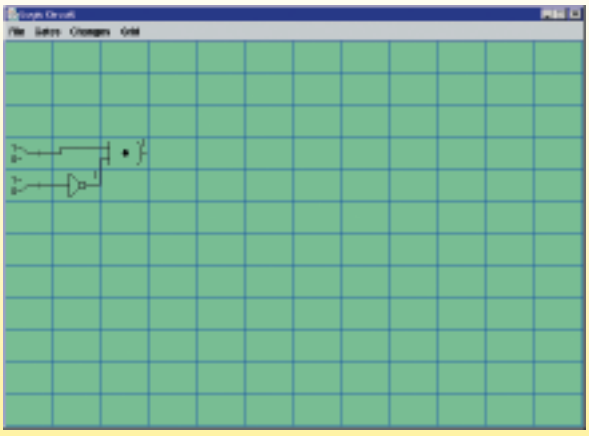
This is the **exclusive-OR** operation. It is true if and only if *a* is 1 or *b* is 1, but not both.

3. Build a circuit using AND, OR, and NOT gates to implement the following truth table.

a	*b*	*c*	Output
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

This is called a **full-ON/full-OFF** circuit. It is true if and only if all three of its inputs are OFF (0) or all three are ON (1).

LABORATORY EXPERIENCE 7



To give you hands-on experience working with logic circuits, the first laboratory experience in this chapter introduces you to a software package called a **circuit simulator**. This is a program that enables you to construct logic circuits from the AND, OR, and NOT gates just described, and then test them by observing the outputs of the circuits using any desired inputs. To the left is a simple example of the type of circuits that can be created—a circuit with one NOT gate, one AND gate, and two input switches (currently set to the values 1 and 0).

Note that the output of each gate is displayed on the screen, which allows you to determine if your circuit is or is not behaving correctly.

4.4.3 *Examples of Circuit Design and Construction*

Let's use the algorithm described in Section 4.4.2 to construct two circuits important to the operation of any real-world computer: a compare-for-equality circuit and an addition circuit.

A COMPARE-FOR-EQUALITY CIRCUIT. The first circuit we will construct is a **compare-for-*e*quality circuit**, or CE circuit, which tests two unsigned binary numbers for exact equality. The circuit produces the value 1 (*true*) if the two numbers are equal and the value 0 (*false*) if they are not. Such a circuit could be used in many situations. For example, in the shampooing algorithm in Figure 1.3(a), there is an instruction that says,

> Repeat steps 4 through 6 until the value of *WashCount* equals 2

Our CE circuit could accomplish the comparison between *WashCount* and 2 and return a true or false, depending on whether these two values were equal or not equal.

Let's start by using the sum-of-products algorithm in Figure 4.21 to construct a simpler circuit called 1-CE, short for *1*-bit *c*ompare for *e*quality. A 1-CE circuit compares two 1-bit values *a* and *b* for equality. That is, the circuit 1-CE produces a 1 as output if both its inputs are 0 or both its inputs are 1. Otherwise, 1-CE produces a 0. After designing 1-CE, we will use it to create a "full-blown" comparison circuit that can handle numbers of any size.

Step 1 of the algorithm says to construct the truth table that describes the behavior of the desired circuit. The truth table for the 1-CE circuit is

a	*b*	*Output*
0	0	1 ⟵ case 1 (both numbers equal to 0)
0	1	0
1	0	0
1	1	1 ⟵ case 2 (both numbers equal to 1)

In the output column of the truth table, there are two 1 values, labeled case 1 and case 2, so step 2 of the algorithm is to construct two subexpressions, one for each of these two cases. The subexpression for case 1 is $(\overline{a} \cdot \overline{b})$ because this produces the value 1 only when $a = 0$ and $b = 0$. The subexpression for case 2 is $(a \cdot b)$, which produces a 1 only when $a = 1$ and $b = 1$.

We now combine the outputs of these two subexpressions with an OR gate, as described in step 3, to produce the Boolean expression

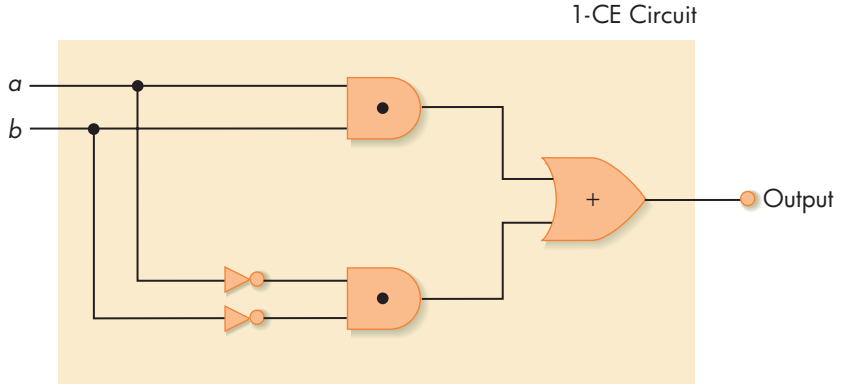
$$(\overline{a} \cdot \overline{b}) + (a \cdot b)$$

Finally, in step 4, we convert this expression to a circuit diagram, which is shown in Figure 4.22. The circuit shown in Figure 4.22 correctly compares two 1-bit quantities and determines if they are equal. If they are equal, it outputs a 1. If they are unequal, it outputs a 0.

The numbers compared for equality by a computer are usually much larger than a single binary digit. We want a circuit that correctly compares two numbers that contain *N* binary digits. To build this "*N*-bit compare-for-equality" circuit, we use *N* of the 1-CE circuits shown in Figure 4.22, one for each bit position in the numbers to be compared. Each 1-CE circuit produces a 1 if the two binary digits in its specific location are identical and produces a 0 if they are not. If every circuit produces a 1, then the two numbers are identical in every bit position, and they are equal. To check whether all our 1-CE circuits produce a 1, we simply AND together (two at a time) the outputs of all *N* 1-CE circuits. Remember that an AND gate produces a 1 if and only if both of its inputs are a 1. Thus the final output of the *N*-bit compare circuit is a 1 if and only if every pair of bits in the corresponding location of the two numbers is identical—that is, the two numbers are equal.

Figure 4.23 shows the design of a complete ***N*-bit compare-for-equality circuit** called CE. Each of the two numbers being compared, *a* and *b*, contains *N* bits, and they are labeled $a_{N-1}\ a_{N-2} \ldots a_0$ and $b_{N-1}\ b_{N-2} \ldots b_0$. The box labeled 1-CE in Figure 4.23 is the 1-bit compare-for-equality circuit shown in Figure 4.22. Looking at these figures, you can see that we have designed a very complex

FIGURE 4.22
One-Bit Compare for Equality Circuit



electrical circuit without the specification of a single electrical device. The only "devices" in those diagrams are gates to implement the logical operations AND, OR, and NOT, and the only "rules" we need to know in order to understand the diagrams are the transformation rules of Boolean logic. George Boole's "not very important" work is the starting point for the design of every circuit found inside a modern computer.

AN ADDITION CIRCUIT. Our second example of circuit construction is an addition circuit called ADD that performs binary addition on two unsigned *N*-bit integers. Typically, this type of circuit is called a **full adder**. For example, assuming $N = 6$, our ADD circuit would be able to perform the following 6-bit addition operation:

```
  1 1          (⟵ the carry bit)
  0 0 1 1 0 1  (the binary value 13)
+ 0 0 1 1 1 0  (the binary value 14)
  0 1 1 0 1 1  (the binary value 27, which is the correct sum)
```

Just as we did with the CE circuit, we carry out the design of the ADD circuit in two stages. First, we use the circuit construction algorithm of Figure 4.21 to

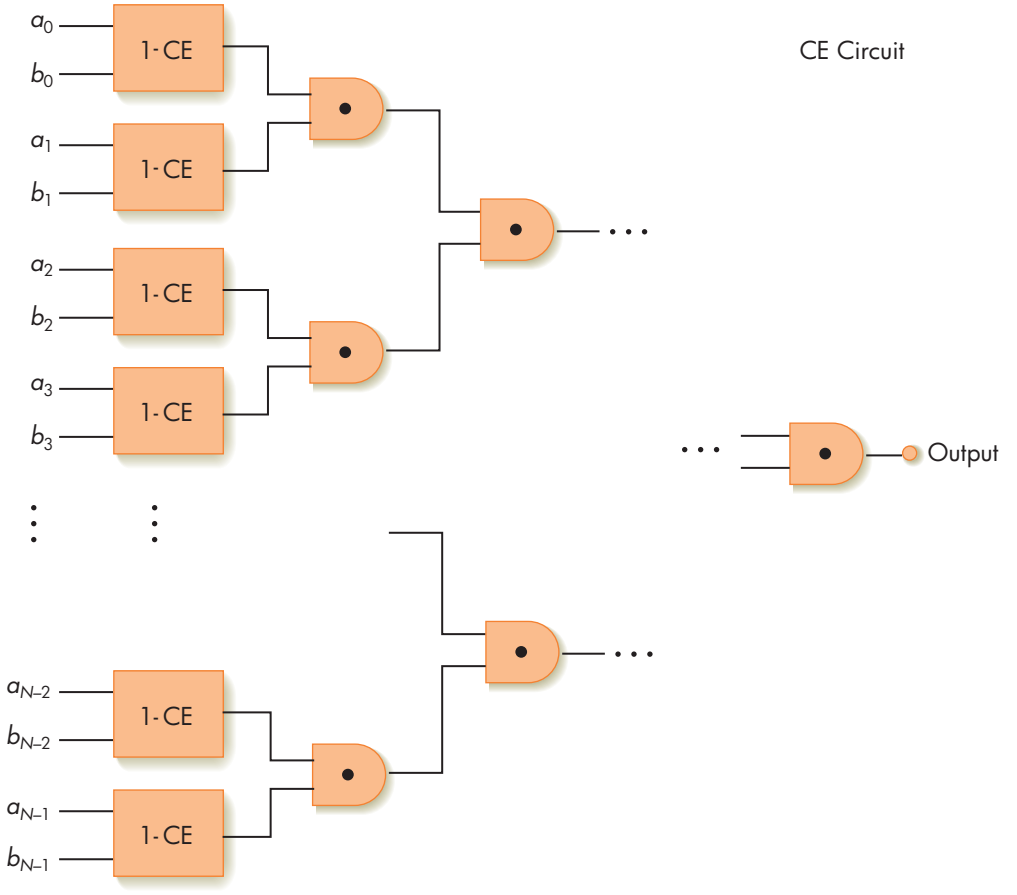
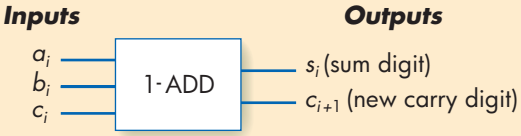


FIGURE 4.23
N-Bit Compare for Equality Circuit

FIGURE 4.24
The 1-ADD Circuit and Truth Table



Inputs			Outputs	
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

build a circuit called 1-ADD that adds a single pair of binary digits, along with a carry digit. We then interconnect N of these 1-ADD circuits to produce the complete N-bit full adder circuit ADD.

Looking at the addition example just shown, we see that summing the values in column i requires us to add three binary values—the two binary digits in that column, a_i and b_i, and the carry digit from the previous column, called c_i. Furthermore, the circuit must produce two binary outputs: a sum digit s_i and a new carry digit c_{i+1} that propagates to the next column. The pictorial representation of the 1-bit adder circuit 1-ADD and its accompanying truth table are shown in Figure 4.24.

Because the 1-ADD circuit being constructed has two outputs, s_i and c_{i+1}, we must use steps 2, 3, and 4 of the circuit construction algorithm twice, once for each output. Let's work on the sum output s_i first.

The s_i output column of Figure 4.24 contains four 1s, so we need to construct four subexpressions. In accordance with the guidelines given in step 2 of the construction algorithm, these four subexpressions are

Case 1: $\overline{a}_i \cdot \overline{b}_i \cdot c_i$
Case 2: $\overline{a}_i \cdot b_i \cdot \overline{c}_i$
Case 3: $a_i \cdot \overline{b}_i \cdot \overline{c}_i$
Case 4: $a_i \cdot b_i \cdot c_i$

Step 3 says to combine the outputs of these four subexpressions using three OR gates to produce the output labeled s_i in the truth table of Figure 4.24. The final Boolean expression for the sum output is

$$s_i = (\overline{a}_i \cdot \overline{b}_i \cdot c_i) + (\overline{a}_i \cdot b_i \cdot \overline{c}_i) + (a_i \cdot \overline{b}_i \cdot \overline{c}_i) + (a_i \cdot b_i \cdot c_i)$$

The logic circuit to produce the output whose expression is given above is shown in Figure 4.25. (This circuit diagram has been labeled to highlight the four separate subexpressions created during step 2, as well as the combining of the subexpressions in step 3 of the construction algorithm.)

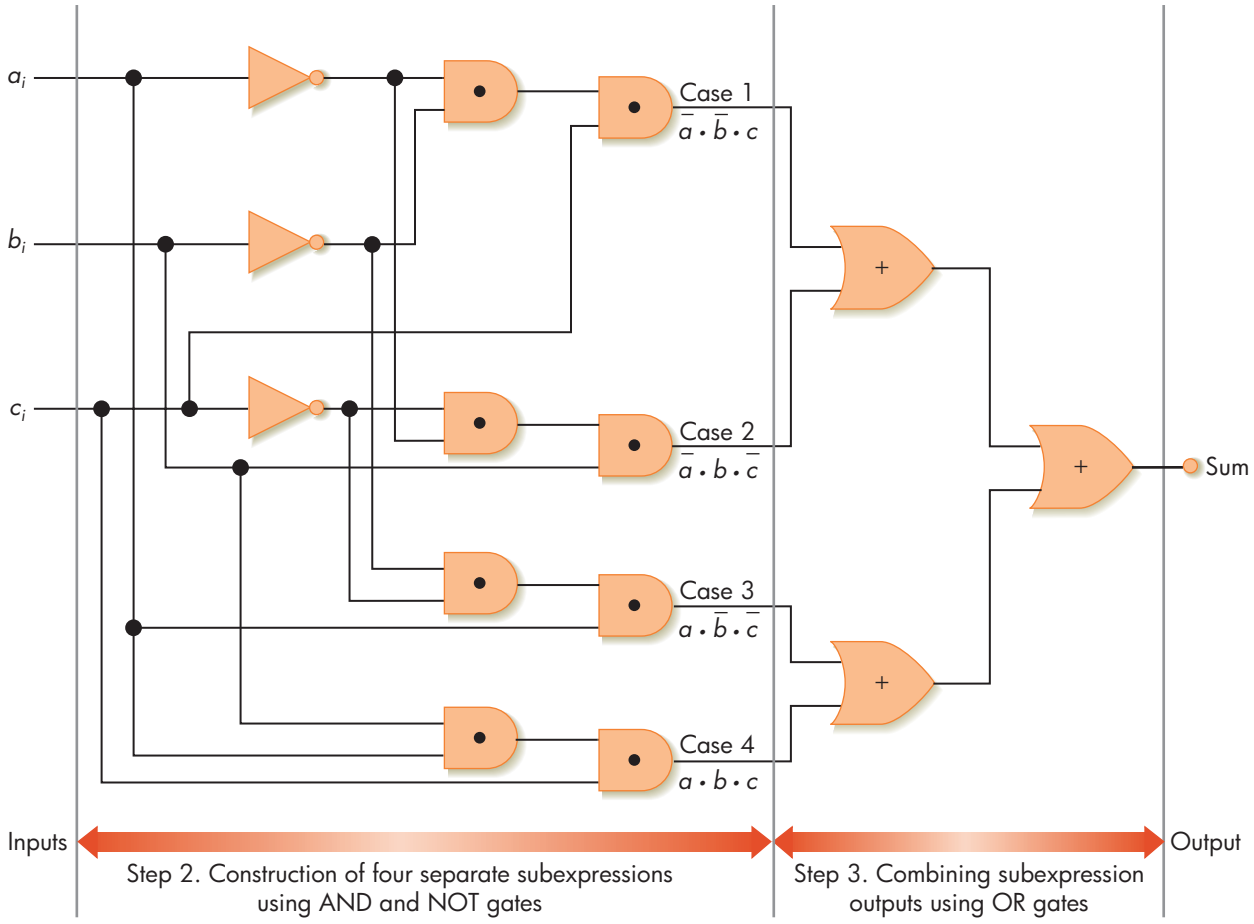


FIGURE 4.25
Sum Output for the 1-ADD Circuit

We are not yet finished, because the 1-ADD circuit in Figure 4.24 has a second output—the carry into the next column. That means the circuit construction algorithm must be repeated for the second output column, labeled c_{i+1}.

The c_{i+1} column also contains four 1s, so we again need to build four separate subcircuits, just as for the sum output, and combine them using OR gates. The construction proceeds in a fashion similar to the first part, so we leave the details as an exercise for the reader. The Boolean expression describing the carry output c_{i+1} of the 1-ADD circuit is

$$c_{i+1} = (\bar{a}_i \cdot b_i \cdot c_i) + (a_i \cdot \bar{b}_i \cdot c_i) + (a_i \cdot b_i \cdot \bar{c}_i) + (a_i \cdot b_i \cdot c_i)$$

We have now built the two parts of the 1-ADD circuit that produce the sum and the carry outputs. The complete 1-ADD circuit is constructed by simply putting these two pieces together. Figure 4.26 shows the complete (and admittedly quite complex) 1-ADD circuit to implement 1-bit addition. To keep the diagram from becoming an incomprehensible tangle of lines, we have drawn it in a slightly different orientation from Figures 4.22 and 4.25. Everything else is exactly the same.

When looking at this rather imposing diagram, do not become overly concerned with the details of every gate, every connection, and every operation. Figure 4.26 more importantly illustrates the *process* by which we design such a complex and intricate circuit: by transforming the idea of 1-bit binary

addition into an electrical circuit using the tools of algorithmic problem solving and symbolic logic.

How is the 1-ADD circuit shown in Figure 4.26 used to add numbers that contain *N* binary digits rather than just one? The answer is simple if we think about the way numbers are added by hand. (We discussed exactly this topic when developing the addition algorithm of Figure 1.2.) We add numbers one column at a time, moving from right to left, generating the sum digit, writing

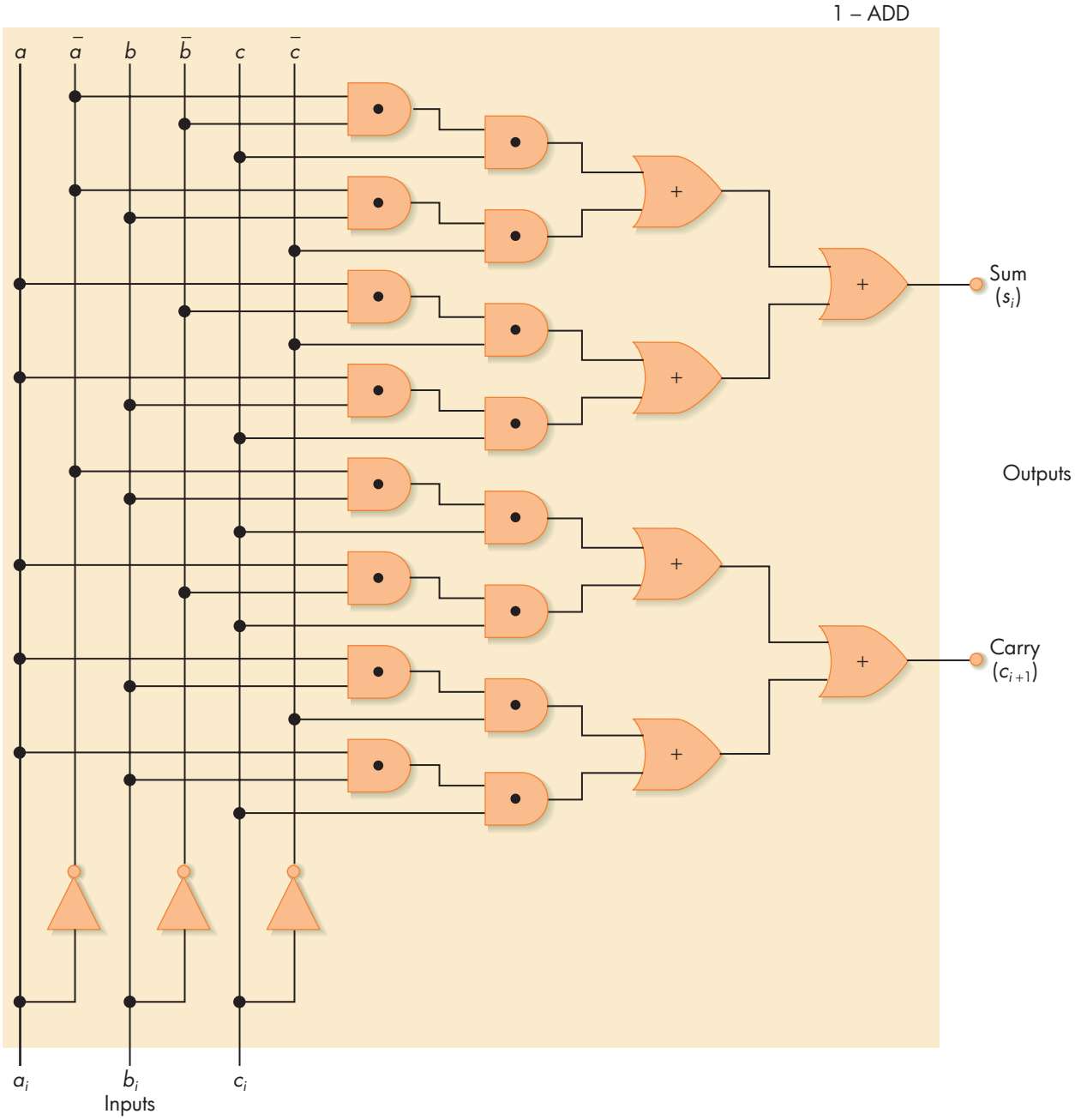


FIGURE 4.26
Complete 1-ADD Circuit for 1-Bit Binary Addition

it down, and sending any carry to the next column. The same thing can be done in hardware. We use N of the 1-ADD circuits shown in Figure 4.26, one for each column. Starting with the rightmost circuit, each 1-ADD circuit adds a single column of digits, generates a sum digit that is part of the final answer, and sends its carry digit to the 1-ADD circuit on its left, which replicates this process. After N repetitions of this process, all sum digits have been generated, and the N circuits have correctly added the two numbers.

The complete full adder circuit called ADD is shown in Figure 4.27. It adds the two N-bit numbers $a_{N-1}\ a_{N-2} \ldots a_0$ and $b_{N-1}\ b_{N-2} \ldots b_0$ to produce the $(N + 1)$ - bit sum $s_N s_{N-1}\ s_{N-2} \ldots s_0$. Because addition is one of the most common arithmetic operations, the circuit shown in Figure 4.27 (or something equivalent) is one of the most important and most frequently used arithmetic components. Addition circuits are found in every computer, workstation, and handheld calculator in the marketplace. They are even found in computer-controlled thermostats, clocks, and microwave ovens, where they enable us, for example, to add 30 minutes to the cooking time.

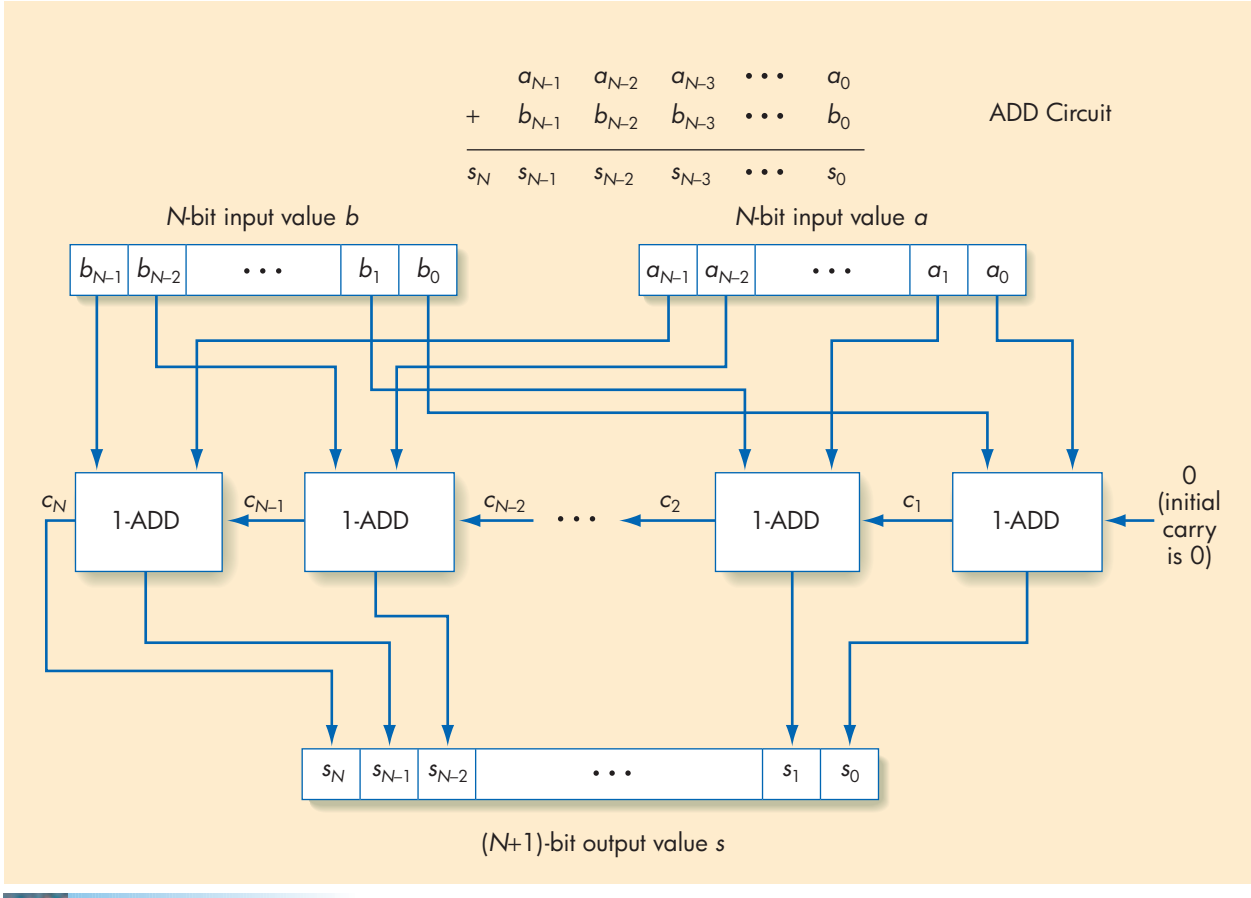


FIGURE 4.27
The Complete Full Adder ADD Circuit

Figure 4.27 is, in a sense, the direct hardware implementation of the addition algorithm shown in Figure 1.2. Although Figure 1.2 and Figure 4.27 are quite different, both represent essentially the same algorithm: the column-by-column addition of two *N*-bit numerical values. This demonstrates quite clearly that there are many different ways to express the same algorithm—in this case, pseudocode (Figure 1.2) and hardware circuits (Figure 4.27). Later chapters show additional ways to represent algorithms, such as machine language programs and high-level language programs. However, regardless of whether we use English, pseudocode, mathematics, or transistors to describe an algorithm, its fundamental properties are the same, and the central purpose of computer science—algorithmic problem solving—remains the same.

It may also be instructive to study the size and complexity of the ADD circuit just designed. Figure 4.27 shows that the addition of two *N*-bit integer values requires *N* separate 1-ADD circuits. Let's assume that $N = 32$, a typical value for modern computers. Referring to Figure 4.26, we see that each 1-ADD circuit uses 3 NOT gates, 16 AND gates, and 6 OR gates, a total of 25 logic gates. Thus the total number of logic gates used to implement 32-bit binary addition is $32 \times 25 = 800$ gates. Figures 4.16, 4.17(b), and 4.18(b) show that each AND and OR gate requires three transistors and each NOT gate requires one. Therefore, more than 2,200 transistors are needed to build a 32-bit adder circuit:

NOT:	$32 \times 3 = 96$ NOT gates	$\times$ 1 transistor/gate	$= 96$
AND:	$32 \times 16 = 512$ AND gates	$\times$ 3 transistors/gate	$= 1{,}536$
OR:	$32 \times 6 = 192$ OR gates	$\times$ 3 transistors/gate	$= 576$
		Total	$= 2{,}208$

(*Note:* Optimized 32-bit addition circuits can be constructed using as few as 500 to 600 transistors. However, this does not change the fact that it takes many, many transistors to accomplish this addition task.)

This computation emphasizes the importance of the continuing research into the miniaturization of electrical components. (See the box feature on Moore's Law earlier in this chapter.) If vacuum tubes were used instead of transistors, as was done in computers from about 1940 to 1955, the adder circuit shown in Figure 4.27 would be extraordinarily bulky; 2,208 vacuum tubes would occupy a space about the size of a large refrigerator. It would also generate huge amounts of heat, necessitating sophisticated cooling systems, and it would be very difficult to maintain. (Imagine the time it would take to locate a single burned-out vacuum tube from a cluster of 2,000.) Using something on the scale of the magnetic core technology described in Section 4.2.4 and shown in Figure 4.4, the adder circuit would fit into an area a few inches square. However, modern circuit technology can now achieve transistor densities greater than 1 billion transistors/cm^2. At this level, the entire ADD circuit of Figure 4.27 would easily fit in an area much, much smaller than the size of the period at the end of this sentence. That is why it is now possible to put powerful computer processing facilities not only in a room or on a desk but also inside a watch, a thermostat, or even inside the human body.

PRACTICE PROBLEMS

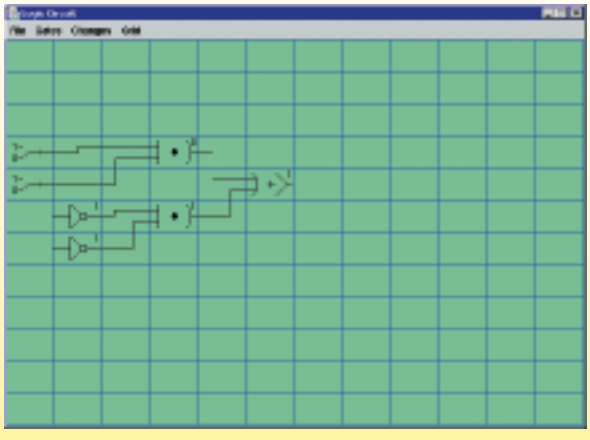
1. Design a circuit that implements a 1-bit compare-for-greater-than (1-GT) operation. This circuit is given two 1-bit values, a and b. It outputs a 1 if $a > b$, and outputs a 0 otherwise.
2. Use the circuit construction algorithm just described to implement the NOR operation shown in Figure 4.18(a). Remember that the truth table for the NOR operation is:

a	*b*	(*a* NOR *b*)
0	0	1
0	1	0
1	0	0
1	1	0

3. Use the circuit construction algorithm to implement the NXOR, the Not of the Exclusive OR operation, whose truth table is the following:

a	*b*	(*a* NXOR *b*)
0	0	1
0	1	0
1	0	0
1	1	1

LABORATORY EXPERIENCE 8



In the second laboratory experience of this chapter, you again use the circuit simulator software package. This time you construct circuits using the sum-of-products algorithm discussed in this section and shown in Figure 4.21. Using the simulator to design, build, and test actual circuits will give you a deeper understanding of how to use the sum-of-products algorithm to create circuits that solve specific problems. Here is an example of the implementation of the NXOR circuit described in Question 3 of the previous set of Practice Problems.

Dr. William Shockley (1910–1989)

Dr. William Shockley was the inventor (along with John Bardeen and Walter Brattain) of the transistor. His discovery has probably done as much to shape our modern world as any scientific advancement of the 20th century. He received the 1956 Nobel Prize in Physics and, at his death, was a distinguished professor at Stanford University.

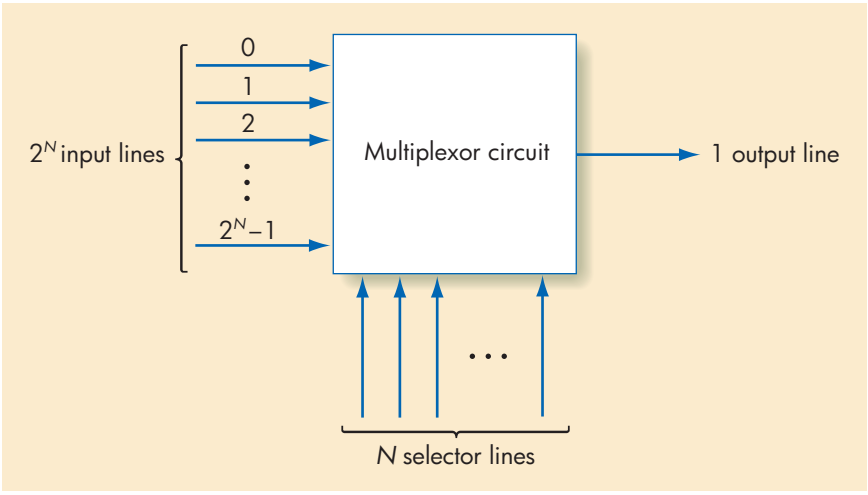
Shockley and his team developed the transistor in 1947 while working at Bell Laboratories. He left there in 1954 to set up the Shockley Semiconductor Laboratory in California—a company that was instrumental in the birth of the high-technology region called Silicon Valley. The employees of this company eventually went on to develop other fundamental advances in computing, such as the integrated circuit and the microprocessor.

Although Shockley's work has been compared to that of Pasteur, Salk, and Einstein in importance, his reputation and place in history have been forever tarnished by his outrageous and controversial racial theories. His education and training were in physics and electrical engineering, but Shockley spent the last years of his life trying to convince people of the genetic inferiority of blacks. He became obsessed with these ideas, even though he was ridiculed and shunned by colleagues who abandoned all contact with him. Although his work on the design of the transistor was of seminal importance, Shockley himself felt that his genetic theory on race and intelligence would ultimately be viewed as his most important contribution to science. By the time of his death in 1989, his intense racial bigotry prevented him from receiving the recognition that would otherwise have been his for monumental contributions in physics, engineering, and computer science.

4.5 Control Circuits

The previous section described the design of circuits for implementing arithmetic and logical operations. However, there are other, quite different, types of circuits that are also essential to the proper functioning of a computer system. This section briefly describes one of these important circuit types, **control circuits**, which are used not to implement arithmetic operations but to determine the order in which operations are carried out and to select the correct data values to be processed. In a sense, they are the sequencing and decision-making circuits inside a computer. These circuits are essential to the proper function of a computer because, as we noted in Chapter 1, algorithms and programs must be well ordered and must always know which operation to do next. The two major types of control circuits are called **multiplexors** and **decoders**, and, like everything else described in this chapter, they can be completely described in terms of gates and the rules of logic.

A **multiplexor** is a circuit that has 2^N **input lines** and 1 **output line**. Its function is to select exactly one of its 2^N input lines and copy the binary value on that input line onto its single output line. A multiplexor chooses one specific input by using an additional set of N lines called **selector lines**. (Thus the total number of inputs to the multiplexor circuit is $2^N + N$.) The 2^N input lines of a multiplexor are numbered 0, 1, 2, 3, . . . , $2^N - 1$. Each of the N selector lines can be set to either a 0 or a 1, so we can use the N selector lines to represent all binary values from 000 . . . 0 (N zeros) to 111 . . . 1 (N ones), which represent all integer values from 0 to $2^N - 1$. These numbers correspond exactly to the numbers of the input lines. Thus the binary number that appears on the selector lines can be interpreted as the identification number of the input line that is to be selected. Pictorially, a multiplexor looks like this:



For example, if we had four (2^2) input lines (i.e., $N = 2$) coming into our multiplexor, numbered 0, 1, 2, and 3, then we would need two selector lines. The four binary combinations that can appear on this pair of selector lines are 00, 01, 10, and 11, which correspond to the decimal values 0, 1, 2, and 3, respectively (refer to Figure 4.2). The multiplexor selects the one input line whose identification number corresponds to the value appearing on the selector lines and copies the value on that input line to the output line. If, for example, the two selector lines were set to 1 and 0, then a multiplexor circuit would pick input line 2 because 10 in binary is 2 in decimal notation.

Implementing a multiplexor using logic gates is not difficult. Figure 4.28 shows a simple multiplexor circuit with $N = 1$. This is a multiplexor with two (2^1) input lines and a single selector line.

In Figure 4.28 if the value on the selector line is 0, then the bottom input line to AND gate 2 is always 0, so its output is always 0. Looking at AND gate 1, we see that the NOT gate changes its bottom input value to a 1. Because (1 AND *a*) is always *a,* the output of the top AND gate is equal to the value of *a,* which is the value of the input from line 0. Thus the two inputs to the OR gate are 0 and *a*. Because the value of the expression (0 OR *a*) is identical to *a,* by setting the selector line to 0 we have, in effect, selected as our output the value that appears on line 0. You should confirm that if the selector line has

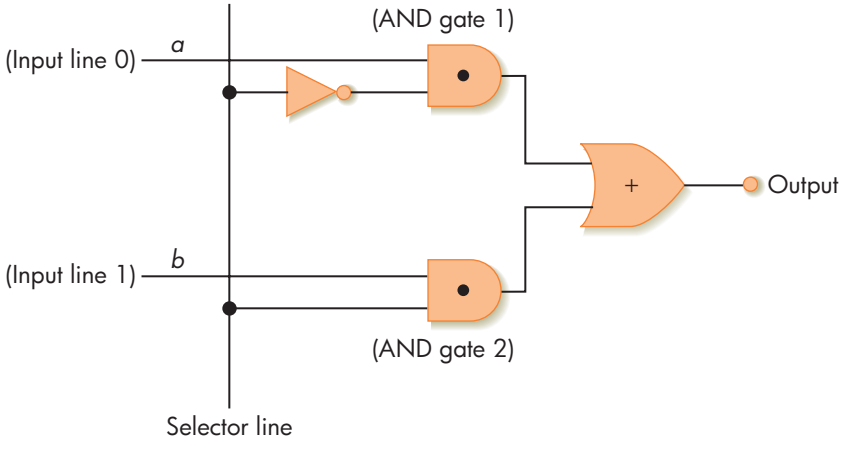
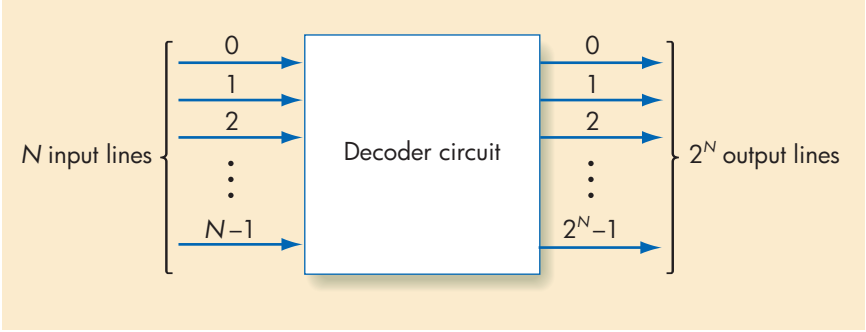


FIGURE 4.28
A Two-Input Multiplexor Circuit

the value 1, then the output of the circuit in Figure 4.28 is *b*, the value appearing on line 1. We can design multiplexors with more than two inputs in a similar fashion, although they rapidly become more complex.

The second type of control circuit is called a **decoder** and it operates in the opposite way from a multiplexor. A decoder has N input lines numbered 0, 1, 2, . . . , $N - 1$ and 2^N output lines numbered 0, 1, 2, 3, . . . , $2^N - 1$.



Each of the N input lines of the decoder can be set to either a 0 or a 1, and when these N values are interpreted as a single binary number, they can represent all integer values from 0 to $2^N - 1$. It is the job of the decoder to determine the value represented on its N input lines and then send a signal (i.e., a 1) on the single output line that has that identification number. All other output lines are set to 0.

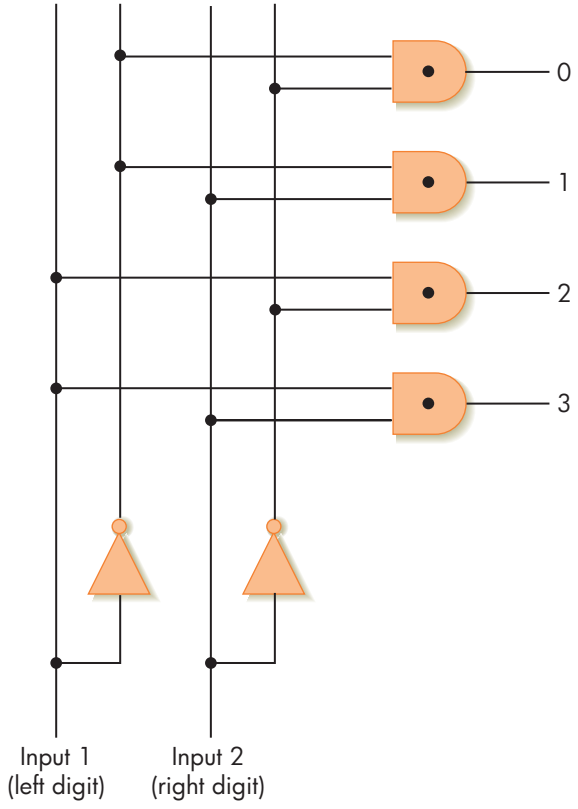
For example, if our decoder has three input lines, it has eight (2^3) output lines numbered 0 to 7. These three input lines can represent all binary values from 000 to 111, which is from 0 to 7 in decimal notation. If, for example, the binary values on the three input lines are 101, which is a 5, then a signal (a binary 1) would be sent out by the decoder on output line 5. All other output lines would contain a 0.

Figure 4.29 shows the design of a 2-to-4 decoder circuit with two input lines and four (2^2) output lines. These four output lines are labeled 0, 1, 2, and 3, and the only output line that carries a signal value of 1 is the line whose identification number is identical to the value appearing on the two input lines. For example, if the two inputs are 11, then line 3 should be set to a 1 (11 in binary is 3 in decimal). This is, in fact, what happens because the AND gate connected to line 3 is the only one whose two inputs are equal to a 1. You should confirm that this circuit behaves properly when it receives the inputs 00, 01, and 10 as well.

Together, decoder and multiplexor circuits enable us to build computer systems that execute the correct instructions using the correct data values. For example, assume we have a computer that can carry out four different types of arithmetic operations—add, subtract, multiply, and divide. Furthermore, assume that these four instructions have code numbers 0, 1, 2, and 3, respectively. We could use a decoder circuit to ensure that the computer performs the correct instruction. We need a decoder circuit with two input lines. It receives as input the 2-digit code number (in binary) of the instruction that we want to perform: 00 (add), 01 (subtract), 10 (multiply), or 11 (divide). The decoder interprets this value and sends out a signal on the correct output line. This signal is used to select the proper arithmetic circuit and cause it to perform the desired operation. This behavior is diagrammed in Figure 4.30.

Whereas a decoder circuit can be used to select the correct instruction, a multiplexor can help ensure that the computer executes this instruction using the correct data. For example, suppose our computer has four special registers

FIGURE 4.29
A 2-to-4 Decoder Circuit



called R0, R1, R2, and R3. (For now, just consider a register to be a place to store a data value. We describe registers in more detail in the next chapter.) Assume that we have built a circuit called *test-if-zero* that can test whether any of these four registers contains the value 0. (This is actually quite similar to the CE circuit of Figure 4.23.) We can use a multiplexor circuit to select the register that we wish to send to the test-if-zero circuit. This is shown in Figure 4.31. If we want to test if register R2 in Figure 4.31 is 0, we simply put the binary value 10 (2 in decimal notation) on the two selector lines. This selects register R2, and only its value passes through the multiplexor and is sent to the test circuit.

There are many more examples of the use of control circuits in Chapter 5, which examines the execution of programs and the overall organization of a computer system.

FIGURE 4.30
Example of the Use of a Decoder Circuit

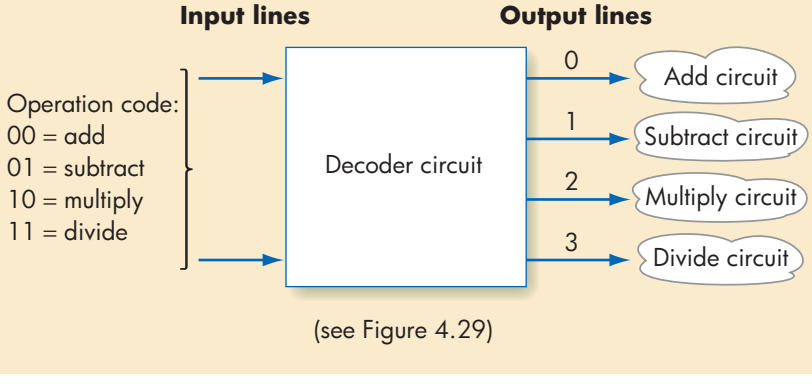
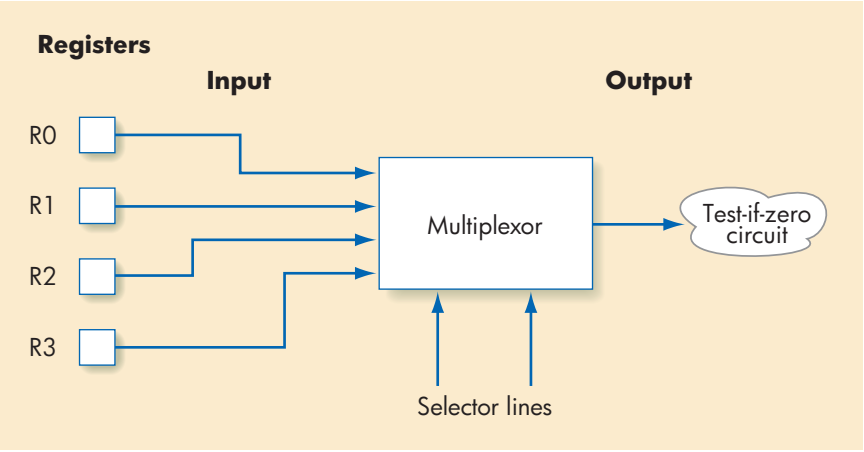


FIGURE 4.31
Example of the Use of a Multiplexor Circuit



4.6 Conclusion

We began our discussion on the representation of information and the design of computer circuits with the most elementary component, bistable electronic devices such as transistors. We showed how they can be used to construct logic gates that in turn can be used to implement circuits to carry out useful functions. Our purpose here was not to make you an expert in specifying and designing computer circuits but to demonstrate how it is possible to implement high-level arithmetic operations using only low-level electronic components such as transistors. We also demonstrated how it is possible to reorient our viewpoint and raise our level of abstraction. We changed the level of discussion from electricity to arithmetic, from hardware devices to mathematical behavior, from form to function. This is one of the first steps up the hierarchy of abstractions introduced in Figure 1.9.

Chapter 5 continues this "upward climb" to yet higher levels of abstraction. It shows how arithmetic circuits, such as compare for equality and addition (Section 4.4.3), and control circuits, such as multiplexors and decoders (Section 4.5), can be used to construct entire computer systems.

After reading this chapter, you may have the feeling that although you understand the individual concepts that are covered, you still don't understand, in the grand sense, what computers are or how they work. You may feel that you can follow the details but can't see the "big picture." One possible reason is that this chapter looks at computers from a very elementary viewpoint, by studying different types of specialized circuits. This is analogous to studying the human body as a collection of millions of cells of different types—blood cells, brain cells, skin cells, and so on. Cytology is certainly an important part of the field of biology, but understanding only the cellular structure of the human body provides no intuitive understanding of what people are and how we do such characteristic things as walk, eat, and breathe. Understanding these complex actions derives not from a study of molecules, genes, or cells, but from a study of higher-level organs, such as lungs, stomach, and muscles, and their interactions.

That is exactly what happens in Chapter 5, in which we examine higher-level computer components such as processors, memory, and instructions and begin our study of the topic of computer organization.

EXERCISES

1. Given our discussion of positional numbering systems in Section 4.2.1, see whether you can determine the decimal value of the following number:

a. 133 (base 4)

b. 367 (base 8, also called *octal*)

c. 1BA (base 16, also called *hexadecimal*. B is the digit that represents 11; A is the digit that represents 10.)

2. In Exercise 1(c), we use the letters A and B as digits of the base-16 number. Explain why that is necessary.

3. Determine the decimal value of the following unsigned binary numbers:

a. 11000 c. 1111111

b. 110001 d. 1000000000

4. Using 8 bits, what is the unsigned binary representation of each of the following values:

a. 23

b. 55

c. 275

Did anything "unusual" happen when determining the correct answer to part (c)?

5. Assume that the following 10-bit numbers represent signed integers using sign/magnitude notation. The sign is the leftmost bit and the remaining 9 bits represent the magnitude. What is the decimal value of each?

a. 1000110001 c. 1000000001

b. 0110011000 d. 1000000000

6. Assume that we use 10 bits to represent signed integers, using sign/magnitude notation. What are the largest (in absolute value) positive and negative numbers that can be represented on our system?

7. Show the step-by-step addition of the following two 10-bit unsigned binary values, including showing the carry bit to each successive column:

```
  0011100011
+ 0001101110
```

8. Assume that our computer stores decimal numbers using 16 bits—10 bits for a sign/magnitude mantissa and 6 bits for a sign/magnitude base-2 exponent. (This is exactly the same representation used in the text.) Show the internal representation of the following decimal quantities.

a. +7.5 b. –20.25 c. –1/64

9. Using the ASCII code set given in Figure 4.3, show the internal binary representation for the following character strings:

a. AbC c. $25.00

b. Mike d. (*a*+*b*)

10. How many binary digits would it take to represent the following phrase in ASCII code? In UNICODE? (Do not include the " " marks.)

"Invitation to Computer Science"

11. a. How many bits does it take to store a 3-minute song using an audio encoding method that samples at the rate of 40,000 bits/second, has a bit depth of 16, and does not use compression? What if it uses a compression scheme with a compression ratio of 5:1?

b. How many bits does it take to store an uncompressed 1,200 × 800 RGB color image? If we found out that the image actually takes only 2.4 Mbits, what is the compression ratio?

12. Show how run-length encoding can be used to compress the following text stream:

xxxyyyyyyzzzzAAxxxx

What is the compression ratio? (Assume each digit and letter requires 8 bits.)

13. Using the variable length code shown in Figure 4.8, give the internal coding of the following Hawaiian words along with the amount of savings over the standard fixed-length four bit representation:

a. KAI

b. MAUI

c. MOLOKAI

Explain the problem that occurred with part (c).

14. The primary advantage of using the binary numbering system rather than the decimal system to represent data is reliability, as noted in Section 4.2.3. Describe two disadvantages of using binary rather than decimal notation for the internal representation of information.

15. Assume that $a = 1$, $b = 2$, and $c = 2$. What is the value of each of the following Boolean expressions?

a. $(a > 1)$ OR $(b = c)$

b. $[(a + b) > c]$ AND $(b \leq c)$

c. NOT $(a = 1)$

d. NOT $[(a = b)$ OR $(b = c)]$

16. Assume that $a = 5$, $b = 2$, and $c = 3$. What problem do you encounter when attempting to evaluate the following Boolean expression?

$(a = 1)$ AND $(b = 2)$ OR $(c = 3)$

How can this problem be solved?

17. Using the circuit construction algorithm of Section 4.4.2, design a circuit using only AND, OR, and NOT gates to implement the following truth table:

a	*b*	*Output*
0	0	1
0	1	1
1	0	1
1	1	0

This operation is termed **NAND,** for *Not AND,* and it can be constructed as a single gate, as shown in Figure 4.17(a). Assume that you do not have access to a NAND gate and must construct it from AND, OR, and NOT.

18. Using the circuit construction algorithm of Section 4.4.2, design a circuit using only AND, OR, and NOT gates to implement the following truth table.

a	*b*	*Output*
0	0	1
0	1	1
1	0	0
1	1	1

This operation is termed **logical implication**, and it is an important operator in symbolic logic.

19. Build a **majority-rules circuit**. This is a circuit that has three inputs and one output. The value of its output is 1 if and only if two or more of its inputs are 1; otherwise, the output of the circuit is 0. For example, if the three inputs are 0, 1, 1, your circuit should output a 1. If its three inputs are 0, 1, 0, it should output a 0. This circuit is frequently used in **fault-tolerant computing**—environments where a computer must keep working correctly no matter what, for example as on a deep-space vehicle where making repairs is impossible. In these conditions, we might choose to put three computers on board and have all three do every computation; if two or more of the systems produce the same answer, we accept it. Thus, one of the machines could fail and the system would still work properly.

20. Design an **odd-parity circuit**. This is a circuit that has three inputs and one output. The circuit outputs a 1 if and only if an even number (0 or 2) of its inputs are a 1. Otherwise, the circuit outputs a 0. Thus the sum of the number of 1 bits in the input and the output is always an odd number. (This circuit is used in error checking. By adding up the number of 1 bits, we can determine whether any single input bit was accidentally changed. If it was, the total number of 1s is an even number when we know it should be an odd value.)

21. Design a **1-bit subtraction circuit**. This circuit takes three inputs—two binary digits a and b and a borrow digit from the previous column. The circuit has two outputs—the difference ($a - b$), including the borrow, and a new borrow digit that propagates to the next column. Create the truth table and build the circuit. This circuit can be used to build N-bit subtraction circuits.

22. How many selector lines would be needed on a four-input multiplexor? On an eight-input multiplexor?

23. Design a **four-input multiplexor circuit**. Use the design of the two-input multiplexor shown in Figure 4.28 as a guide.

24. Design a **3-to-8 decoder circuit**. Use the design of the 2-to-4 decoder circuit shown in Figure 4.29 as a guide.

CHALLENGE WORK

1. Circuit optimization is a very important area of hardware design. As we mentioned earlier in the chapter, each gate in the circuit represents a real hardware device that takes up space on the chip, generates heat that must be dissipated, and increases costs. Therefore, the elimination of unneeded gates can represent a real savings. Circuit optimization investigates techniques to construct a new circuit that behaves identically to the original one but with fewer gates. The basis for circuit optimization is the transformation rules of symbolic logic. These rules allow you to transform one Boolean expression into an equivalent one that entails fewer operations. For example, the *distributive law* of logic says that $(a \cdot b) + (a \cdot c) = a \cdot (b + c)$. The expressions on either side of the = sign are functionally identical, but the one on the right determines its value using one less gate (one AND gate and one OR gate instead of two AND gates and one OR gate).

Read about the transformation rules of binary logic and techniques of circuit optimization. Using these rules, improve the full adder circuit of Figure 4.27 so that it requires fewer than 2,208 transistors. Explain your improvements and determine exactly how many fewer transistors are required for your "new and improved" full adder circuit.

2. This chapter briefly described an alternative signed integer representation technique called **two's complement representation**. This popular method is based on the concepts of *modular arithmetic,* and it does not suffer from the problem of two different representations for the quantity 0. Read more about two's complement and write a report describing how this method works, as well as algorithms for adding and subtracting numbers represented in two's complement notation. In your report give the 16-bit, two's complement representation for the signed integer values +45, −68, −1, and 0. Then show

how to carry out the arithmetic operations 45 + 45, 45 + (–68), and 45 – (–1).

3. In Section 4.2.2 we describe lossless compression schemes, such as run-length encoding and variable length codes. However, most compression schemes in use today are lossy and only achieve extremely high rates of compression at the expense of losing some of the detail contained in the sound or image. Often they base their compression techniques on specific knowledge of the characteristics of the human ear or eye. For example, it is well known that the eye is much more sensitive to changes in brightness (luminance) than to changes in color (chrominance). The JPEG compression algorithm exploits this fact when it is compressing a photographic image.

Read about the JPEG image compression algorithm to learn how it is able to achieve compression ratios of 10:1 or even 20:1. A good place to start would be the JPEG home page, located at *www.jpeg.org*.

FOR FURTHER READING

The following book offers an excellent discussion of the major topics covered in this chapter—the representation of information, logic gates, and circuit design. It is one of the most widely used texts in the field of hardware and logic design.

Patterson, D., and Hennessey, J. *Computer Organization and Design: The Hardware Software Interface,* 3rd ed. revised, San Francisco: Morgan Kaufman, 2007.

Chapter 4: "Arithmetic for Computers." This is an excellent introduction to the representation of information inside a computer.

Appendix B: "The Basics of Logic Design."

Among the other excellent books about gates, circuits, hardware, and logic design are

Mano, M., Kime, C. *Logic and Computer Design Fundamentals*, 4th ed. Englewood Cliffs, NJ: Prentice-Hall, 2007.

Reid, T. R. *The Chip: How Two Americans Invented the Microchip and Launched a Revolution*. New York: Random House, 2001.

Wakerly, J.F. *Digital Design*, 4th ed. Englewood Cliffs, NJ: Prentice-Hall, 2005.

Finally, a good reference text on the internal representation of numeric information and arithmetic algorithms is

Koren, I. *Computer Arithmetic Algorithms*, 2nd ed. Natick, MA: A.K. Peters, 2001.

CHAPTER 5

Computer Systems Organization

5.1 Introduction

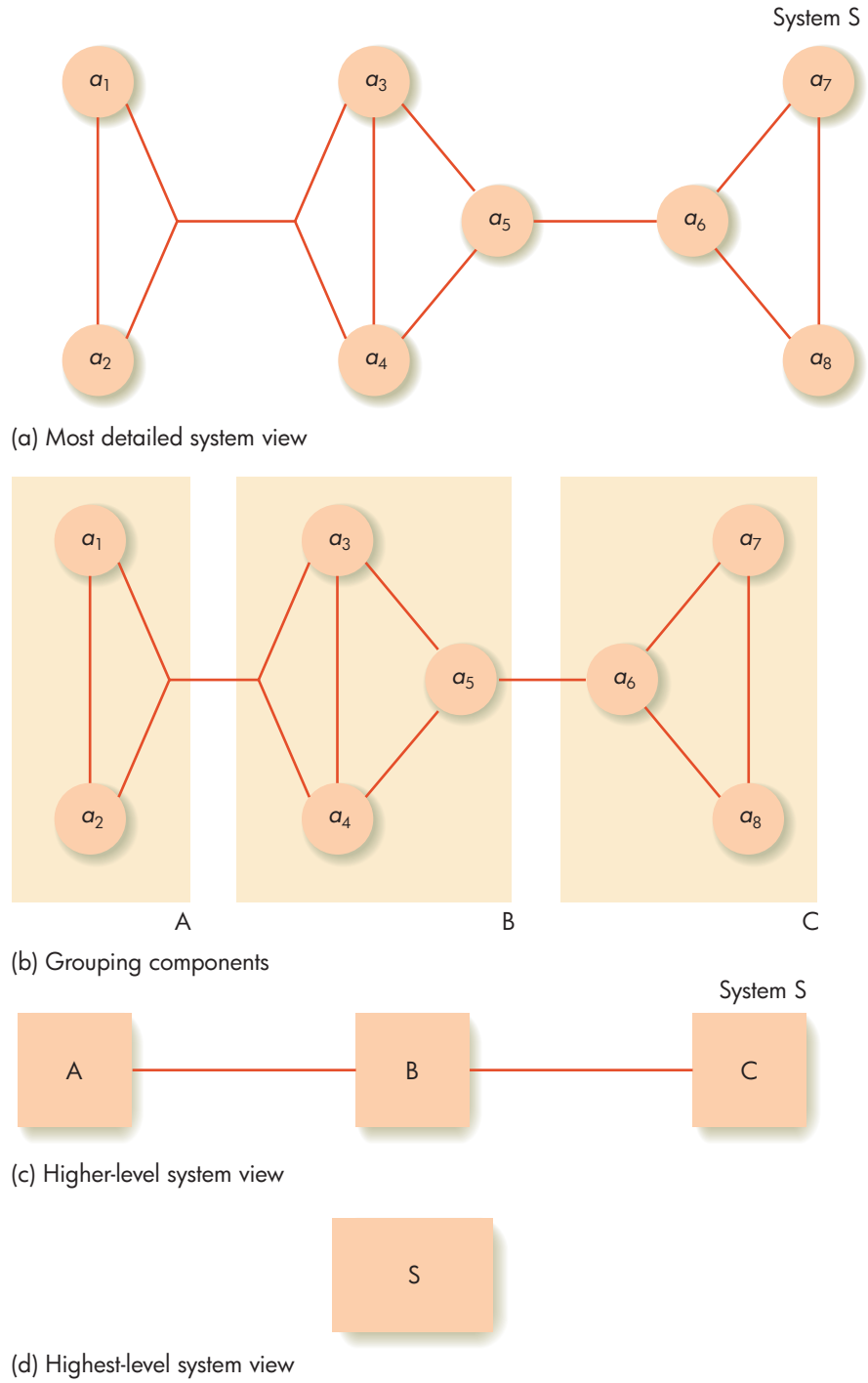
Chapter 4 introduced the elementary building blocks of computer systems—transistors, gates, and logic circuits. Though this information is essential to understanding computer hardware—just as knowledge of atoms and molecules is necessary for any serious study of chemistry—it produces a very low-level view of computer systems. Even students who have mastered the material may still ask, "OK, but how do computers *really* work?" Gates and circuits operate on the most elemental of data items, binary 0s and 1s, whereas people reason and work with more complex units of information, such as decimal numbers, character strings, and instructions. To understand how computers process this kind of information, we must look at higher-level components than gates and circuits. We must study computers as collections of **functional units** or **subsystems** that perform tasks such as instruction processing, information storage, computation, and data transfer. The branch of computer science that studies computers in terms of their major functional units and how they work is **computer organization**, and that is the subject of this chapter. This higher-level viewpoint will give us a much better understanding of how a computer really works.

All of the functional units introduced in this chapter are built from the gates and circuits of Chapter 4. However, those elementary components will no longer be visible because we will adopt a different viewpoint, a different perspective, a different **level of abstraction**. This is an extremely important point; as we have said, the concept of abstraction is used throughout computer science. Without it, it would be virtually impossible to study computer design or any other large, complex system.

For example, suppose that system S is composed of a large number of elementary components a_1, a_2, a_3, . . . interconnected in very intricate ways, as shown in Figure 5.1(a). This is equivalent to viewing a computer system as thousands or millions of individual gates. For some purposes it may be necessary to view system S at this level of detail, but for other applications the details could be overwhelming. To deal with this problem, we can redefine the primitives of system S by grouping the elementary components a_1, a_2, a_3, . . ., as shown in Figure 5.1(b), and calling these larger units (A, B, C) the basic building blocks of system S. A, B, and C are treated as nondecomposable elements whose internal construction is hidden from view. We care only about what functions these components perform and how they interact. This leads to the higher-level system view shown in Figure 5.1(c), which is certainly a great deal simpler than the one shown in Figure 5.1(a), and this is how this chapter approaches the topic of computer hardware. Our primitives are much larger components, similar to A, B, and C, but internally they are still made up of the gates and circuits of Chapter 4.

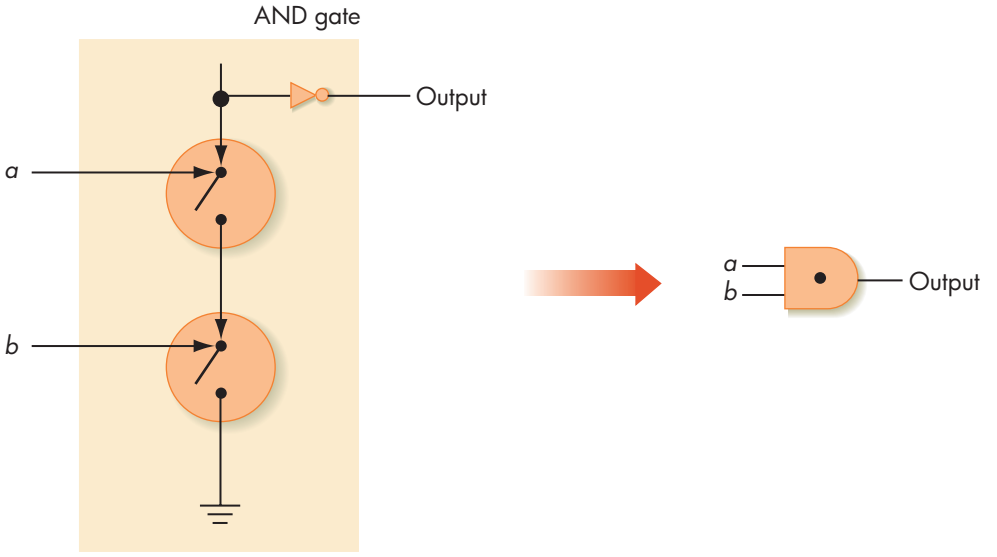
FIGURE 5.1

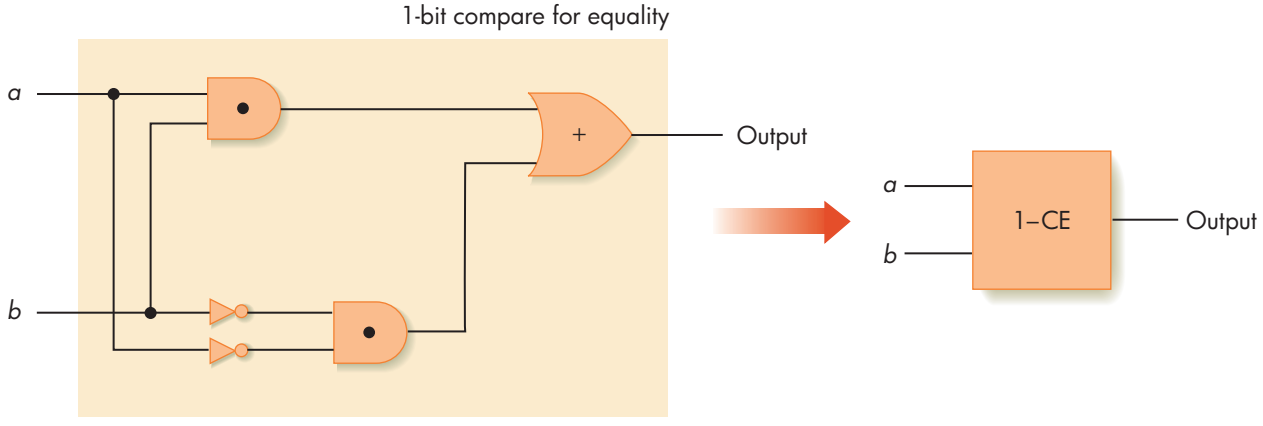
The Concept of Abstraction



This "abstracting away" of unnecessary detail can be done more than once. For example, at a later point in the study of system S, we may no longer care about the behavior of individual components A, B, and C. We may now wish to treat the entire system as a single primitive, nondecomposable entity whose inner workings are no longer important. This leads to the extremely simple system view shown in Figure 5.1(d), a view that we will adopt in later chapters.

Figures 5.1(a), (c), and (d) form what is called a **hierarchy of abstractions**. A hierarchy of abstractions of computer science forms the central theme of this text, and it was initially diagrammed in Figure 1.9. We have already seen this idea in action in Chapter 4, where transistors are grouped into gates and gates into circuits:





This process continues into Chapter 5, where we use the addition and comparison circuits of Section 4.4.3 to build an arithmetic unit and use the multiplexor and decoder circuits of Section 4.5 to construct a processor. These higher-level components become our building blocks in all future discussions.

5.2 The Components of a Computer System

There are a huge number of computer systems on the market, manufactured by dozens of different vendors. There are $50 million supercomputers, $1 million mainframes, midsize systems, workstations, laptops, tiny handheld "personal digital assistants," and smart phones that cost only a few hundred

dollars. In addition to size and cost, computers also differ in speed, memory capacity, input/output capabilities, and available software. The hardware marketplace is diverse, multifaceted, and ever changing.

However, in spite of all these differences, virtually every computer in use today is based on a single design. Although a $1 million mainframe and a $1,000 laptop may not seem to have much in common, they are both based on the same fundamental principles.

The same thing is true of automotive technology. Although a pickup truck, family sedan, and Ferrari racing car do not seem very similar, "under the hood" they are all constructed from the same basic technology: a gasoline-powered internal combustion engine turning an axle that turns the wheels. Differences among various models of trucks and cars are not basic theoretical differences but simply variations on a theme, such as a bigger engine, a larger carrying capacity, or a more luxurious interior.

The structure and organization of virtually all modern computers are based on a single theoretical model of computer design called the **Von Neumann architecture,** named after the brilliant mathematician John Von Neumann who proposed it in 1946. (You read about Von Neumann and his enormous contributions to computer science in the historical overview in Section 1.4.)

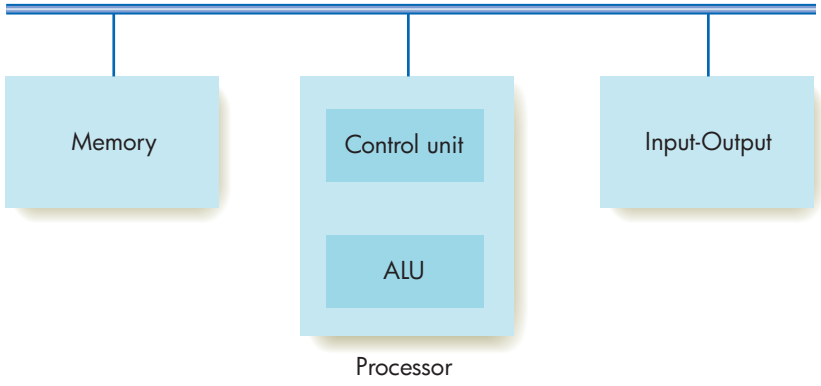
The Von Neumann architecture is based on the following three characteristics:

- Four major subsystems called **memory**, **input/output**, the **arithmetic/logic unit (ALU)**, and the **control unit**. These four subsystems are diagrammed in Figure 5.2.
- The **stored program concept**, in which the instructions to be executed by the computer are represented as binary values and stored in memory.
- The **sequential execution of instructions**, in which one instruction at a time is fetched from memory and passed to the control unit, where it is decoded and executed.

This section looks individually at each of the four subsystems that make up the Von Neumann architecture and describes their design and operation. In the following section we put all these pieces together to show the operation of the overall Von Neumann model.

FIGURE 5.2

Components of the Von Neumann Architecture



5.2.1 *Memory and Cache*

Memory is the functional unit of a computer that stores and retrieves the instructions and the data being executed. All information stored in memory is represented internally using the binary numbering system described in Section 4.2.

Computer memory uses an access technique called **random access**, and the acronym **RAM** (*r*andom *a*ccess *m*emory) is frequently used to refer to the memory unit. Random access memory has the following three characteristics:

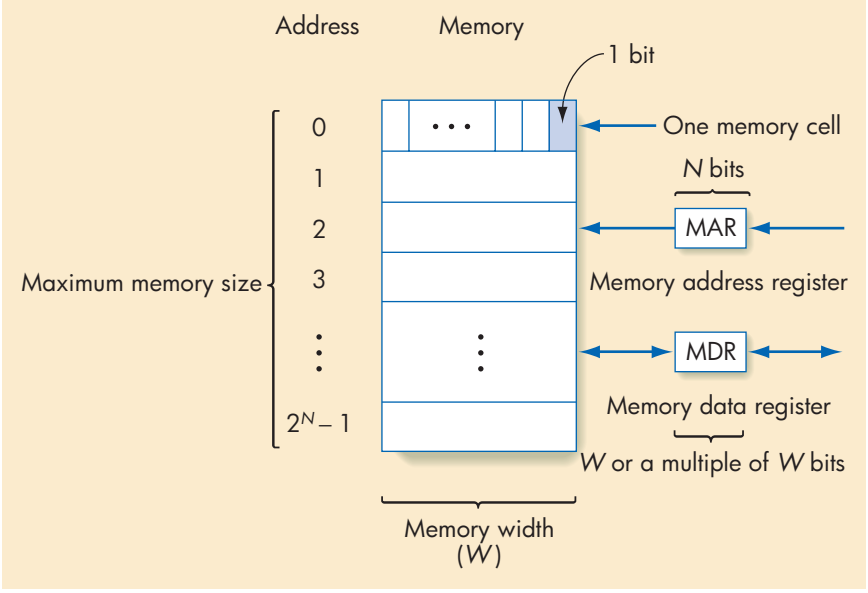
- Memory is divided into fixed-size units called **cells**, and each cell is associated with a unique identifier called an **address**. These addresses are the unsigned integers 0, 1, 2, . . . , MAX.
- All accesses to memory are to a specified address, and we must always fetch or store a complete cell—that is, all the bits in that cell. The cell is the minimum unit of access.
- The time it takes to fetch or store the contents of a cell is the same for all the cells in memory.

A model of a random access memory unit is shown in Figure 5.3. (*Note:* **Read-only memory**, abbreviated **ROM**, is random access memory into which information has been prerecorded during manufacture. This information cannot be modified or removed, only fetched. ROM is used to hold important system instructions and data in a place where a user cannot accidentally or intentionally overwrite them.)

As shown in Figure 5.3, the memory unit is made up of cells that contain a fixed number of binary digits. The number of bits per cell is called the **cell size** or the **memory width**, and it is usually denoted as *W*.

Earlier generations of computers had no standardized value for cell size, and computers were built with values of *W* = 6, 8, 12, 16, 24, 30, 32, 36, 48, and 60 bits. However, computer manufacturers now use a standard cell size of 8 bits, and this 8-bit unit is universally called a **byte**. Thus, the generic term

FIGURE 5.3
Structure of Random Access Memory



cell has become relatively obsolete, and it is more common now to refer to **memory bytes** as the basic unit. However, keep in mind that this is not a generic term but rather refers to a cell that contains exactly 8 binary digits.

With a cell size of 8 bits, the largest unsigned integer value that can be stored in a single cell is 11111111, which equals 255—not a very large number. Therefore, computers with a cell size of $W = 8$ often use multiple memory cells to store a single data value. For example, many computers use 2 or 4 bytes (16 or 32 bits) to store one integer, and either 4 or 8 bytes (32 or 64 bits) to store a single real number. This gives the range needed, but at a price. It may take several trips to memory, rather than one, to fetch a single data item.

Each memory cell in RAM is identified by a unique unsigned integer address 0, 1, 2, 3, ... If there are N bits available to represent the address of a cell, then the smallest address is 0 and the largest address is a string of N 1s:

$$\underbrace{1111\ldots11}_{N \text{ digits}}$$

which is equal to the value $2^N - 1$. Thus the range of addresses available on a computer is $[0\ .\ .\ (2^N - 1)]$, where N is the number of binary digits used to represent an address. This is a total of 2^N memory cells. The value 2^N is called the **maximum memory size** or the **address space** of the computer. Typical values of N in the 1960s and 1970s were 16, 20, 22, and 24. Today all computers have at least 32 address bits allowing for up to 2^{32}, or about 4 billion, memory bytes. However, 2^N represents the *maximum theoretical* memory size; a computer with N address bits does not necessarily come equipped with 2^N memory cells. It simply means that its memory can be expanded to 2^N. Figure 5.4 gives the value of 2^N for a number of values of N.

Because numbers like 65,536 (2^{16}) and 1,048,576 (2^{20}) are hard to remember, computer scientists use a convenient shorthand to refer to memory sizes (and other values that are powers of 2). It is based on the fact that the values 2^{10}, 2^{20}, 2^{30}, 2^{40}, and 2^{50} are quite close in magnitude to one thousand, one million, one billion, one trillion, and one quadrillion, respectively. Therefore, the letters K (kilo, or thousand), M (mega, or million), G (giga, or billion), T (tera, or trillion), and P (peta, or quadrillion) are used to refer to these units.

$2^{10} = 1\text{K}\ (= 1{,}024)$	1 KB = 1 *kilobyte*
$2^{20} = 1\text{M}\ (= 1{,}048{,}576)$	1 MB = 1 *megabyte*
$2^{30} = 1\text{G}\ (= 1{,}073{,}741{,}824)$	1 GB = 1 *gigabyte*
$2^{40} = 1\text{T}\ (= 1{,}099{,}511{,}627{,}776)$	1 TB = 1 *terabyte*
$2^{50} = 1\text{P}\ (= 1{,}125{,}899{,}906{,}842{,}624)$	1 PB = 1 *petabyte*

Thus, a computer with a 16-bit address and $2^{16} = 65{,}536$ bytes of storage would have 64 KB of memory, because $2^{16} = 2^6 \times 2^{10} = 64 \times 2^{10} = 64$ KB. This was a

FIGURE 5.4
Maximum Memory Sizes

N	Maximum Memory Size (2^N)
16	65,536
20	1,048,576
22	4,194,304
24	16,777,216
32	4,294,967,296
40	1,099,511,627,776
50	1,125,899,906,842,624

popular size for computers of the 1960s and early 1970s. Most computers today contain at least 1 GB of memory, and 2-4 GB is quite common. The 32-bit address, common in the 1980s, 1990s, and 2000s, and which supports an address space of 2^{32} = 4 GB, has reached its limits. Therefore, most processors today provide 40-, 48-, or even 64-bit addresses. A 64-bit address would allow, at least theoretically, an address space of 2^{64} bytes, or 17 billion gigabytes!

When dealing with memory, it is important to keep in mind the distinction between an **address** and the **contents** of that address.

Address	*Contents*
42	1

The address of this memory cell is 42. The content of cell 42 is the integer value 1. As you will soon see, some instructions operate on addresses, whereas others operate on the contents of an address. A failure to distinguish between these two values can cause confusion about how some instructions behave.

The two basic memory operations are **fetching** and **storing**, and they can be described formally as follows:

- *value = Fetch(address)*

  Meaning: Fetch a copy of the contents of the memory cell with the specified *address* and return those contents as the result of the operation. The original contents of the memory cell that was accessed are unchanged. This is termed a **nondestructive fetch**. Given the preceding diagram, the operation Fetch(42) returns the number 1. The value 1 remains in address 42.

- *Store(address, value)*

  Meaning: Store the specified *value* into the memory cell specified by *address*. The previous contents of the cell are lost. This is termed a **destructive store**. The operation Store(42, 2) stores a 2 in cell 42, overwriting the previous value 1.

One of the characteristics of random access memory is that the time to carry out either a fetch or a store operation is the same for all 2^N addresses. At current levels of technology, this time, called the **memory access time**, is typically about 5–10 nsec (**nanosecond** = 1 nsec = 10^{-9} sec = 1 billionth of a second). Also note that fetching and storing are allowed only to an entire cell. If we wish, for example, to modify a single bit of memory, we first need to fetch the entire cell containing that bit, change the one bit, and then store the entire cell. The cell is the minimum accessible unit of memory.

There is one component of the memory unit shown in Figure 5.3 that we have not yet discussed, the **memory registers**. These two registers are used to implement the fetch and store operations. Both operations require two operands: the *address* of the cell being accessed, and *value,* either the value stored by the store operation or the value returned by the fetch operation.

The memory unit contains two special registers whose purpose is to hold these two operands. The **Memory Address Register (MAR)** holds the address of the cell to be fetched or stored. Because the MAR must be capable of holding any address, it must be at least N bits wide, where 2^N is the address space of the computer.

Powers of 10

When we talk about volumes of information such as megabytes, gigabytes, and terabytes, it is hard to fathom exactly what those massive numbers mean. Here are some rough approximations (say, to within an order of magnitude) of how much textual information corresponds to each of the storage quantities just introduced, as well as the next few on the scale.

Quantity In Bytes	Base-10 Value	Amount of Textual Information
1 byte	10^0	One character
1 kilobyte	10^3	One typed page
1 megabyte	10^6	Two or three novels
1 gigabyte	10^9	A departmental library or a large personal library
1 terabyte	10^{12}	The library of a major academic research university
1 petabyte	10^{15}	All printed material in all libraries in North America
1 exabyte	10^{18}	All words ever printed throughout human history
1 zettabyte	10^{21}	—
1 yottabyte	10^{24}	—

The **Memory Data Register (MDR)** contains the data value being fetched or stored. We might be tempted to say that the MDR should be *W* bits wide, where *W* is the cell size. However, as mentioned earlier, on most computers the cell size is only 8 bits, and most data values occupy multiple cells. Thus the size of the MDR is usually a multiple of 8. Typical values of MDR width are 32 and 64 bits, which would allow us to fetch, in a single step, either an integer or a real value.

Given these two registers, we can describe a little more formally what happens during the fetch and store operations in a random access memory.

- Fetch(address)
  1. Load the address into the MAR.
  2. Decode the address in the MAR.
  3. Copy the contents of that memory location into the MDR.

- Store(address, value)
  1. Load the address into the MAR.
  2. Load the value into the MDR.
  3. Decode the address in the MAR.
  4. Store the contents of the MDR into that memory location.

For example, to retrieve the contents of cell 123, we would load the value 123 (in binary, of course) into the MAR and perform a fetch operation. When the operation is done, a copy of the contents of cell 123 would be in the MDR. To store the value 98 into cell 4, we load a 4 into the MAR and a 98 into the MDR and perform a store. When the operation is completed the contents of cell 4 will have been set to 98, discarding whatever was there previously.

The operation "Decode the address in the MAR" means that the memory unit must translate the *N*-bit address stored in the MAR into the set of signals needed to access that one specific memory cell. That is, the memory unit must be able to convert the integer value 4 in the MAR into the electronic signals needed to access

only address 4 from all 2^N addresses in the memory unit. This may seem like magic, but it is actually a relatively easy task that applies ideas presented in the previous chapter. We can decode the address in the MAR using a **decoder circuit** of the type described in Section 4.5 and shown in Figure 4.29. (Remember that a decoder circuit has N inputs and 2^N outputs numbered 0, 1, 2, . . . , $2^N - 1$. The circuit puts the signal 1 on the output line whose number equals the numeric value on the N input lines.) We simply copy the N bits in the MAR to the N input lines of a decoder circuit. Exactly one of its 2^N output lines is ON, and this line's identification number corresponds to the address value in the MAR.

For example, if $N = 4$ (the MAR contains 4 bits), then we have 16 addressable cells in our memory, numbered 0000 to 1111 (that is, 0 to 15). We could use a 4-to-16 decoder whose inputs are the 4 bits of the MAR. Each of the 16 output lines is associated with the one memory cell whose address is in the MAR, and enables us to fetch or store its contents. This situation is shown in Figure 5.5.

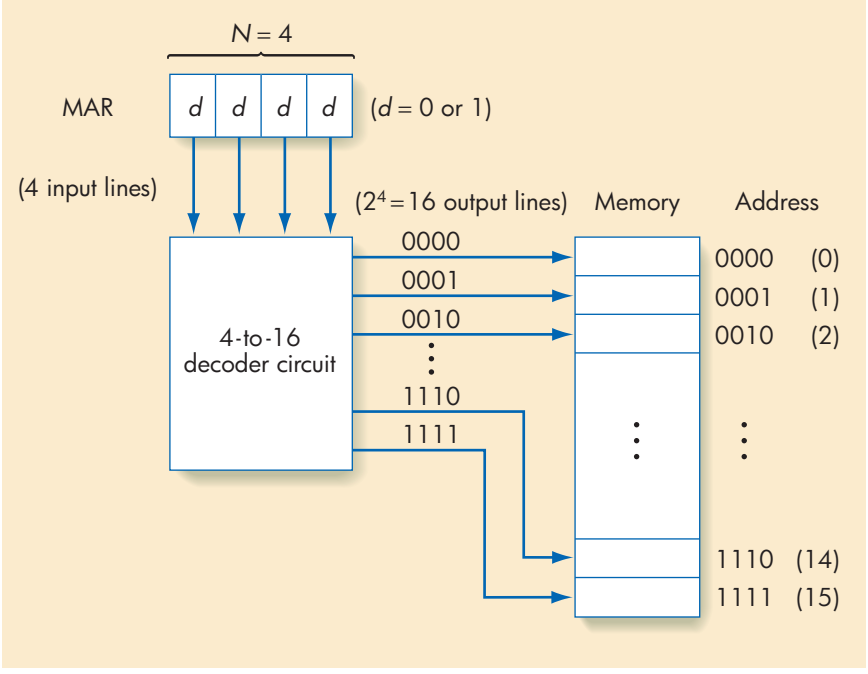
If the MAR contains the 4-bit address 0010 (decimal 2), then only the output line labeled 0010 in Figure 5.5 is ON (that is, carries a value of 1). All others are OFF. The output line 0010 is associated with the unique memory cell that has memory address 2, and the appearance of an ON signal on this line causes the memory hardware to copy the contents of location 2 to the MDR if it is doing a fetch, or to load its contents from the MDR if it is doing a store.

The only problem with the memory organization shown in Figure 5.5 is that it does not **scale** very well. That is, it cannot be used to build a large memory unit. In modern computers a typical value for N, the number of bits used to represent an address, is 32. A decoder circuit with 32 input lines would have 2^{32}, or more than 4 billion, output lines.

To solve this problem, memories are physically organized into a **two-dimensional** rather than a one-dimensional organization. In this structure, the 16-byte memory of Figure 5.5 would be organized into a two-dimensional

FIGURE 5.5

Organization of Memory and the Decoding Logic



4 × 4 structure, rather than the one-dimensional 16 × 1 organization shown earlier. This two-dimensional layout is shown in Figure 5.6.

The memory locations are stored in **row major** order, with bytes 0–3 in row 0, bytes 4–7 in row 1 (01 in binary), bytes 8–11 in row 2 (10 in binary), and bytes 12–15 in row 3 (11 in binary). Each memory cell is connected to two selection lines, one called the **row selection line** and the other called the **column selection line**. When we send a signal down a single row selection line and a single column selection line, only the memory cell located at the *intersection* of these two selection lines carries out a memory fetch or a memory store operation.

How do we choose the correct row and column selection lines to access the proper memory cell? Instead of using one decoder circuit, we use two. The first two binary digits of the addresses in Figure 5.6 are identical to the row number. Similarly, the last two binary digits of the addresses are identical to the column number. Thus, we should no longer view the MAR as being composed of a single 4-bit address, but as a 4-bit address made up of two distinct parts—the leftmost 2 bits, which specify the number of the row containing this cell, and the rightmost 2 bits, which specify the number of the column containing this cell. Each of these 2-bit fields is input to a separate decoder circuit that pulses the correct row and column lines to access the desired memory cell.

For example, if the MAR contains the 4-bit value 1101 (a decimal 13), then the two **high-order** (leftmost) bits 11 are sent to the row decoder, whereas the two **low-order** (rightmost) bits 01 are sent to the column decoder. The row decoder sends a signal on the line labeled 11 (row 3), and the column decoder sends a signal on the line labeled 01 (column 1). Only the single memory cell in row 3, column 1 becomes active and performs the fetch or store operation. Figure 5.6 shows that the memory cell in row 3, column 1 is the correct one—the cell with memory address 1101.

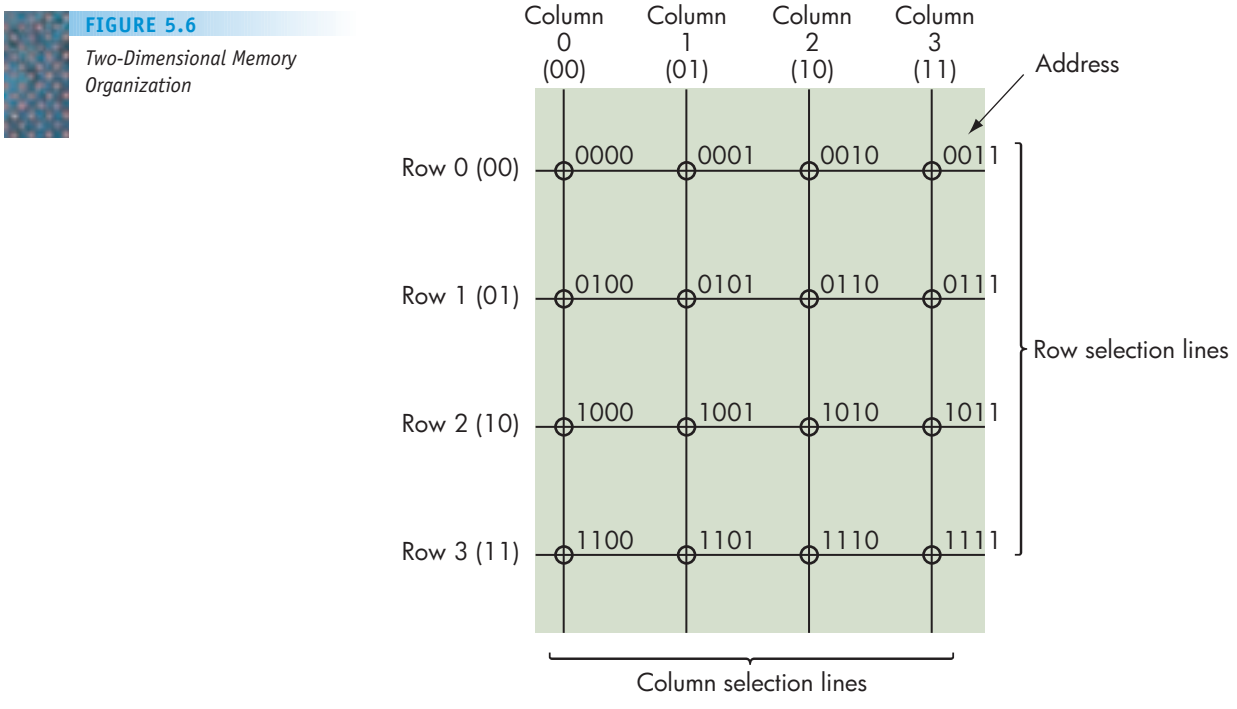


FIGURE 5.6
Two-Dimensional Memory Organization

The two-dimensional organization of Figure 5.6 is far superior to the one-dimensional structure in Figure 5.5, because it can accommodate a much larger number of cells. For example, a memory unit containing 256 MB (2^{28} bytes) is organized into a 16,384 $\times$ 16,384 two-dimensional array. To select any one row or column requires a decoder with 14 input lines (2^{14} = 16,384) and 16,384 output lines. This is a large number of output lines, but it is certainly more feasible to build two 14-to-16,384 decoders than a single 28-to-256 million decoder required for a one-dimensional organization. If necessary, we can go to a three-dimensional memory organization, in which the address is broken up into three parts and sent to three separate decoders.

To control whether memory does a fetch or a store operation, our memory unit needs one additional device called a **fetch/store controller**. This unit determines whether we put the contents of a memory cell into the MDR (a fetch operation) or put the contents of the MDR into a memory cell (a store operation). The fetch/store controller is like a traffic officer controlling the direction in which traffic can flow on a two-way street. This memory controller must determine in which direction information flows on the two-way link connecting memory and the MDR. In order to know what to do, this controller receives a signal telling it whether it is to perform a fetch operation (an F signal) or a store operation (an S signal). On the basis of the value of that signal, the controller causes information to flow in the proper direction and the correct memory operation to take place.

A complete model of the organization of a typical random access memory in a Von Neumann architecture is shown in Figure 5.7.

Let's complete this discussion by considering how complex it would be to study the memory unit of Figure 5.7, not at the abstraction level presented in that diagram, but at the gate and circuit level presented in Chapter 4. Let's assume that our memory unit contains 2^{30} cells (1 GB), each byte containing 8 bits. There is a total of about 8 billion bits of storage in this memory unit. A typical memory circuit used to store a single bit generally requires about 3 gates (1 AND, 1 OR, and 1 NOT) containing 7 transistors (3 per AND, 3 per OR, and 1 per NOT). Thus, our 1 GB memory unit (which is actually quite modest by today's standards) would contain roughly 24 billion gates and 56 billion transistors, and this does not even include the circuitry required to construct the decoder circuits, the controller, and the MAR and MDR registers! These numbers should help you appreciate the power and advantages of abstraction. Without it, studying a memory unit like the one in Figure 5.7 is a much more formidable task.

CACHE MEMORY. When Von Neumann created his idealized model of a computer, he described only a single type of memory. Whenever the computer needed an instruction or a piece of data, Von Neumann simply assumed it would get it from RAM using the fetch operation just described. However, as computers became faster, designers noticed that, more and more, the processor was sitting idle waiting for data or instructions to arrive. Processors were executing instructions so quickly that memory access was becoming a bottleneck. (It is hard to believe that a memory that can fetch a piece of data in a few billionths of a second can slow anything down, but it does.) As the following graph shows, during the period from 1980 to 2000, processors increased in performance by a factor of about 3,000, whereas memories became faster by a factor of only

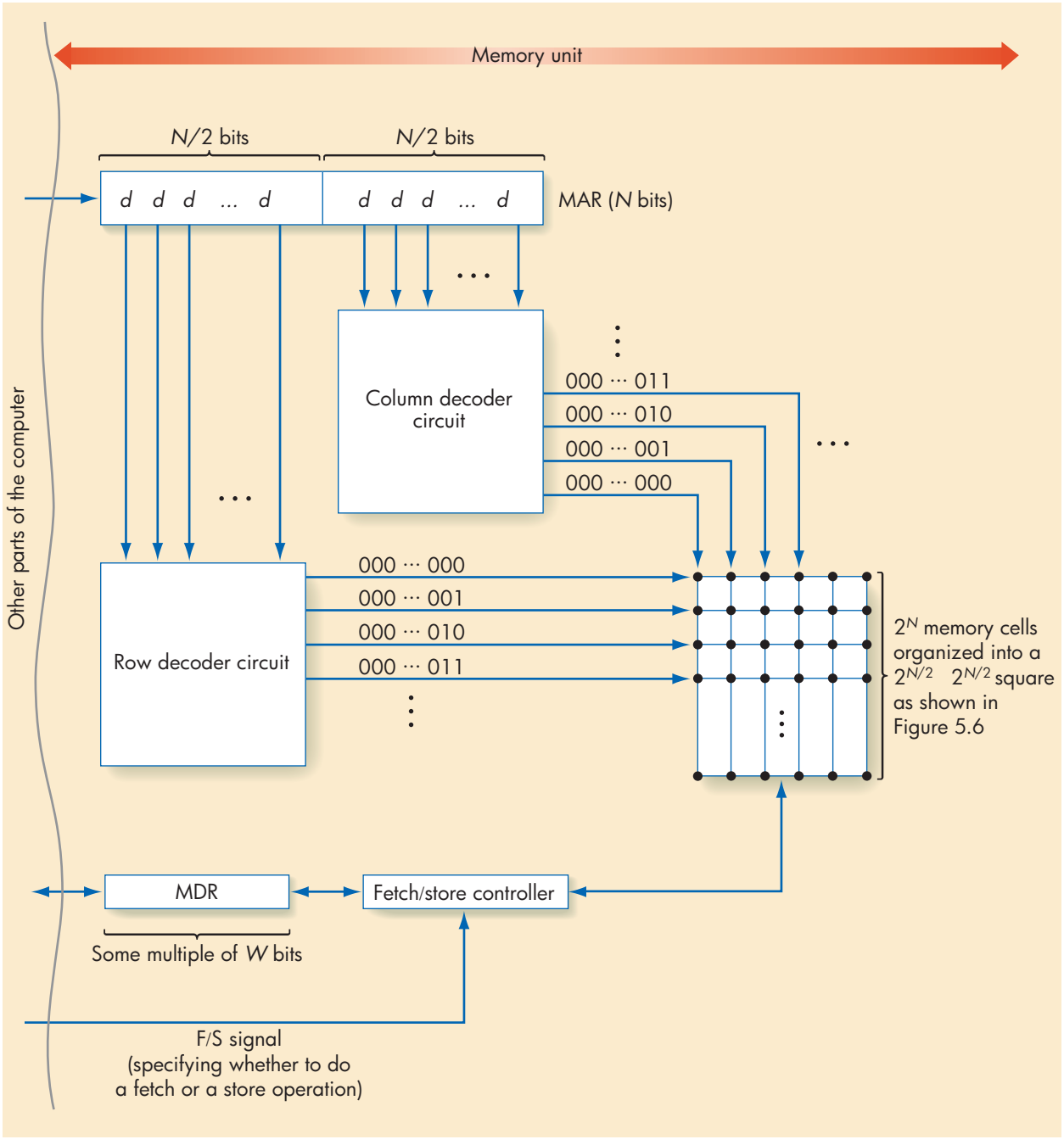
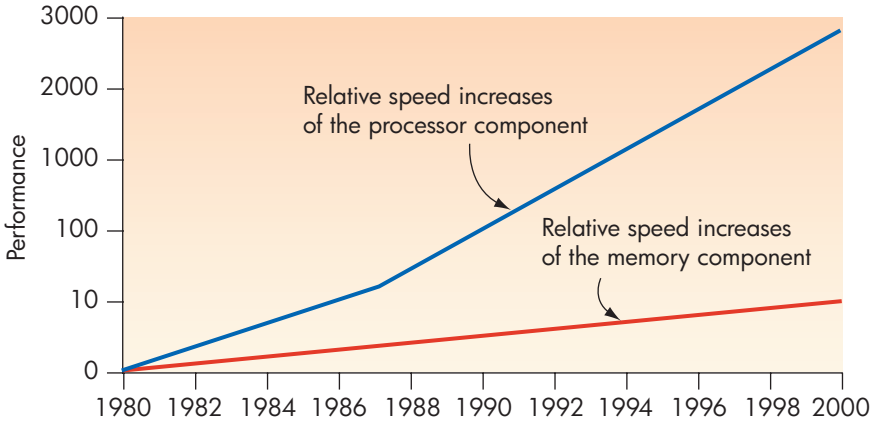


FIGURE 5.7
Overall RAM Organization

about 10.[1] This led to a huge imbalance between the capabilities of the processor and the capabilities of memory.

To solve this problem, designers needed to decrease memory access time to make it comparable with the time needed to carry out an instruction. It is possible to build extremely fast memory, but it is also quite expensive, and providing a few billion bytes or so of ultra–high-speed memory would make a computer prohibitively expensive.

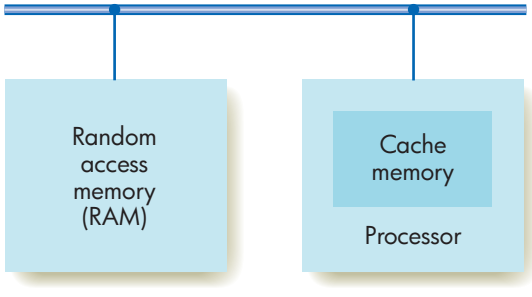
1 From *Computer Architecture: A Quantitative Approach,* 4th ed., J. Hennessy, D. Patterson, Morgan Kaufmann, 2006.



However, computer designers discovered that it is not necessary to construct *all* of memory from expensive, high-speed cells to obtain a significant increase in speed. They observed that when a program fetches a piece of data or an instruction, there is a high likelihood that

1. It will access that same instruction or piece of data in the very near future.
2. It will likely access the instructions or data that are located near that piece of data, where "near" means an address whose numerical value is close to this one.

Simply stated, this observation, called the **Principle of Locality**, says that when the computer uses something, it will probably use it again very soon, and it will probably use the "neighbors" of this item very soon. (Think about a loop in an algorithm that keeps repeating the same instruction sequence over and over.) To exploit this observation, the first time that the computer references a piece of data, it should move that data from regular RAM memory to a special, high-speed memory unit called **cache memory** (pronounced "cash," from the French word *cacher,* meaning "to hide"). It should also move the memory cells located near this item into the cache. A cache is typically 5 to 10 times faster than RAM but much smaller—on the order of a few megabytes rather than a few gigabytes. This limited size is not a problem because the computer does not keep all of the data there, just those items that were accessed most recently and that, presumably, will be needed again immediately. The organization of the "two-level memory hierarchy" is as follows:



When the computer needs a piece of information, it does not immediately do the memory fetch operation described earlier. Instead, it carries out the following three steps:

1. Look first in cache memory to see whether the information is there. If it is, then the computer can access it at the higher speed of the cache.
2. If the desired information is not in the cache, then access it from RAM at the slower speed, using the fetch operation described earlier.
3. Copy the data just fetched into the cache along with the *k* immediately following memory locations. If the cache is full, then discard some of the older items that have not recently been accessed. (The assumption is that we will not need them again for a while.)

This algorithm significantly reduces the average time to access information. For example, assume that the average access time of our RAM is 10 nsec, whereas the average access time of the cache is 2 nsec. Furthermore, assume that the information we need is in the cache 70% of the time, a value called the **cache hit rate**. In this situation, 70% of the time we get what we need in 2 nsec, and 30% of the time we have wasted that 2 nsec because the information is not in the cache and must be obtained from RAM, which will take 10 nsec. Our overall average access time will now be

$$\text{Average access time} = (0.70 \times 2) + 0.30 \times (2 + 10) = 5.0 \text{ nsec}$$

which is a 50% reduction in access time from the original value of 10 nsec. A higher cache hit rate can lead to even greater savings.

A good analogy to cache memory is a home refrigerator. Without one we would have to go to the grocery store every time we needed an item; this corresponds to slow, regular memory access. Instead, when we go to the store we buy not only what we need now but also what we think we will need in the near future, and we put those items into our refrigerator. Now, when we need something, we first check the refrigerator. If it is there, we can get it at a much higher rate of speed. We only need to go to the store when the food item we want is not there.

Caches are found on every modern computer system, and they are a significant contributor to the higher computational speeds achieved by new machines. Even though the formal Von Neumann model contained only a single memory unit, most computers built today have a multilevel hierarchy of random access memory.

PRACTICE PROBLEMS

Assume that our memory unit is organized as a 1,024 × 1,024 two-dimensional array.

1. How big does the MAR register have to be?
2. How many bits of the MAR must be sent to the row decoder? To the column decoder?

3. If the average access time of this memory is 25 nsec and the average access time for cache memory is 10 nsec, what is the overall average access time if our cache hit rate is 80%?
4. In the previous problem, what would the cache hit rate have to be to reduce the average access time to 12.0 nsec?
5. Do you think that human memory is or is not a random access memory? Give an argument why or why not.

5.2.2 *Input/Output and Mass Storage*

The **input/output (I/O)** units are the devices that allow a computer system to communicate and interact with the outside world as well as store information. The random access memory described in the previous section is **volatile** memory—the information disappears when the power is turned off. Without some type of long-term, **nonvolatile** archival storage, information could not be saved between shutdowns of the machine. Nonvolatile storage is the role of **mass storage devices** such as disks and tapes.

Of all the components of a Von Neumann machine, the I/O and mass storage subsystems are the most ad hoc and the most variable. Unlike the memory unit, I/O does not adhere to a single well-defined theoretical model. On the contrary, there are dozens of different I/O and mass storage devices manufactured by dozens of different companies and exhibiting many alternative organizations, making generalizations difficult. However, two important principles transcend the device-specific characteristics of particular vendors—**I/O access methods** and **I/O controllers**.

Input/output devices come in two basic types: those that represent information in *human-readable* form for human consumption, and those that store information in *machine-readable* form for access by a computer system. The former includes such well-known I/O devices as keyboards, screens, and printers. The latter group of devices, usually referred to as **mass storage systems**, includes floppy disks, flash memory, hard disks, CDs, DVDs, and streaming tapes. Mass storage devices themselves come in two distinct forms, **direct access storage devices (DASDs)** and **sequential access storage devices (SASDs)**.

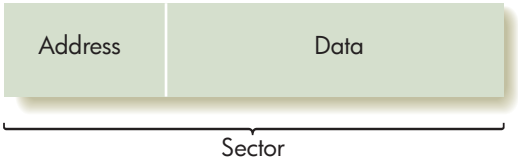
Our discussion on random access memory in Section 5.2.1 described the fundamental characteristics of random access:

1. Every memory cell has a unique address.
2. It takes the same amount of time to access every cell.

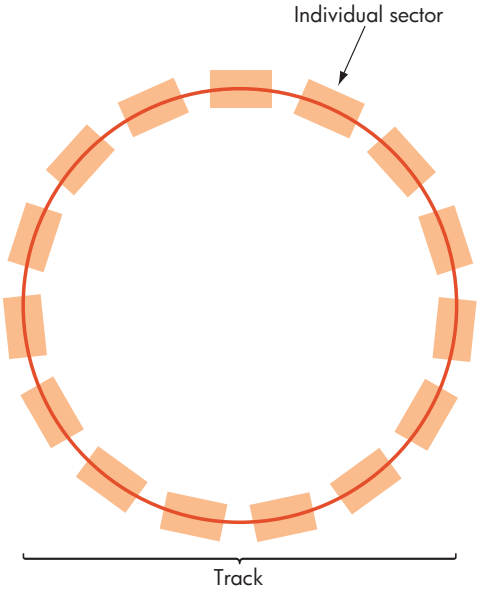
A *direct access storage device* is one in which requirement number 2, equal access time, has been eliminated. That is, in a direct access storage device, every unit of information still has a unique address, but the time needed to access that information depends on its physical location and the current state of the device.

The best examples of DASDs are the types of disks listed earlier: hard disks, floppy disks, CDs, DVDs, and so on. A disk stores information in units

called **sectors**, each of which contains an address and a data block containing a fixed number of bytes:

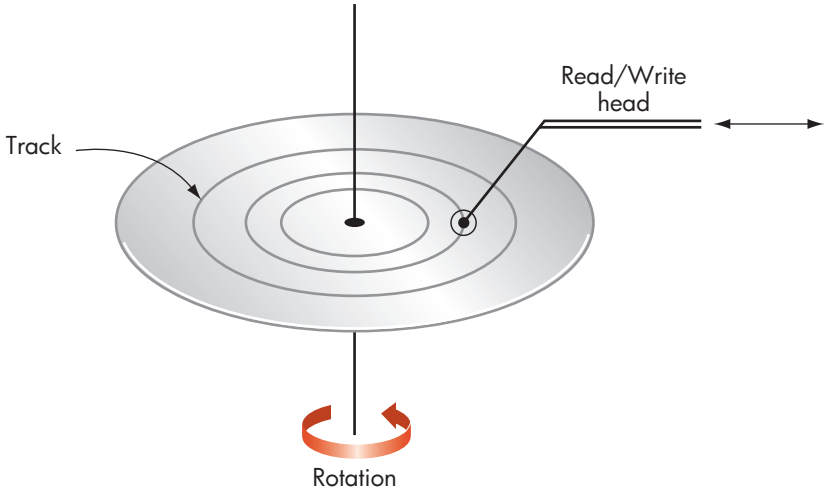


A fixed number of these sectors are placed in a concentric circle on the surface of the disk, called a **track**:



Finally, the surface of the disk contains many tracks, and there is a single **read/write head** that can be moved in or out to be positioned over any track on the disk surface. The entire disk rotates at high speed under the read/write head. The overall organization of a typical disk is shown in Figure 5.8.

FIGURE 5.8
Overall Organization of a Typical Disk



The access time to any individual sector of the disk is made up of three components: seek time, latency, and transfer time. **Seek time** is the time needed to position the read/write head over the correct track; **latency** is the time for the beginning of the desired sector to rotate under the read/write head; and **transfer time** is the time for the entire sector to pass under the read/write head and have its contents read into or written from memory. These values depend on the specific sector being accessed and the current position of the read/write head. Let's assume a disk drive with the following physical characteristics:

Rotation speed = 7,200 rev/min = 120 rev/sec = 8.33 msec/rev
(1 **msec** = 0.001 sec)

Arm movement time = 0.02 msec to move to an adjacent track (i.e., moving from track *i* to either track *i+1* or *i-1*)

Number of tracks/surface = 1,000 (numbered 0 to 999)

Number of sectors/track = 64

Number of bytes/sector = 1,024

The access time for this disk can be determined as follows.

1. Seek Time
- Best case = 0 msec (no arm movement)
- Worst case = 999 × 0.02 = 19.98 msec (move from track 0 to track 999)
- Average case = 300 × 0.02 = 6 msec (assume that on average, the read/write head must move about 300 tracks)

2. Latency
- Best case = 0 msec (sector is just about to come under the read/write head)
- Worst case = 8.33 msec (we just missed the sector and must wait one full revolution)
- Average case = 4.17 msec (one-half a revolution)

3. Transfer
- 1/64 × 8.33 msec = 0.13 msec (the time for one sector, or 1/64th of a track, to pass under the read/write head; this time will be the same for all sectors)

The following table summarizes these access time computations (all values are in milliseconds).

	Best	Worst	Average
Seek Time	0	19.98	6
Latency	0	8.33	4.17
Transfer	0.13	0.13	0.13
Total	0.13	28.44	10.3

The best-case time and the worst-case time to fetch or store a sector on the disk differ by a factor of more than 200, that is, 0.13 msec versus 28.44 msec. The average access time is about 10 msec, a typical value for current disk drive technology. This table clearly demonstrates the fundamental characteristic of all

direct access storage devices, not just disks: They enable us to specify the address of the desired unit of data and go directly to that data item, but they cannot provide a uniform access time. Today, there is an enormous range of direct access storage devices in the marketplace, from small flash memory sticks that hold a few gigabytes, to hard disks, CDs, and DVDs that can store hundreds of gigabytes, to massive online storage devices that are capable of recording and accessing terabytes or even petabytes of data. (See the "Powers of Ten" box feature in this chapter for a definition of the metric prefix *peta-*).

The second type of mass storage device uses the old access technique called **sequential access**. A sequential access storage device (SASD) does not require that all units of data be identifiable via unique addresses. To find any given data item, we must search all data sequentially, repeatedly asking the question, "Is this what I'm looking for?" If not, we move on to the next unit of data and ask the question again. Eventually we find what we are looking for or come to the end of the data.

A sequential access storage device behaves just like the old audio cassette tapes of the 1980s and 1990s. To locate a specific song, we run the tape for a while and then stop and listen. This process is repeated until we find the desired song or come to the end of the tape. In contrast, a direct access storage device behaves like a CD or DVD that numbers all the songs and allows you to select any one. (The song number is the address.) Direct access storage devices are generally much faster at accessing individual pieces of information, and that is why they are much more widely used for mass storage. However, sequential access storage devices can be useful in specific situations, such as sequentially copying the entire contents of memory or of a disk drive. This **backup** operation fits the SASD model well, and **streaming tape backup units** are common storage devices on computer systems.

One of the fundamental characteristics of many (though not all) I/O devices is that they are very, very *slow* when compared to other components of a computer. For example, a typical memory access time is about 10 nsec. The time to complete the I/O operation "locate and read one disk sector" was shown in the previous example to be about 10 msec.

Units such as nsec (billionths of a second), μsec (millionths of a second), and msec (thousandths of a second) are so small compared to human time scales that it is sometimes difficult to appreciate the immense difference between values like 10 nsec and 10 msec. The difference between these two quantities is a factor of 1,000,000, that is, 6 orders of magnitude. Consider that this is the same order of magnitude difference as between 1 mile and 40 complete revolutions of the earth's equator, or between 1 day and 30 centuries!

It is not uncommon for I/O operations such as displaying an image on a monitor or printing a page on a printer to be 3, 4, 5, or even 6 orders of magnitude slower than any other aspect of computer operation. If there isn't something in the design of a computer to account for this difference, components that operate on totally incompatible time scales will be trying to talk to each other, which will produce enormous inefficiencies. The high-speed components will sit idle for long stretches of time while they wait for the slow I/O unit to accept or deliver the desired character. It would be like talking at the normal human rate of 240 words/min (4 words/sec) to someone who could respond only at the rate of 1 word every 8 hours—a difference of 5 orders of magnitude. You wouldn't get much useful work done!

The solution to this problem is to use a device called an **I/O controller**. An I/O controller is like a special-purpose computer whose responsibility is to

handle the details of input/output and to compensate for any speed differences between I/O devices and other parts of the computer. It has a small amount of memory, called an **I/O buffer**, and enough **I/O control and logic** processing capability to handle the mechanical functions of the I/O device, such as the read/write head, paper feed mechanism, and screen display. It is also able to transmit to the processor a special hardware signal, called an **interrupt signal**, when an I/O operation is done. The organization of a typical I/O controller is shown in Figure 5.9.

Let's assume that we want to display one line (80 characters) of text on a screen. First the 80 characters are transferred from their current location in memory to the I/O buffer storage within the I/O controller. This operation takes place at the high-speed data transfer rates of most computer components—hundreds of millions of characters per second. Once this information is in the I/O buffer, the processor can instruct the I/O controller to begin the output operation. The control logic of the I/O controller handles the actual transfer and display of these 80 characters to the screen. This transfer may be at a much slower rate—perhaps only hundreds or thousands of characters per second. However, the processor does not sit idle during this output operation. It is free to do something else, perhaps work on another program. The slowness of the I/O operation now affects *only* the I/O controller. When all 80 characters have been displayed, the I/O controller sends an *interrupt signal* to the processor. The appearance of this special signal indicates to the processor that the I/O operation is finished.

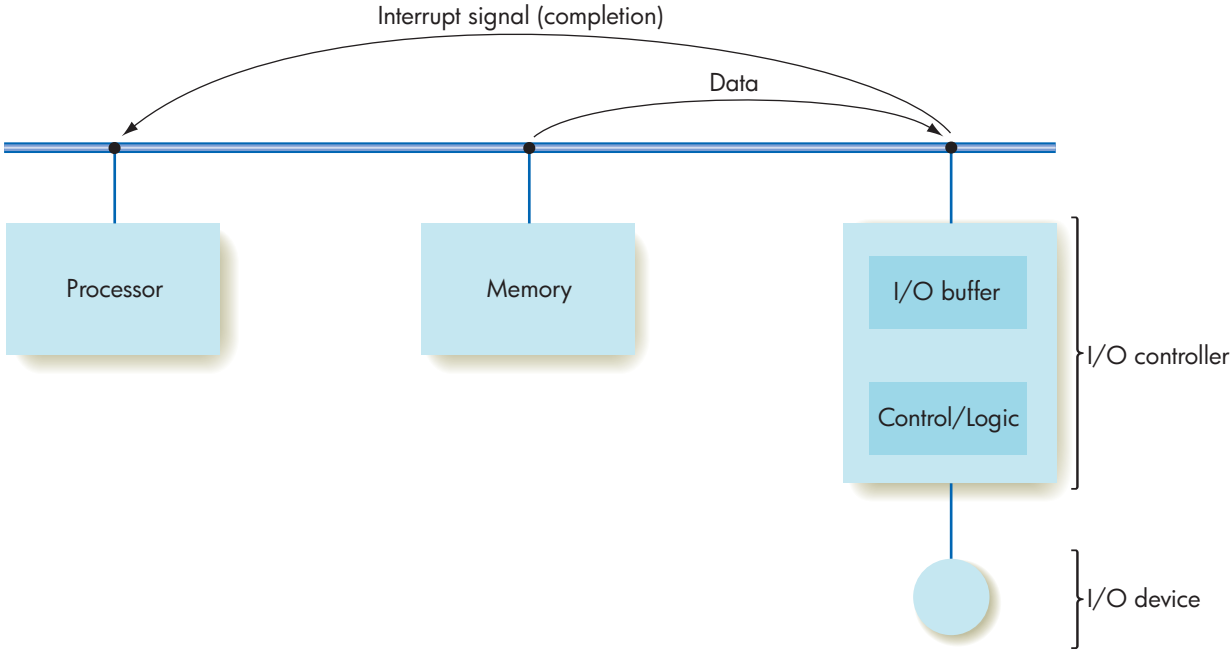


FIGURE 5.9
Organization of an I/O Controller

PRACTICE PROBLEMS

Assume a disk with the following characteristics:

Number of sectors per track = 20
Number of tracks per surface = 50
Number of surfaces = 2 (called a **double-sided** disk)
Number of characters per sector = 1,024
Arm movement time = 0.4 msec to move 1 track in any direction
Rotation speed = 2,400 rev/min

1. How many characters can be stored on this disk?
2. What are the best-case, worst-case, and average-case access times for this disk? (Assume that the average seek operation must move 20 tracks.)
3. What would be the average case access time if we could increase the rotation speed from 2,400 rev/min to 7,200 rev/min?
4. **Defragmenting** a disk means to reorganize files on the disk so that as many pieces of the file as possible are stored in sectors on the same track, regardless of the surface it is on. Explain why defragmentation can be beneficial.

5.2.3 *The Arithmetic/Logic Unit*

The **arithmetic/logic unit** (referred to by the abbreviation **ALU**) is the subsystem that performs such mathematical and logical operations as addition, subtraction, and comparison for equality. Although they can be conceptually viewed as separate components, in all modern machines the ALU and the control unit (discussed in the next section) have become fully integrated into a single component called the **processor**. However, for reasons of clarity and convenience, we will describe the functions of the ALU and the control unit separately.

The ALU is made up of three parts: the registers, the interconnections between components, and the ALU circuitry. Together these components are called the **data path**.

A **register** is a storage cell that holds the operands of an arithmetic operation and that, when the operation is complete, holds its result. Registers are quite similar to the random access memory cells described in the previous section, with the following minor differences:

- They do not have a numeric memory address but are accessed by a special **register designator** such as A, X, or R0.
- They can be accessed much more quickly than regular memory cells. Because there are few registers (typically, a few dozen up to a hundred), it is reasonable to utilize the expensive circuitry needed to make the fetch and store operations 5 to 10 times faster than regular memory cells, of which there will be hundreds of millions or billions.
- They are not used for general-purpose storage but for specific purposes such as holding the operands for an upcoming arithmetic computation.

For example, an ALU might have three special registers called A, B, and C. Registers A and B hold the two input operands, and register C holds the result. This organization is diagrammed in Figure 5.10.

In most cases, however, three registers are not nearly enough to hold all the values that we might need. A typical ALU has 16, 32, or 64 registers. To see why this many ALU registers are needed, let's take a look at what happens during the evaluation of the expression $(a / b) \times (c - d)$. After we compute the expression (a / b), it would be nice to keep this result temporarily in a high-speed ALU register while evaluating the second expression $(c - d)$. Of course, we could always store the result of (a / b) in a memory cell, but keeping it in a register allows the computer to fetch it more quickly when it is ready to complete the computation. In general, the more registers available in the ALU, the faster programs run.

A more typical ALU organization is illustrated in Figure 5.11, which shows an ALU data path containing 16 registers designated R0 to R15. Any of the 16 ALU registers in Figure 5.11 could be used to hold the operands of the computation, and any register could be used to store the result.

To perform an arithmetic operation with the ALU of Figure 5.11, we first move the operands from memory to the ALU registers. Then we specify which register holds the left operand by connecting that register to the communication path called "Left." In computer science terminology, a path for electrical signals (think of this as a wire) is termed a **bus**. We then specify which register to use for the right operand by connecting it to the bus labeled "Right." (Like RAM, registers also use nondestructive fetch so that when it is needed, the value is only copied to the ALU. It is still in the register.) The ALU is enabled to perform the desired operation, and the answer is sent to any of the 16 registers along the bus labeled "Result." (The destructive store principle says that the previous contents of the destination register will be lost.) The result can be moved from an ALU register back into memory for longer-term storage.

FIGURE 5.10

Three-Register ALU Organization

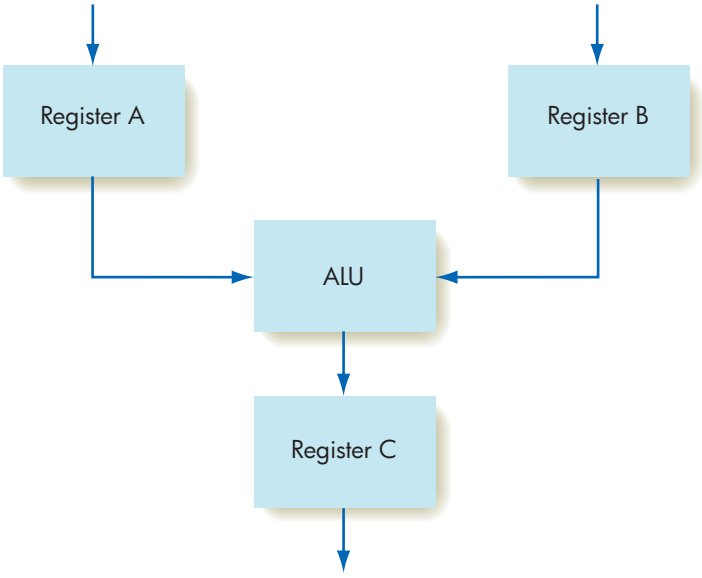
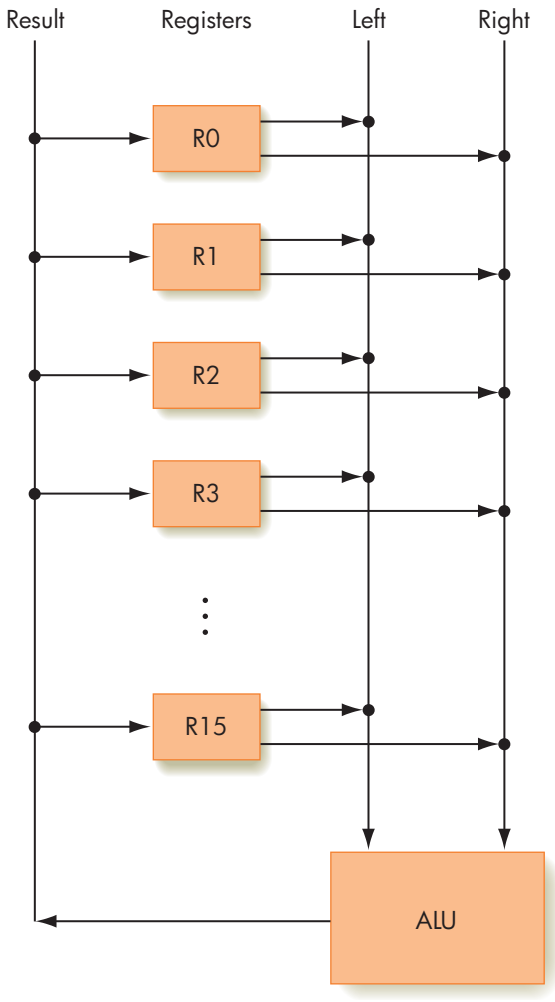


FIGURE 5.11
Multiregister ALU Organization



The final component of the ALU is the **ALU circuitry** itself. These are the circuits that carry out such operations as

$a + b$ (Figure 4.27)
$a = b$ (Figure 4.23)
$a - b$
$a \times b$
a / b
$a < b$
$a > b$
a AND b

Chapter 4 showed how circuits for these operations can be constructed from the three basic logic gates AND, OR, and NOT, and it showed the construction of logic circuits to perform the operations $a + b$ and $a = b$. The primary issue now is how to select the desired operation from among all the possibilities for a

given ALU. For example, how do we tell an ALU that can perform the preceding eight operations that we want only the results of one operation, say $a - b$?

One possibility is to use the **multiplexor** control circuit introduced in Chapter 4 and shown in Figure 4.28. Remember that a multiplexor is a circuit with 2^N input lines numbered 0 to $2^N - 1$, N selector lines, and 1 output line. The selector lines are interpreted as a single binary number from 0 to $2^N - 1$, and the input line corresponding to this number has its value placed on the single output line.

Let's imagine for simplicity that we have an ALU that can perform four functions instead of eight. The four functions are $a + b$, $a - b$, $a = b$, and *a AND b*, and these operations are numbered 0, 1, 2, and 3, respectively (00, 01, 10, and 11 in binary). Finally, let's assume that every time the ALU is enabled and given values for a and b, it automatically performs all four possible operations rather than just the desired one. These four outputs can be input to a multiplexor circuit, as shown in Figure 5.12.

Now place on the selector lines the identification number of the operation whose output we want to keep. The result of the desired operation appears on the output line, and the other three answers are discarded. For example, to select the output of the subtraction operation, we input the binary value 01 (decimal 1) on the selector lines. This places the output of the subtraction circuit on the output line of the multiplexor. The outputs of the addition, comparison, and AND circuits are discarded.

Thus, the design philosophy behind an ALU is not to have it perform only the correct operation. Instead, it is to have *every* ALU circuit "do its thing" but then keep only the one desired answer.

Putting Figures 5.11 and 5.12 together produces the overall organization of the ALU of the Von Neumann architecture. This model is shown in Figure 5.13.

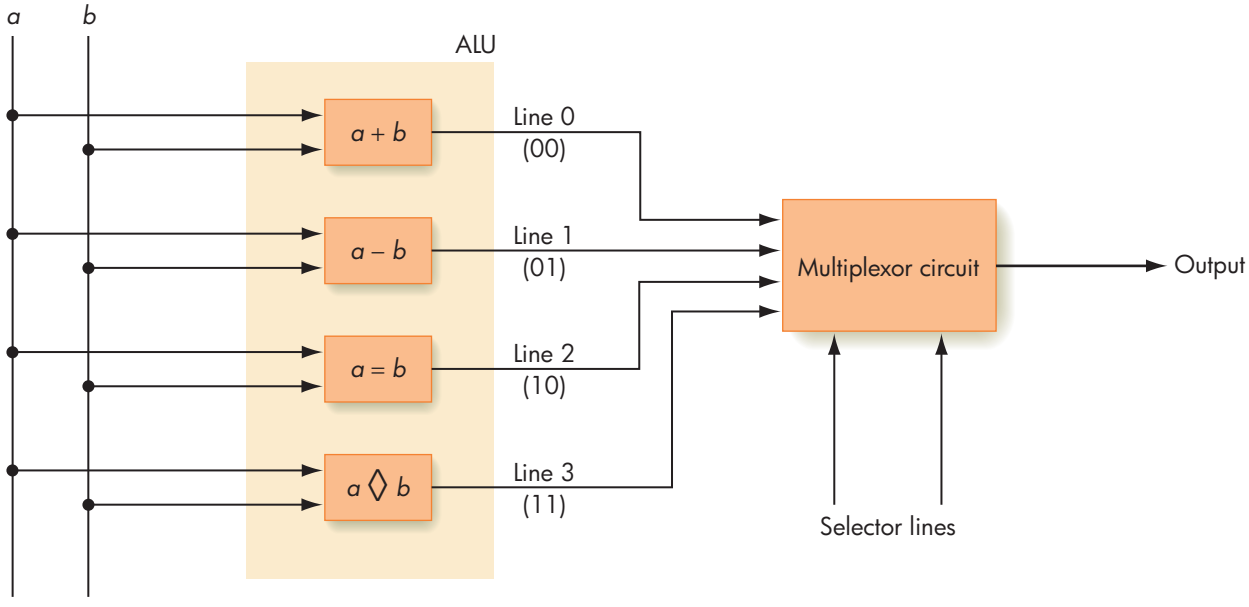
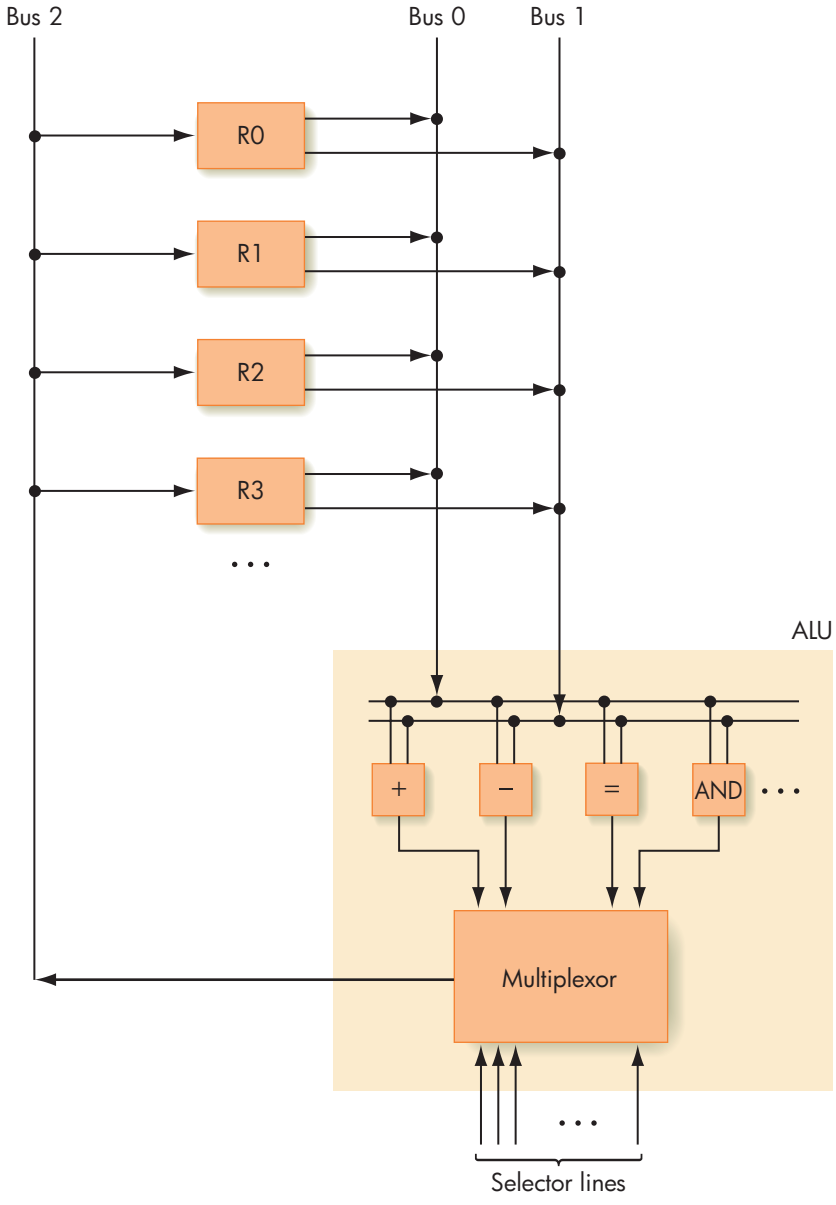


FIGURE 5.12

Using a Multiplexor Circuit to Select the Proper ALU Result

FIGURE 5.13
Overall ALU Organization



5.2.4 *The Control Unit*

The most fundamental characteristic of the Von Neumann architecture is the **stored program**—a sequence of machine language instructions stored as binary values in memory. It is the task of the **control unit** to (1) **fetch** from memory the next instruction to be executed, (2) **decode** it—that is, determine what is to be done, and (3) **execute** it by issuing the appropriate command to the ALU, memory, or I/O controllers. These three steps are repeated over and over until we reach the last instruction in the program, typically something called HALT, STOP, or QUIT.

To understand the behavior of the control unit, we must first investigate the characteristics of machine language instructions.

MACHINE LANGUAGE INSTRUCTIONS. The instructions that can be decoded and executed by the control unit of a computer are represented in **machine language**. Instructions in this language are expressed in binary, and a typical format is shown in Figure 5.14.

The **operation code** field (referred to by the shorthand phrase **op code**) is a unique unsigned integer code assigned to each machine language operation recognized by the hardware. For example, 0 could be an ADD, 1 could be a COMPARE, and so on. If the operation code field contains k bits, then the maximum number of machine language operation codes is 2^k.

The **address field(s)** are the memory addresses of the values on which this operation will work. If our computer has a maximum of 2^N memory cells, then each address field must be N bits wide to enable us to address every cell, because it takes N binary digits to represent all addresses in the range 0 to $2^N - 1$. The number of address fields in an instruction typically ranges from 0 to about 3, depending on what the operation is and how many operands it needs to do its work. For example, an instruction to add the contents of memory cell X to memory cell Y requires at least two addresses, X and Y. It could require three if the result were stored in a location different from either operand. In contrast, an instruction that tests the contents of memory cell X to see whether it is negative needs only a single address field, the location of cell X.

To see what this might produce in terms of machine language instructions, let's see what the following hypothetical instruction would actually look like when stored in memory.

Operation	Meaning
ADD X, Y	Add contents of addresses X and Y and put the sum back into Y

Let's assume that the op code for ADD is a decimal 9, X and Y correspond to memory addresses 99 and 100 (decimal), and the format of instructions is

	op code	*address 1*	*address 2*
bits →	**8**	**16**	**16**

A decimal 9, in 8-bit binary, is 00001001. Address 99, when converted to an unsigned 16-bit binary value, is 0000000001100011. Address 100 is 1 greater: 0000000001100100. Putting these values together produces the instruction ADD X, Y as it would appear in memory:

00001001	0000000001100011	0000000001100100
op code	*address 1*	*address 2*

Somewhat cryptic to a person, but easily understood by a control unit.

FIGURE 5.14
Typical Machine Language Instruction Format

The set of all operations that can be executed by a processor is called its **instruction set**, and the choice of exactly what operations to include or exclude from the instruction set is one of the most important and difficult decisions in the design of a new computer. There is no universal agreement on this issue, and the instruction sets of processors from different vendors may be completely different. This is one reason why a computer that uses a Macintosh Power Mac G5 processor cannot directly execute programs written for a system that contains an Intel Pentium 4 Dual Core. The operation codes and address fields that these two processors recognize are different and completely incompatible.

The machine language operations on most machines are quite elementary, and each operation typically performs a very simple task. The power of a processor comes not from the sophistication of the operations in its instruction set, but from the fact that it can execute each instruction very quickly, typically in a few billionths of a second.

One approach to designing instruction sets is to make them as small and as simple as possible, with as few as 30–50 instructions. Machines with this sort of instruction set are called **reduced instruction set computers** or **RISC machines**. This approach minimizes the amount of hardware circuitry (gates and transistors) needed to build a processor. The extra space on the chip can be used to optimize the speed of the instructions and allow them to execute very quickly. A RISC processor may require more instructions to solve a problem (because the instructions are so simple), but this is compensated for by the fact that each instruction runs much faster so the overall running time is less. The opposite philosophy is to include a much larger number, say 300–500, of very powerful instructions in the instruction set. These types of processors are called **complex instruction set computers**, or **CISC machines**, and they are designed to directly provide a wide range of powerful features so that finished programs for these processors are shorter. Of course, CISC machines are more complex, more expensive, and more difficult to build. As is often the case in life, it turns out that compromise is the best path—most modern processors use a mix of the two design philosophies.

A little later in this chapter we will present an instruction set for a hypothetical computer to examine how machine language instructions are executed by a control unit. For clarity, we will not show these instructions in binary, as we did earlier. Instead, we will write out the operation code in English (for example, ADD, COMPARE, MOVE), use the capital letters X, Y, and Z to symbolically represent binary memory addresses, and use the letter R to represent an ALU register. Remember, however, that this notation is just for convenience. All machine language instructions are stored internally using binary representation.

Machine language instructions can be grouped into four basic classes called data transfer, arithmetic, compare, and branch.

1. *Data Transfer* These operations move information between or within the different components of the computer—for example:

   Memory cell → ALU register

   ALU register → memory cell

   One memory cell → another memory cell

   One ALU register → another ALU register

All data transfer instructions follow the nondestructive fetch/destructive store principle described earlier. That is, the contents of the **source cell** (where it is now) are never destroyed, only copied. The contents of the **destination cell** (where it is going) are overwritten, and its previous contents are lost.

Examples of data transfer operations include

Operation	Meaning
LOAD X	Load register R with the contents of memory cell X.
STORE X	Store the contents of register R into memory cell X.
MOVE X,Y	Copy the contents of memory cell X into memory cell Y.

2. *Arithmetic* These operations cause the arithmetic/logic unit to perform a computation. Typically, they include arithmetic operations like +, −, ×, and /, as well as logical operations such as AND, OR, and NOT. Depending on the instruction set, the operands may reside in memory or they may be in an ALU register.

   Possible formats for arithmetic operations include the following examples. (*Note*: The notation CON(X) means the contents of memory address X.)

Operation	Meaning
ADD X,Y, Z	Add the contents of memory cell X to the contents of memory cell Y and put the result into memory cell Z. This is called a **three-address instruction**, and it performs the operation CON(Z) = CON(X) + CON(Y)
ADD X,Y	Add the contents of memory cell X to the contents of memory cell Y. Put the result back into memory cell Y. This is called a **two-address instruction**, and it performs the operation CON(Y) = CON(X) + CON(Y)
ADD X	Add the contents of memory cell X to the contents of register R. Put the result back into register R. This is called a **one-address instruction,** and it performs the operation R = CON(X) + R. (Of course, R must be loaded with the proper value before executing the instruction.)

   Other arithmetic operations such as SUBTRACT, MULTIPLY, DIVIDE, AND, and OR would be structured in a similar fashion.

3. *Compare* These operations compare two values and set an indicator on the basis of the results of the compare. Most Von Neumann machines have a special set of bits inside the processor called **condition codes** (or a special register called a **status register** or **condition register**); these bits are set by the compare operations. For example, assume there are three 1-bit condition codes called GT, EQ, and LT that stand for greater than, equal to, and less than, respectively. The operation

Operation	Meaning
COMPARE X,Y	Compare the contents of memory cell X to the contents of memory cell Y and set the condition codes accordingly.

   would set these three condition codes in the following way:

Condition	How the Condition Codes Are Set
CON (X) > CON (Y)	GT = 1 EQ = 0 LT = 0
CON (X) = CON (Y)	GT = 0 EQ = 1 LT = 0
CON (X) < CON (Y)	GT = 0 EQ = 0 LT = 1

4. *Branch* These operations alter the normal sequential flow of control. The normal mode of operation of a Von Neumann machine is *sequential*. After completing the instruction in address *i,* the control unit executes the instruction in address $i + 1$. (*Note:* If each instruction occupies k memory cells rather than 1, then after finishing the instruction starting in address *i,* the control unit executes the instruction starting in address $i + k$. In the following discussions, we assume for simplicity that each instruction occupies one memory cell.) The **branch instructions** disrupt this sequential mode.

   Typically, determining whether to branch is based on the current settings of the condition codes. Thus, a branch instruction is almost always preceded by either a compare instruction or some other instruction that sets the condition codes. Typical branch instructions include

Operation	Meaning
JUMP X	Take the next instruction unconditionally from memory cell X.
JUMPGT X	If the GT indicator is a 1, take the next instruction from memory cell X. Otherwise, take the next instruction from the next sequential location.
(JUMPEQ and JUMPLT would work similarly on the other two condition codes.)	
JUMPGE X	If *either* the GT or the EQ indicator is a 1, take the next instruction from memory location X. Otherwise, take the next instruction from the next sequential location.
(JUMPLE and JUMPNEQ would work in a similar fashion.)	
HALT	Stop program execution. Don't go on to the next instruction.

These are some of the typical instructions that a Von Neumann computer can decode and execute. The second challenge question at the end of this chapter asks you to investigate the instruction set of a real processor found inside a modern computer and compare it with what we have described here.

The instructions presented here are quite simple and easy to understand. The power of a Von Neumann computer comes not from having thousands of complex built-in instructions but from the ability to combine a great number of simple instructions into large, complex programs that can be executed extremely quickly. Figure 5.15 shows examples of how these simple machine language instructions can be combined to carry out some of the high-level algorithmic operations first introduced in Level 1 and shown in Figure 2.9. (The examples assume that the variables *a, b,* and *c* are stored in memory locations 100, 101, and 102, respectively, and that the instructions occupy one cell each and are located in memory locations 50, 51, 52, ...)

Don't worry if these "mini-programs" are a little confusing. We treat the topic of machine language programming in more detail in the next chapter. For now, we simply want you to know what machine language instructions look like so that we can see how to build a control unit to carry out their functions.

PRACTICE PROBLEMS

Assume that the variables *a, b, c,* and *d* are stored in memory locations 100, 101, 102, and 103, respectively. Using any of the sample machine language instructions given in this section, translate the following

pseudocode operations into machine language instruction sequences. Have your instruction sequences begin in memory location 50.

1. Set a to the value $b + c + d$
2. Set a to the value $(b \times d) - (c / d)$
3. If $(a = b)$ then set c to the value of d
4. If $(a \leq b)$ then

   Set c to the value of d

   Else

   Set c to the value of $2d$ (that is, $d + d$)
5. Initialize a to the value d

   While $a \leq c$

   Set a to the value $(a + b)$

   End of the loop

CONTROL UNIT REGISTERS AND CIRCUITS. It is the task of the control unit to fetch and execute instructions of the type shown in Figures 5.14 and 5.15. To accomplish this task, the control unit relies on two special registers called the **program counter (PC)** and the **instruction register (IR)** and on an **instruction decoder circuit**. The organization of these three components is shown in Figure 5.16.

The program counter holds the address of the *next* instruction to be executed. It is like a "pointer" specifying which address in memory the control unit must go to in order to get the next instruction. To get that instruction, the control unit sends the contents of the PC to the MAR in memory and executes the Fetch(address) operation described in Section 5.2.1. For example, if the PC holds the value 73 (in binary, of course), then when the current instruction is finished, the control unit sends the value 73 to the MAR and fetches the instruction contained in cell 73. The PC gets incremented by 1 after each fetch, because the normal mode of execution in a Von Neumann machine is sequential. (Again, we are assuming that each instruction occupies one cell. If an instruction occupied k cells, then the PC would be incremented by k.) Therefore, the PC frequently has its own incrementor (+1) circuit to allow this operation to be done quickly and efficiently.

The instruction register (IR) holds a copy of the instruction fetched from memory. The IR holds both the op code portion of the instruction, abbreviated IR_{op}, and the address(es), abbreviated IR_{addr}.

To determine what instruction is in the IR, the op code portion of the IR must be decoded using an **instruction decoder**. This is the same type of decoder circuit discussed in Section 4.5 and used in the construction of the memory unit (Figure 5.7). The k bits of the op code field of the IR are sent to the instruction decoder, which interprets them as a numerical value between 0 and $2^k - 1$. Exactly one of the 2^k output lines of the decoder is set

FIGURE 5.15

Examples of Simple Machine Language Instruction Sequences

Address	Contents
100	Value of *a*
101	Value of *b*
102	Value of *c*

Algorithmic notation	Machine Language Instruction Sequences		
	Address	Contents	(Commentary)
		⋮	
1. Set *a* to the value *b* + *c*	50	LOAD 101	Put the value of *b* into register R.
	51	ADD 102	Add *c* to register R. It now holds *b* + *c*.
	52	STORE 100	Store the contents of register R into *a*.
2. If *a* > *b* then	50	COMPARE 100, 101	Compare *a* and *b* and set condition codes.
set *c* to the value *a*	51	JUMPGT 54	Go to location 54 if *a* > *b*.
Else	52	MOVE 101, 102	Get here if *a* ≤ *b*, so move *b* into *c*
set *c* to the value *b*	53	JUMP 55	and skip the next instruction.
	54	MOVE 100, 102	Move *a* into *c*.
	55	• • •	Next statement begins here.

to a 1—specifically, the output line whose identification number matches the operation code of this instruction.

Figure 5.17 shows a decoder that accepts a 3-bit op code field and has $2^3 = 8$ output lines, one for each of the eight possible machine language operations. The three bits of the IR_{op} are fed into the instruction decoder, and they are interpreted as a value from 000 (0) to 111 (7). If the bits are, for example, 000, then line 000 in Figure 5.17 is set to a 1. This line enables the ADD operation because the operation code for ADD is 000. When a 1 appears on this line, the ADD operation: (1) fetches the two operands of the add and sends them to the ALU, (2) has the ALU perform all of its possible operations, (3) selects the output of the adder circuit, discarding all others, and (4) moves the result of the add to the correct location.

If the op code bits are 001 instead, then line 001 in Figure 5.17 is set to a 1. This time the LOAD operation is enabled, because the operation code for LOAD is

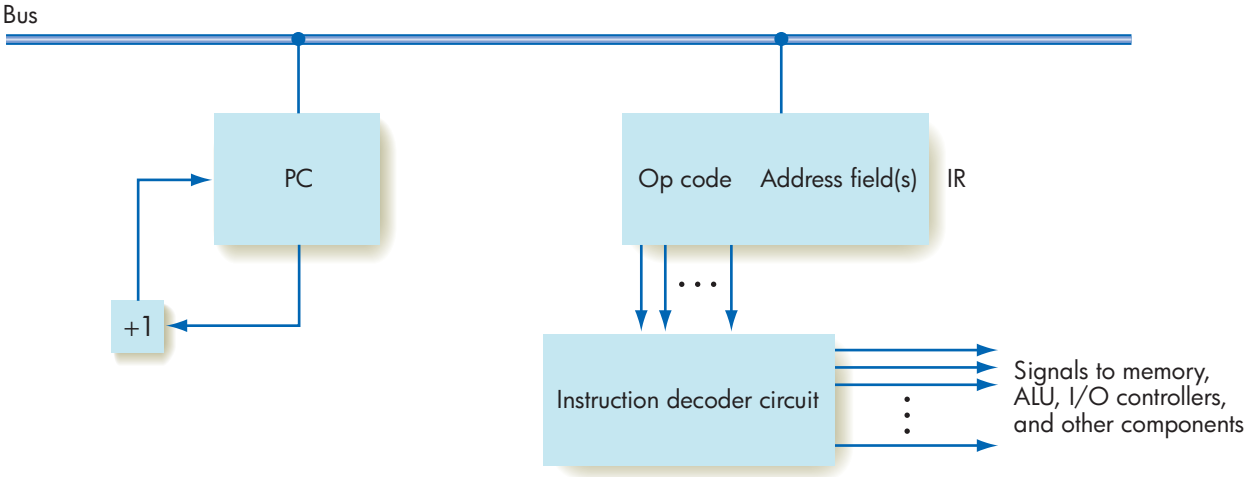


FIGURE 5.16

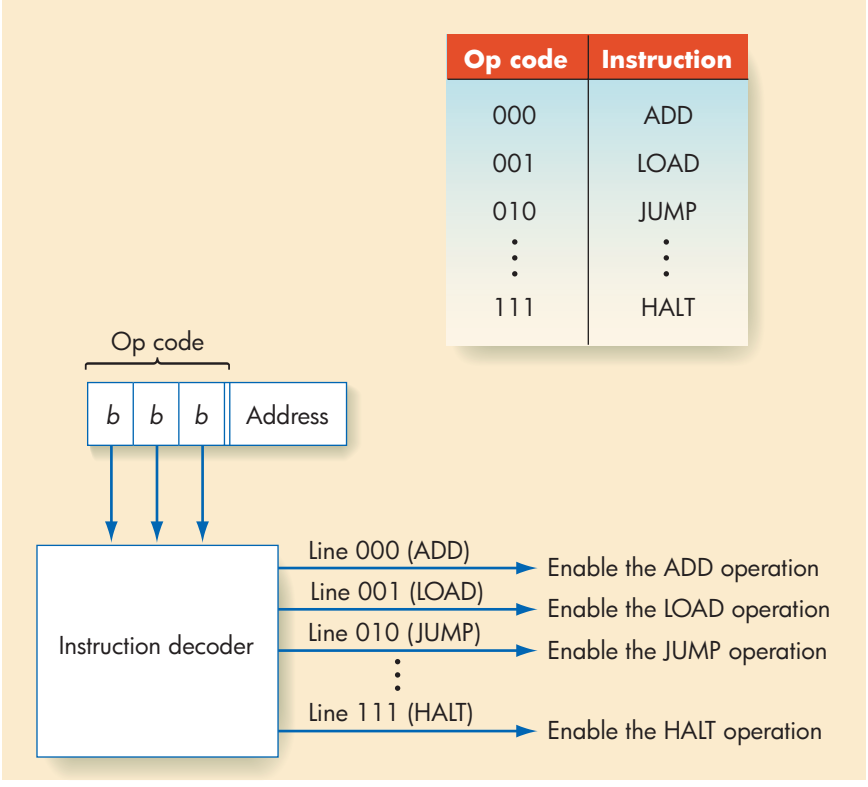
Organization of the Control Unit Registers and Circuits

the binary value 001. Instead of performing the previous four steps, the hardware carries out the LOAD operation by: (1) sending the value of IR_{addr} to the MAR in the memory unit, (2) fetching the contents of that address and putting it in the MDR, and (3) copying the contents of the MDR into ALU register R.

For every one of the 2^k machine language operations in our instruction set, there exists the circuitry needed to carry out, step by step, the function of that operation. The instruction decoder has 2^k output lines, and each output line enables the circuitry that performs the desired operation.

FIGURE 5.17

The Instruction Decoder



5.3 Putting All the Pieces Together—the Von Neumann Architecture

We have now described each of the four components that make up the Von Neumann architecture:

- Memory (Figure 5.7)
- Input/output (Figure 5.9)
- ALU (Figure 5.13)
- Control unit (Figures 5.16, 5.17)

This section puts these pieces together and shows how the entire model functions. The overall organization of a Von Neumann computer is shown in Figure 5.18. Although highly simplified, the structure in this diagram is quite similar to virtually every computer ever built!

To see how the Von Neumann machine of Figure 5.18 executes instructions, let's pick a hypothetical instruction set for our system, as shown in Figure 5.19. We will use the same instruction set in the laboratory experience for this chapter and again in Chapter 6 when we introduce and study assembly languages. (*Reminder*: CON(X) means the contents of memory cell X; R stands for an ALU register; and GT, EQ, and LT are condition codes that have the value of 1 for ON and 0 for OFF.)

The execution of a program on the computer shown in Figure 5.18 proceeds in three distinct phases: **fetch**, **decode**, and **execute**. These three steps are repeated for every instruction, and they continue until either the computer executes a HALT instruction or there is a fatal error that prevents it from continuing (such as an illegal op code, a nonexistent memory address, or division by zero). Algorithmically the process can be described as follows:

```
While we do not have a HALT instruction or a fatal error
    Fetch phase
    Decode phase
    Execute phase
End of the loop
```

This repetition of the fetch/decode/execute phase is called the *Von Neumann cycle*. To describe the behavior of our computer during each of these three phases, we will use the following notational conventions:

CON(A) The *con*tents of memory cell A. We assume that an instruction occupies 1 cell.

A → B Send the value stored in register A to register B. The following abbreviations refer to the special registers and functional units of the Von Neumann architecture introduced in this chapter:

PC	The program counter
MAR	The memory address register
MDR	The memory data register
IR	The instruction register, which is further divided into IR_{op} and IR_{addr}

	ALU	The arithmetic/logic unit
	R	Any ALU register
	GT, EQ, LT	The condition codes of the ALU
	+1	A special increment unit attached to the PC
FETCH	Initiate a memory fetch operation (that is, send an F signal on the F/S control line of Figure 5.18).	
STORE	Initiate a memory store operation (that is, send an S signal on the F/S control line of Figure 5.18).	
ADD	Instruct the ALU to select the output of the adder circuit (that is, place the code for ADD on the ALU selector lines shown in Figure 5.18).	

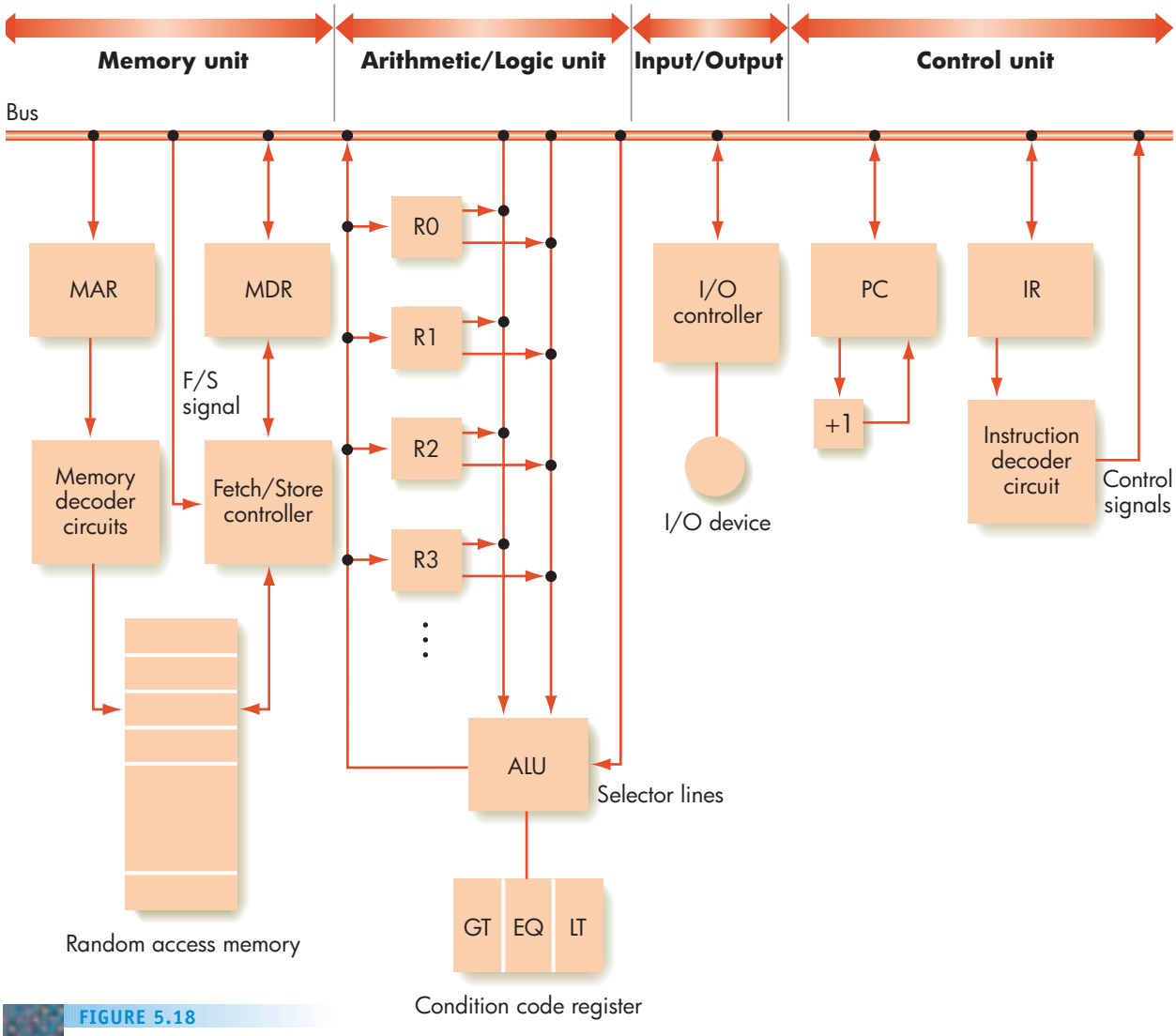


FIGURE 5.18
The Organization of a Von Neumann Computer

FIGURE 5.19

Instruction Set for Our Von Neumann Machine

Binary Op Code	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON(X)
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	if CON(X) > R then GT = 1 else 0 if CON(X) = R then EQ = 1 else 0 if CON(X) < R then LT = 1 else 0
1000	JUMP X	Get the next instruction from memory location X.
1001	JUMPGT X	Get the next instruction from memory location X if GT = 1.
1010	JUMPEQ X	Get the next instruction from memory location X if EQ = 1.
1011	JUMPLT X	Get the next instruction from memory location X if LT = 1.
1100	JUMPNEQ X	Get the next instruction from memory location X if EQ = 0.
1101	IN X	Input an integer value from the standard input device and store into memory cell X.
1110	OUT X	Output, in decimal notation, the value stored in memory cell X.
1111	HALT	Stop program execution.

SUBTRACT Instruct the ALU to select the output of the subtract circuit (that is, place the code for SUBTRACT on the ALU selector lines shown in Figure 5.18).

A. *Fetch Phase* During the fetch phase, the control unit gets the next instruction from memory and moves it into the IR. The fetch phase is the same for every instruction and consists of the following four steps.

1. PC → MAR — Send the address in the PC to the MAR register.
2. FETCH — Initiate a fetch operation using the address in the MAR. The contents of that cell are placed in the MDR.
3. MDR → IR — Move the instruction in the MDR to the instruction register so that we are ready to decode it during the next phase.
4. PC + 1 → PC — Send the contents of the PC to the incrementor and put it back. This points the PC to the next instruction.

The control unit now has the current instruction in the IR and has updated the program counter so that it will correctly fetch the next instruction when the execution of this instruction is completed. It is ready to begin decoding and executing the current instruction.

B. *Decode Phase* Decoding the instruction is simple because all that needs to be done is to send the op code portion of the IR to the instruction

decoder, which determines its type. The op code is the 4-bit binary value in the first column of Figure 5.19.

1. $IR_{op} \rightarrow$ instruction decoder

The instruction decoder generates the proper control signals to activate the circuitry to carry out the instruction.

C. *Execution Phase* The specific actions that occur during the execution phase are different for each instruction in the instruction set. The control unit circuitry generates the necessary sequence of control signals and data transfer signals to the other units (ALU, memory, and I/O) to carry out the instruction. The following examples show what signals and transfers take place during the execution phase of some of the instructions in Figure 5.19 using the Von Neumann model of Figure 5.18.

a) LOAD X	Meaning: Load register R with the current contents of memory cell X.
1. $IR_{addr} \rightarrow MAR$	Send address X (currently in IR_{addr}) to the MAR.
2. FETCH	Fetch contents of cell X and place that value in the MDR.
3. MDR $\rightarrow$ R	Copy the contents of the MDR into register R.
b) STORE X	Meaning: Store the current contents of register R into memory cell X.
1. $IR_{addr} \rightarrow MAR$	Send address X (currently in IR_{addr}) to the MAR.
2. R $\rightarrow$ MDR	Send the contents of register R to the MDR.
3. STORE	Store the value in the MDR into memory cell X.
c) ADD X	Meaning: Add the contents of cell X to the contents of register R and put the result back into register R.
1. $IR_{addr} \rightarrow MAR$	Send address X (currently in IR_{addr}) to the MAR.
2. FETCH	Fetch the contents of cell X and place it in the MDR.
3. MDR $\rightarrow$ ALU	Send the two operands of the ADD to the ALU.
4. R $\rightarrow$ ALU	
5. ADD	Activate the ALU and select the output of the add circuit as the desired result.
6. ALU $\rightarrow$ R	Copy the selected result into the R register.
d) JUMP X	Meaning: Jump to the instruction located in memory location X.
1. $IR_{addr} \rightarrow PC$	Send address X to the PC so the instruction stored there is fetched during the next fetch phase.

e) COMPARE X — Meaning: Determine whether CON(X) < R, CON(X) = R, or CON(X) > R, and set the condition codes GT, EQ, and LT to appropriate values. (Assume all codes are initially 0.)

1. $IR_{addr} \rightarrow MAR$ — Send address X to the MAR.
2. FETCH — Fetch the contents of cell X and place it in the MDR.
3. MDR → ALU — Send the contents of address X and register R to the ALU.
4. R → ALU
5. SUBTRACT — Evaluate CON(X) – R. The result is not saved, and is used only to set the condition codes. If CON(X) – R > 0, then CON(X) > R and set GT to 1. If CON(X) – R = 0, then they are equal and set EQ to 1. If CON(X) – R < 0, then CON(X) < R and set LT to 1.

f) JUMPGT X — Meaning: If GT condition code is 1, jump to the instruction in location X. We do this by loading the address field of the IR, which is the address of location X, into the PC. Otherwise, continue to the next instruction.

1. IF GT = 1 THEN $IR_{addr} \rightarrow PC$

These are six examples of the sequence of signals and transfers that occur during the execution phase of the fetch/decode/execute cycle. There is a unique sequence of actions for each of the 16 instructions in the sample instruction set of Figure 5.19 and for the 50–300 instructions in the instruction set of a typical Von Neumann computer. When the execution of one instruction is done, the control unit fetches the next instruction, starting the cycle all over again. That is the fundamental sequential behavior of the Von Neumann architecture.

These six examples clearly illustrate the concept of abstraction at work. In Chapter 4 we built complex arithmetic/logic circuits to do operations like addition and comparison. Using these circuits, this chapter describes a computer that can execute machine language instructions such as ADD X and COMPARE X,Y. A machine language instruction such as ADD X is a complicated concept, but it is quite a bit easier to understand than the enormously detailed full adder circuit shown in Figure 4.27, which contains 800 gates and more than 2,000 transistors.

Abstraction has allowed us to replace a complex sequence of gate-level manipulations with the single machine language command ADD, which does addition without our having to know how—the very essence of abstraction. Well, why should we stop here? Machine language commands, though better than hardware, are hardly user-friendly. (Some might even call them "user-intimidating.") Programming in binary and writing sequences of instructions such as

```
010110100001111010100001
```

is cumbersome, confusing, and very error prone. Why not take these machine language instructions and make them more user-oriented and user-friendly? Why not give them features that allow us to write correct, reliable, and efficient programs more easily? Why not develop **user-oriented programming languages** designed for people, not machines? This is the next level of abstraction in our hierarchy, and we introduce that important concept in Level 3 of the text.

An Alphabet Soup of Speed Measures: MHz, GHz, MIPS, and GFLOPS

It is easy to identify the fastest car, plane, or train—just compare their top speeds in miles/hour (or km/hr) and pick the greatest. However, in the computer arena things are not so simple, and there are many different measures of speed.

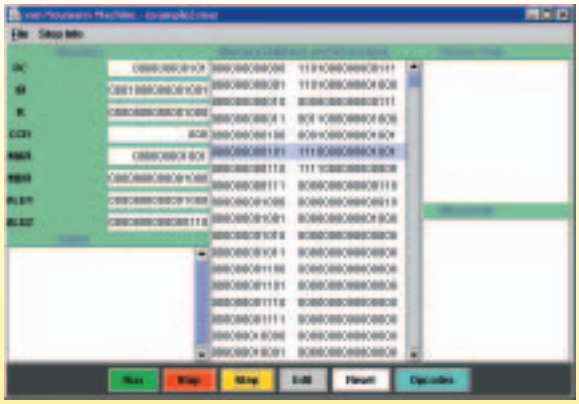
The unit you may be most familiar with is *clock speed,* measured in either millions of cycles per second, called megahertz (MHz) or billions of cycles per second, called gigahertz (GHz). The actions of every computer are controlled by a central clock, and the "tick" rate of this clock is one possible speed measure. Processors today have clock rates of up to 5 GHz. However, clock speed can be misleading, because a machine's capability depends not only on the tick rate but also on how much work it can do during each tick. If machine A has a clock rate twice as fast as machine B, but each instruction on machine A takes twice as many clock cycles as machine B to complete, then there is no discernable speed difference.

Therefore, a more accurate measure of machine speed is *instruction rate,* measured in MIPS, an acronym for *m*illions of *i*nstructions *p*er *s*econd. The instruction rate measures how many machine language instructions of the type listed in Figure 5.19 (e.g., LOAD, STORE, COMPARE, ADD) can be fetched, decoded, and executed in one second. If a computer completes one instruction for every clock cycle, then the instruction rate is identical to the clock rate. However, many instructions require multiple clock ticks, whereas parallel computers can often complete multiple instructions in a single tick. Thus, MIPS is a better measure of performance because it tells you how much work is actually being done, in terms of completed instructions, in a given amount of time.

Finally, some people are only interested in how fast a computer executes the subset of instructions most important to their applications. For example, scientific programs do an enormous amount of floating-point (i.e., decimal) arithmetic, so the computers that execute these programs must be able to execute arithmetic instructions as fast as possible. For these machines, a better measure of speed might be the *floating-point instruction rate,* measured in GFLOPS—for billions of floating-point operations per second. This is like MIPS, except the instructions we focus most closely on are those for adding, subtracting, multiplying, and dividing real numbers. Modern processors can perform at a rate of about 1-5 GFLOPS.

There is no universal measure of computer speed, and that is what allows different computer vendors all to stand up and claim, My machine is the fastest!

LABORATORY EXPERIENCE 9



This laboratory experience introduces a software package that simulates the behavior of a Von Neumann computer. It will give you a chance to work with and observe the behavior of a Von Neumann machine quite similar to the one shown in Figure 5.18. Our simulated computer contains the same functional units introduced in this section, including memory, registers, arithmetic/logic unit, and control unit, and it uses the instruction set shown in Figure 5.19. The simulator allows you to observe the step-by-step execution of machine language instructions and watch the flow of information that occurs during the fetch, decode, and execute phases. It also allows you to write and execute your own machine language programs. Shown here is a typical example of what you will see when you run this laboratory.

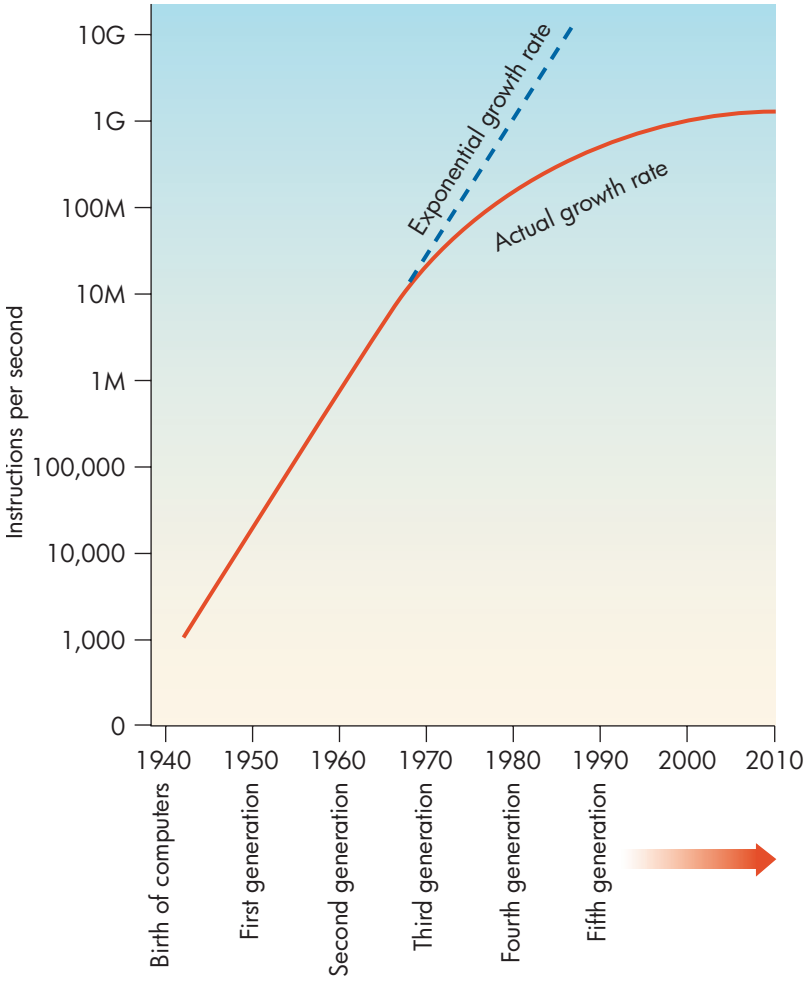
5.4 Non–Von Neumann Architectures

The Von Neumann architecture, which is the central theme of this chapter, has served the field well for almost 60 years, but some computer scientists believe it may be reaching the end of its useful life.

The problems that computers are being asked to solve have grown significantly in size and complexity since the appearance of the first-generation machines in the late 1940s and early 1950s. Designers have been able to keep up with these larger and larger problems by building faster and faster Von Neumann machines. Through advances in hardware design, manufacturing methods, and circuit technology, computer designers have been able to take the basic sequential architecture described by Von Neumann in 1946 and improve its performance by 4 or 5 orders of magnitude. First-generation machines were able to execute about 10,000 machine language instructions per second. By the second generation, that had grown to about 1 million instructions per second. Today, even a small desktop PC can perform about 1 billion instructions per second, whereas larger and more powerful workstations can execute instructions at the rate of 2-5 billion instructions per second. Figure 5.20 shows the changes in computer speeds from the mid-1940s to the present.

FIGURE 5.20

Graph of Processor Speeds, 1945 to the Present



(*Note:* The graph shown in Figure 5.20 is logarithmic. Each unit on the vertical axis is 10 times the previous one.) The period from about 1945 to about 1970 is characterized by exponential increases in computation speed. However, as Figure 5.20 shows, even though computer speeds are still increasing, the rate of improvement appears to be slowing down.

This slowdown is due to many things. One important limit on increased processor speed is the inability to place gates any closer together on a chip. (See the box on "Moore's Law and the Limits of Chip Design" in Chapter 4.) Today's high-density chips contain tens of billions of transistors separated by distances of less than 0.000001 cm, and it is becoming exceedingly difficult (not to mention expensive) to accurately place individual components closer together. However, the time it takes to send signals between two parts of a computer separated by a given distance is limited by the fact that electronic signals cannot travel faster than the speed of light—299,792,458 meters per second. That is, when we carry out an operation such as:

$$PC \rightarrow MAR$$

The signals traveling between these two registers cannot exceed 300 million meters/sec. If, for example, these two components were separated by 1 meter, it would take signals leaving the PC about 3 nanoseconds to reach the MAR, and nothing in this universe can reduce that value except a reduction of the distance separating them.

Even while the rate of increase in the performance of newer machines is slowing down, the problems that researchers are attempting to solve are growing ever larger and more complex. New applications in such areas as computational modeling, real-time graphics, and bioinformatics are rapidly increasing the demands placed on new computer systems. (We will look at some of these applications in Level 5, Chapters 13-16.) For example, to have a computer generate and display animated images without flicker it must generate 30 new frames each second. Each frame may contain as many as 3,000 $\times$ 3,000 separate picture elements (**pixels**) whose position, color, and intensity must be individually recomputed. This means that 30 $\times$ 3,000 $\times$ 3,000 = 270,000,000 pixel computations need to be completed every second. Each of those computations may require the execution of many instructions. (Where does this point move to in the next frame? How bright is it? Is it visible or hidden behind something else?) If we assume that it requires about 100 instructions per pixel to answer these questions (a reasonable approximation), then real-time computer animation requires a computer capable of executing 270,000,000 $\times$ 100 = 27 billion instructions per second. This is beyond the abilities of current processors, which are limited to about 1-5 billion instructions per second. The inability of the sequential one-instruction-at-a-time Von Neumann model to handle today's large-scale problems is called the **Von Neumann bottleneck**, and it is a major problem in computer organization.

To solve this problem, computer engineers are rethinking many of the fundamental ideas presented in this chapter, and they are studying nontraditional approaches to computer organization called **non–Von Neumann architectures**. They are asking the question, "Is there a different way to design and build computers that can solve problems 10 or 100 or 1,000 times larger than what can be handled by today's computers?" Fortunately, the answer is a resounding, Yes!

One of the most important areas of research in these non–Von Neumann architectures is based on the following fairly obvious principle:

> *If you cannot build something to work twice as fast, do two things at once. The results will be identical.*

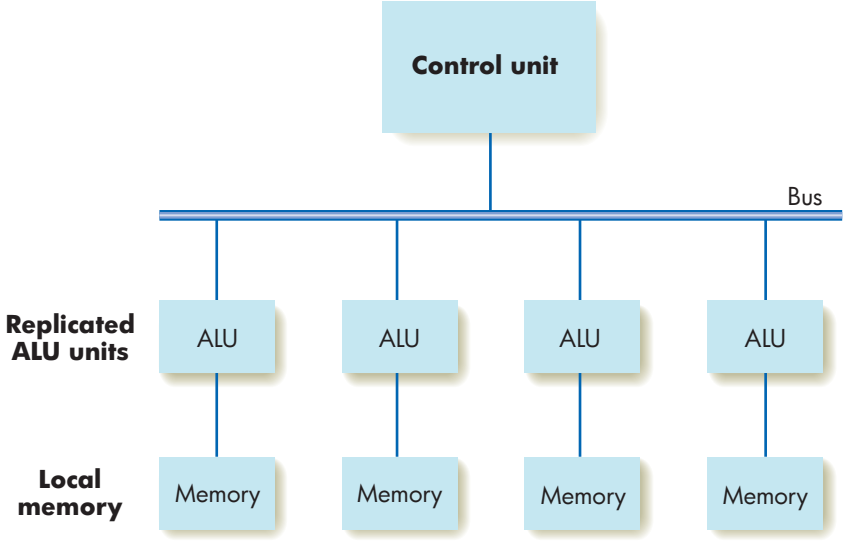
From this truism comes the principle of **parallel processing**—building computers not with one processor, as shown in Figure 5.18, but with tens, hundreds, or even thousands. If we can keep each processor occupied with meaningful work, then it should be possible to speed up the solution to large problems by 1, 2, or 3 orders of magnitude and overcome the Von Neumann bottleneck. For example, in the graphical animation example discussed earlier, we determined that we needed a machine that can execute 27 billion instructions/second, but today's processors are limited to about 2-5 billion instructions/second. But if 4 to 14 processors all work together on this one problem, then we should (in theory!) have a sufficiently powerful system to solve our problem. This is the idea behind the new **dual-core** and **quad-core** processors that have two or four separate processors on a single chip.

The approach of placing multiple processors on a single chip is fine for a small number of processors, say two, four, or eight. However, we need a completely different approach to build large-scale parallel systems that contain hundreds or even thousands of processors.

There are two fundamentally distinct approaches to designing these massively parallel systems. The first technique is termed **SIMD parallel processing**. (SIMD stands for *s*ingle *i*nstruction stream/*m*ultiple *d*ata stream.) It is diagrammed in Figure 5.21.

In the SIMD model there is a single program whose instructions are fetched/decoded/executed in a sequential manner by one control unit, exactly as described earlier. However, the ALU (circuits and registers) is replicated many times, and each ALU has its own local memory where it may keep private data. When the control unit fetches an instruction (such as a LOAD, ADD, or STORE), it **broadcasts** that instruction to every ALU, which executes it in parallel on its own local data. Thus, if we have 100 replicated ALUs, we can perform 100 parallel additions by having every ALU simultaneously execute the instruction

FIGURE 5.21
A SIMD Parallel Processing System



Operation	Meaning
ADD X	Add memory cell X to the contents of register R

on its own local value of X, using its own personal copy of register R.

A good analogy to SIMD parallel processing is the game of Bingo. There is one caller (control unit) calling out a single number (the instruction) to the entire room. In the room listening are many people (ALUs) who simultaneously cover that number on their own private Bingo cards (local memories).

This style of parallelism is especially useful in operations on mathematical structures called **vectors** and **arrays**. A vector *V* is simply an ordered collection of values. For example, here is a six-element vector *V,* whose elements are termed $v_1, v_2, \ldots, v_6$.

	v_1	v_2	v_3	v_4	v_5	v_6
V	1	8	−13	70	9	0

Many operations on vectors work quite well under the SIMD parallel model. For example, to add the constant value +1 to a vector, you add it to every individual element in the vector; that is, you simultaneously compute $v_1 + 1$, $v_2 + 1, \ldots$. Thus the operation $V + 1$, when applied to the previous vector, produces the new vector 2, 9, –12, 71, 10, 1. On a SIMD machine, this vector addition operation can be implemented in a single step by distributing one element of the vector to each separate ALU. Then in parallel, each arithmetic unit executes the following instruction:

Operation	Meaning
INC v	v is an element of the vector V. This instruction increments the contents of that location by +1.

In one time unit, we can update all six elements of the vector *V*. In the traditional Von Neumann machine, we would have to increment each element separately in a sequential fashion, using six instructions:

```
INC v1
INC v2
⋮
INC v6
```

Our parallel vector addition operator runs six times as fast. Similar speedups are possible with other vector and array manipulations.

SIMD parallelism was the first type of parallel processing put into widespread commercial use. It was the technique used to achieve breakthroughs in computational speeds on the first **supercomputers** of the early 1980s.

A much more interesting and much more widely used form of parallelism is called **MIMD parallel processing** (*m*ultiple *i*nstruction stream/*m*ultiple *d*ata stream), also called **cluster computing**. In MIMD parallelism we replicate

Speed to Burn

The first computer to achieve a speed of 1 million floating-point operations per second, 1 **megaflop**, was the Control Data 6600 in the mid-1960s. The first machine to achieve 1 billion floating-point operations per second, 1 **gigaflop**, was the Cray X-MP in the early 1980s. Today almost all machines, even small laptops, can achieve speeds of 1-5 gigaflops. In 1996, the Intel Corporation announced that its ULTRA computer had successfully become the world's first **teraflop** machine. This $55 million computer contained 9,072 Pentium Pro processors, and on December 16, 1996, it achieved a sustained computational speed of 1 trillion operations per second.

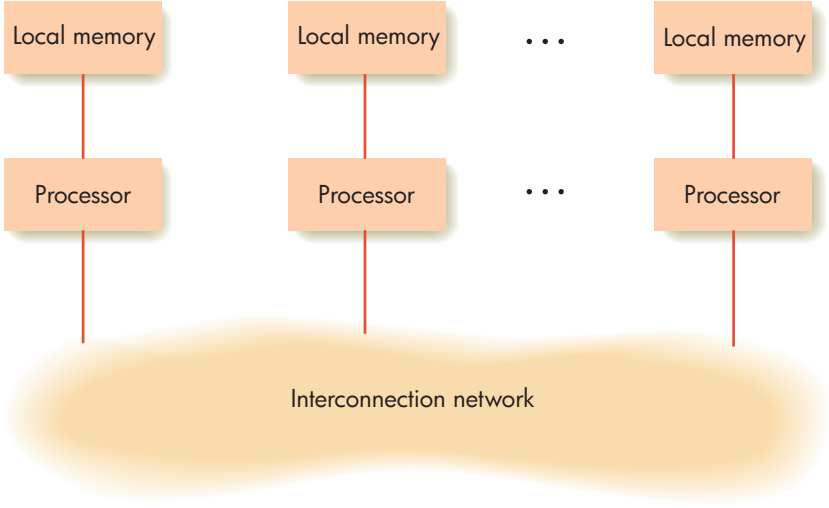
However, on June 9, 2008, a major milestone in computer performance was reached. The Roadrunner massively parallel computer, constructed jointly by Los Alamos National Laboratories and IBM, achieved a sustained computational speed of 1,026 trillion floating point operations per second, or 1 **petaflop** (see "The Tortoise and the Hare" box in Chapter 3). To get an idea of how fast that is, consider that if all 6 billion people in the world worked together on a single problem, each person would have to perform 170,000 computations/second to equal the speed of this one machine. The system, which contains 18,000 processors and 98 terabytes of memory, cost about $100 million to design and build. It will be used for basic research in astronomy, energy, and human genome science.

entire processors rather than just the ALU, and every processor is capable of executing its own separate program in its own private memory at its own rate. This model of parallel processing is diagrammed in Figure 5.22.

Each processor/memory pair in Figure 5.22 is a Von Neumann machine of the type described in this chapter. For example, it could be a processor board of the type shown in Figure 5.18. Alternately, it could be a complete computer system, such as a desktop machine in a computer lab or the laptop in your dorm room. Each system is executing its own program in its own local memory at its own rate. However, rather than each having to solve the entire problem by itself, the problem is solved in a parallel fashion by all processors simultaneously. Each of the processors tackles a small part of the overall problem and then communicates its result to the other processors via the **interconnection network**, a communications system that allows processors to exchange messages and data.

A MIMD parallel processor would be an excellent system to help us speed up the New York City telephone directory lookup problem discussed in Chapter 2.

FIGURE 5.22
Model of MIMD Parallel Processing



In the sequential approach that we described, the single processor doing the work must search all 20,000,000 entries from beginning to end (or until the desired name is found). The analysis in Chapter 3 showed that using sequential search and a computer that can examine 50,000 names per second, this lookup operation takes an average of 3.5 minutes to find a particular name—much too long for the typical person to wait.

If we use 100 processors instead of one, however, the problem is easily solved. We just divide the 20,000,000 names into 100 equal-sized pieces and assign each piece to a different processor. Now each processor searches *in parallel* to see whether the desired name is in its own section. If it finds the name, it broadcasts that information on the interconnection network to the other 99 processors so that they can stop searching. Each processor needs only to look through a list of 200,000 names, which is 1/100 the amount of work it had to do previously. Instead of an average of 3.5 minutes, we now get our answer in 1/100 the time—about 2 seconds. Parallel processing has elegantly solved our problem.

MIMD parallelism is also a scalable architecture. **Scalability** means that, at least theoretically, it is possible to match the number of processors to the size of the problem. If 100 processors are not enough to solve the telephone book lookup problem, then 200 or 500 can be used instead, assuming the interconnection network can provide the necessary communications. (Communications can become a serious bottleneck in a parallel system.) In short, the resources applied to a problem can be in direct proportion to the amount of work that needs to be done. **Massively parallel** MIMD machines containing tens of thousands of independent processors have achieved solutions to large problems thousands of times faster than is possible using a single processor. (For an up-to-date listing of the fastest parallel computers, check the home page of *Top500,* a listing of the 500 most powerful computers in the world. Its URL is *http://www.top500.org*.)

The multiple processors within a MIMD cluster do not have to be identical or belong to a single administrative organization. Computer scientists realized that it is possible to address and solve massive problems by utilizing the resources of idle computers located around the world, regardless of who they belonged to. This realization led to an exciting new form of MIMD parallelism called **grid computing**.

Grid computing enables researchers to easily and transparently access computer facilities without regard for their location. One of the most well-known grid computing applications is the SETI@home project (*S*earch for *E*xtra*t*errestrial *I*ntelligence), which analyzes radio telescope data from distant stars to look for intelligent life in the universe. Users sign up to allow their personal computer, when idle, to participate in this massive search project. About 5.5 million people have signed up to be part of the SETI grid, and on any given day about 1–2 thousand home computers, from Alabama to Wyoming, from Albania to Zimbabwe, contribute computational resources to this one task. You can read about the SETI@home project on their home page at: *http://setiathome.ssl.berkeley.edu*.

The real key to using massively parallel processors is to design solution methods that effectively utilize the large number of available processors. It does no good to have 1,000 processors available if only 1 or 2 are doing useful work while 998 or 999 are sitting idle. That would be like having a large construction crew at a building site, where the roofers, painters, and plumbers sit around waiting for one person to put up the walls. The field of **parallel algorithms**, the study of techniques that makes efficient use of parallel architectures, is an important branch of research in computer science. Advances in this area will go

Quantum Computing

The parallel machines described in this section can overcome the "Von Neumann bottleneck" by performing multiple computations in parallel, rather than one at a time in strict sequential fashion. However, these systems are not a fundamental change in the underlying design of computers. Each machine in the MIMD cluster of Figure 5.22 is typically a traditional Von Neumann computer. The only difference is that we are using multiple machines to solve a problem rather than one.

However, computer scientists are also researching totally new designs unrelated to the Von Neumann architecture. One of the most unusual, most revolutionary, and most exciting is called **quantum computing**, in which computers are designed according to the principles of quantum mechanics, which describe the behavior of matter at the atomic and sub-atomic level. A quantum computer encodes information using some aspect of quantum-mechanical state, such as electron spin or photon polarization. However, unlike "traditional" data bits, which at any instant of time must be either a 0 or a 1 but not both, quantum theory says that a quantum bit, or **qubit**, can be either a 0 or a 1 or both a 0 and a 1 *at the same time*. In theory, a quantum computer could do multiple computations on different numbers at the same time. In fact, with just 500 qubits of quantum memory, each of which could be viewed as being both a 0 and a 1, we could theoretically perform 2^{500} simultaneous computations—a number larger than the total number of atoms in the universe. Now that is real parallel processing!

There are still a lot of obstacles to overcome before quantum computers become a reality, but a great deal of progress has been made in the last few years. There is much debate about whether a workable quantum computer will take 10 years, 25 years, or perhaps another century to design and construct. However, the underlying theory of quantum computing is sound, and a quantum computer will likely be a reality, even if we are not sure exactly when.

a long way toward speeding the development and use of large-scale parallel systems of the type shown in Figures 5.21 and 5.22. (Challenge Work Exercise 1 asks you to design a parallel addition algorithm.)

To solve the complex problems of the twenty-first century, the computers of the twenty-first century will probably be organized much more like the massively parallel processing systems of Figures 5.21 and 5.22 than like the 55-year-old Von Neumann model of Figure 5.18.

5.5 Summary of Level 2

In Chapter 4 we looked at the basic building blocks of computers: binary codes, transistors, gates, and circuits. This chapter examined the standard model for computer design, called the Von Neumann architecture. It also discussed some of the shortcomings of this sequential model of computation and described briefly how these might be overcome by the use of parallel computers.

At this point in our hierarchy of abstractions, we have created a fully functional computer system capable of executing an algorithm encoded as sequences of machine language instructions. The only problem is that the machine we have created is enormously difficult to use and about as unfriendly and unforgiving as it could be. It has been designed and engineered from a machine's perspective, not a person's. Sequences of binary encoded machine language instructions such as

```
1011010000001011
1001101100010111
0000101101011001
```

give a computer no difficulty, but they cause people to throw up their hands in despair. We need to create a friendlier environment—to make the computer and its hardware resources more accessible. Such an environment would be more conducive to developing correct solutions to problems and satisfying a user's computational needs.

The component that creates this kind of friendly, problem-solving environment is called **system software**. It is an intermediary between the user and the hardware components of the Von Neumann machine. Without it, a Von Neumann machine would be virtually unusable by anyone but the most technically knowledgable computer experts. We examine system software in the next level of our investigation of computer science.

EXERCISES

1. What are the advantages and disadvantages of using a very large memory cell size, say, $W = 64$ instead of the standard size $W = 8$? If each integer occupies *one* 64-bit memory cell and is stored using sign/magnitude notation, what are the largest (in terms of absolute value) positive and negative integers that can be stored? What if *two* cells are used to store integers?
2. At a minimum, how many bits are needed in the MAR with each of the following memory sizes?
   a. 1 million bytes
   b. 10 million bytes
   c. 100 million bytes
   d. 1 billion bytes
3. A memory unit that is said to be 640 KB would actually contain how many memory cells? What about a memory of 512 MB? What about a memory of 2 GB?
4. Explain what use a read-only memory (ROM) serves in the design of a computer system. What type of information is kept in a ROM, and how does that information originally get into the memory?
5. Assuming the square two-dimensional memory organization shown in Figure 5.6, what are the dimensions of a memory containing 1 MB (2^{20}) bytes of storage? How large would the MAR be? How many bits are sent to the row and column decoders? How many output lines would these decoders have?
6. Assume a 24-bit MAR that is organized as follows:

row select lines	*column select lines*
12 bits	12 bits

   What is the maximum size of the memory unit on this machine? What are the dimensions of the memory, assuming a square two-dimensional organization?
7. Assume that our MAR contains 20 bits, enabling us to access up to 2^{20} memory cells, which is 1 MB, but our computer has 4 MB of memory. Explain how it might be possible to address all 4 MB memory cells using a MAR that contains only 20 bits.
8. Assume that a 1 gigaflop machine is connected to a printer that can print 780 characters per second. In the time it takes to print 1 page (65 lines of 60 characters per line), how many floating-point operations can the machine perform?
9. Assume that we have an arithmetic/logic unit that can carry out 20 distinct operations. Describe exactly what kind of multiplexor circuit would be needed to select exactly one of those 20 operations.
10. Assume that a hard disk has the following characteristics:

    Rotation speed = 7,200 rev/min

    Arm movement time = 0.5 msec fixed startup time + 0.05 msec for each track crossed. (The startup time is a constant no matter how far the arm moves.)

    Number of surfaces = 2 (This is a **double-sided** disk. A single read/write arm holds both read/write heads.)

    Number of tracks per surface = 500

    Number of sectors per track = 20

    Number of characters per sector = 1,024

    a. How many characters can be stored on this disk?
    b. What are the best-case, worst-case, and average-case access times for this disk?
11. In general, information is stored on a disk not at random but in specific locations that help to minimize the time it takes to retrieve that information. Using the specifications given in Exercise 10, where would you store the information in a 50 KB file on the disk to speed up subsequent access to that information?
12. Assume that our disk unit has one read/write head per *track* instead of only one per surface. (A **head-per-track disk** is sometimes referred to as a **drum**.) Using the specifications given in Exercise 10, what are the best-case, worst-case, and average-case access times? How much have the additional read/write heads helped reduce access times?
13. Discuss some situations wherein a sequential access storage device such as a tape could be a useful form of mass storage.
14. Assume that we are receiving a message across a network using a modem with a rate of 56,000 bits/second. Furthermore, assume that we are working on a workstation with an instruction rate of 500 MIPS. How many instructions can the processor execute between the receipt of each individual bit of the message?
15. Consider the following structure of the instruction register.

op code	*address-1*	*address-2*
6 bits	18 bits	18 bits

    a. What is the maximum number of distinct operation codes that can be recognized and executed by the processor on this machine?
    b. What is the maximum memory size on this machine?
    c. How many bytes are required for each operation?

16. Assume that the variables *v, w, x, y,* and *z* are stored in memory locations 200, 201, 202, 203, and 204, respectively. Using any of the machine language instructions in Section 5.2.4, translate the following algorithmic operations into their machine language equivalents.

a. Set *v* to the value of *x* – *y* + *z*. (Assume the existence of the machine language command SUBTRACT X, Y, Z that computes CON(Z) = CON(X) – CON(Y).)

b. Set *v* to the value (*w* + *x*) - (*y* + *z*)

c. If (*v* ≥ *w*) then

    set *x* to *y*

  Else

    set *x* to *z*

d. While *y* < *z* do

    Set *y* to the value (*y* + *w* + *z*)

    Set *z* to the value (*z* + *v*)

  End of the loop

17. Explain why it would be cumbersome to translate the following algorithmic operation into machine language, given only the instructions introduced in this chapter:

Set *x* to the value of *y* + 19

Can you think of a way to solve this problem?

18. Describe the sequence of operations that might go on inside the computer during the execution phase of the following machine language instructions. Use the notation shown in Section 5.2.5.

a. MOVE X, Y Move the contents of memory cell X to memory cell Y.

b. ADD X, Y Add the contents of memory cells X and Y. Put the result back into memory cell Y.

CHALLENGE WORK

1. It is easy to write a sequential algorithm that sums up a 100-element vector:

$\text{Sum} = a_1 + a_2 + a_3 + \ldots + a_{100}$

It would look something like

Set *i* to 1

Set Sum to 0

While *i* < 101 do the following

  Sum = Sum + a_i

  i = *i* + 1

End of the loop

Write out the value of Sum

Stop

It is pretty obvious that this algorithm will take about 100 units of time, where a unit of time is equivalent to the time needed to execute one iteration of the loop. However, it is less easy to see how we might exploit the existence of *multiple* processors to speed up the solution to this problem.

Assume that instead of having only a single processor, you have 100. Design a parallel algorithm that utilizes these additional resources to speed up the solution to the previous computation. Exactly how much faster would your **parallel summation algorithm** execute than the sequential one? Did you need all 100 processors? Could you have used more than 100?

2. In this chapter we described the Von Neumann architecture in broad, general terms. However, "real" Von Neumann processors, such as the Pentium 4 Dual-Core and the AMD Athlon XP, are much more complex than the simple model shown in Figure 5.18. Pick one of these processors (perhaps the processor inside the computer you are using for this class) and take an in-depth look at its design. Specifically, examine such issues as

- Its instruction set and how it compares with the instruction set shown in Figure 5.19
- The collection of available registers
- The existence of cache memory
- Its computing speed in MIPS and MFLOPS or GFLOPS
- How much primary memory it has and how memory is addressed in the instructions
- Memory access time
- In what size "chunks" memory can be accessed

Write a report describing the real-world characteristics of the processor you selected.

FOR FURTHER READING

In the area of computer organization and machine architecture:

Patterson, D., and Hennessey, J. *Computer Organization and Design: The Hardware/Software Interface,* 3rd ed. revised San Francisco: Morgan Kaufmann, 2007.

Stallings, W. *Computer Organization and Architecture,* 7th ed. Englewood Cliffs, NJ: Prentice-Hall, 2005.

Tanenbaum, A. *Structured Computer Organization,* 5th ed. Englewood Cliffs, NJ: Prentice-Hall, 2005.

A fascinating and thoroughly enjoyable nontechnical book that explores some of the issues involved in designing and building a computer is

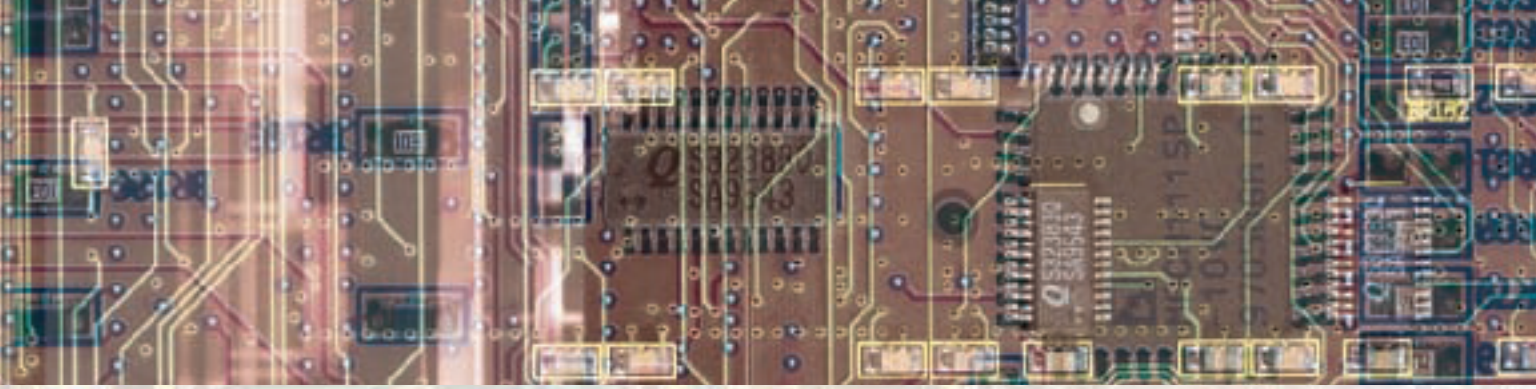
Kidder, T. *The Soul of a New Machine.* San Francisco: Back Bay Books, 2000.

In the area of parallel processing:

Dongarra, J. *Sourcebook of Parallel Computing.* San Francisco: Morgan Kaufman, 2002.

Jordan, H. F. *Fundamentals of Parallel Computing.* Englewood Cliffs, NJ: Prentice-Hall, 2002.

This page intentionally left blank



CHAPTER 6

An Introduction to System Software and Virtual Machines

6.1 Introduction

Chapters 4 and 5 described a computer model, the Von Neumann architecture, that is capable of executing programs written in machine language. This computer has all the hardware needed to solve important real-world problems, but it has no "support tools" to make that problem-solving task easy. The computer described in Chapter 5 is known as a **naked machine**: hardware bereft of any helpful user-oriented features.

Imagine what it would be like to work on a naked machine. To solve a problem, you would have to create hundreds or thousands of cryptic machine language instructions that look like this:

```
10110100110100011100111100001000
```

and you would have to do that without making a single mistake because, to execute properly, a program must be completely error-free. Imagine the likelihood of writing a perfectly correct program containing thousands of instructions like the one shown above. Even worse, imagine trying to locate an error buried deep inside that incomprehensible mass of 0s and 1s.

On a naked machine, the data as well as the instructions must be represented in binary. For example, a program cannot refer to the decimal integer +9 directly but must express it as

```
0000000000001001      (the binary representation of +9 using 16 bits)
```

You cannot use the symbol *A* to refer to the first letter of the alphabet but must represent it using its 8-bit ASCII code value, which is decimal 65:

```
01000001              (the 8-bit ASCII code for A; see Figure 4.3)
```

As you can imagine, writing programs for a naked machine is very complex.

Even if you write the program correctly, your work is still not done. A program for a Von Neumann computer must be stored in memory prior to execution. Therefore, you must now take the program and store its instructions into sequential cells in memory. On a naked machine the programmer must perform this task, one instruction at a time. Assuming that each instruction occupies one memory cell, the programmer loads the first instruction into address 0, the second instruction into address 1, the third instruction into address 2, and so on, until all have been stored.

Finally, what starts the program running? A naked machine does not do this automatically. (As you are probably coming to realize, a naked machine

does not do *anything* automatically, except fetch, decode, and execute machine language instructions.) The programmer must initiate execution by storing a 0, the address of the first instruction of the program, in the program counter (PC) and pressing the START button. This begins the fetch/decode/execute cycle described in Chapter 5. The control unit fetches from memory the contents of the address in the PC, currently 0, and executes that instruction. The program continues sequentially from that point while the user prays that everything works, because he or she cannot bear to face a naked machine again!

As this portrayal demonstrates, working directly with the underlying hardware is practically impossible for a human being. The functional units described in Chapter 5 are built according to what is easy for hardware to do, not what is easy for people to do.

To make a Von Neumann computer usable, we must create an **interface** between the user and the hardware. This interface does the following things:

- Hides from the user the messy details of the underlying hardware
- Presents information about what is happening in a way that does not require in-depth knowledge of the internal structure of the system
- Allows easy user access to the resources available on this computer
- Prevents accidental or intentional damage to hardware, programs, and data

By way of analogy, let's look at how people use another common tool—an automobile. The internal combustion engine is a complex piece of technology. For most of us, the functions of carburetors, distributors, and camshafts are a total mystery. However, most people find driving a car quite easy. This is because the driver does not have to lift the hood and interact directly with the hardware; that is, he or she does not have to drive a "naked automobile." Instead, there is an interface, the **dashboard**, which simplifies things considerably. The dashboard hides the details of engine operation that a driver does not need to know. The important things—such as oil pressure, fuel level, and vehicle speed—*are* presented in a simple, "people-oriented" way: oil indicator warning light, fuel gauge, and speed in miles or kilometers per hour. Access to the engine and transmission is achieved by a few simple operations: a key to start and stop, pedals to speed up or slow down, and a shift lever to go forward or backward.

We need a similar interface for our Von Neumann machine. This "computer dashboard" would eliminate most of the hassles of working on a naked machine and let us view the hardware resources of Chapter 5 in a much friendlier way. Such an interface does exist, and it is called system software.

6.2 System Software

6.2.1 *The Virtual Machine*

System software is a collection of computer programs that manage the resources of a computer and facilitate access to those resources. It is important to remember that we are describing software, not hardware. There are no

black boxes wired to a computer and labeled "system software." Software consists of sequences of instructions—namely, programs—that solve a problem. However, instead of solving *user* problems, such as looking up names in a telephone book, system software makes a computer and its many resources easier to access and use.

System software acts as an *intermediary* between the users and the hardware, as shown in Figure 6.1. System software presents the user with a set of services and resources across the interface labeled A in Figure 6.1. These resources may actually exist, or they may be simulated by the software to give the user the illusion that they exist. The set of services and resources created by the software and seen by the user is called a **virtual machine** or a **virtual environment**. The system software, not the user, interacts with the actual hardware (that is, the naked machine) across the interface labeled B in Figure 6.1.

The system software has the following responsibilities, analogous to those of the automobile dashboard:

- Hides the complex and unimportant (to the user) details of the internal structure of the Von Neumann architecture
- Presents important information to the user in a way that is easy to understand
- Allows the user to access machine resources in a simple and efficient way
- Provides a secure and safe environment in which to operate

For example, to add two numbers, it is much easier to use simple notation such as $a = b + c$ than to worry about (1) loading ALU registers from memory cells b and c, (2) activating the ALU, (3) selecting the output of the addition circuit, and (4) sending the result to memory cell a. The programmer should not have to know about registers, addition circuits, and memory addresses but instead should see a virtual machine that "understands" the symbols + and =.

After the program has been written, it should automatically be loaded into memory without the programmer having to specify where it should be placed or having to set the program counter. Instead, he or she should be able to issue one simple command (or one set of mouse clicks) to the virtual machine that says, Run my program. Finally, when the program is running and generating

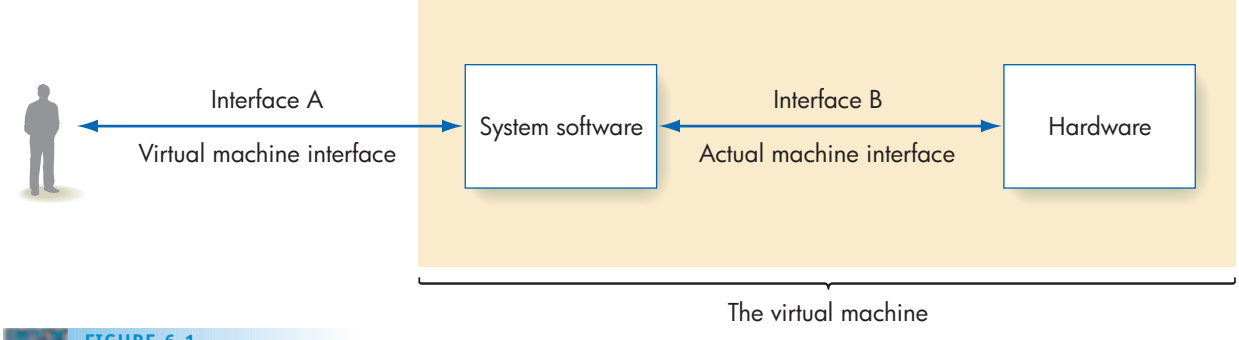


FIGURE 6.1
The Role of System Software

results, the programmer should be able to instruct the virtual machine to send the program's output to the printer in Room 105, without reference to the details related to I/O controllers, interrupt signals, and code sets.

All the useful services just described are provided by the system software available on any modern computer system. The following sections show how this friendly, user-oriented environment is created.

6.2.2 *Types of System Software*

System software is not a single monolithic entity but a collection of many different programs. The types found on a typical computer are shown in Figure 6.2.

The program that controls the overall operation of the computer is the **operating system**, and it is the single most important piece of system software on a computer. It is the operating system that communicates with users, determines what they want, and activates other system programs, applications packages, or user programs to carry out their requests. The software packages that handle these requests include:

- *User interface.* All modern operating systems provide a powerful **graphical user interface (GUI)** that gives the user an intuitive visual overview as well as graphical control of the capabilities and services of the computer.
- *Language services.* These programs, called **assemblers**, **compilers**, and **interpreters**, allow you to write programs in a high-level, user-oriented language rather than the machine language of Chapter 5 and to execute these programs easily and efficiently. They often include components such as text editors and debuggers.
- *Memory managers.* These programs allocate memory space for programs and data and retrieve this memory space when it is no longer needed.
- *Information managers.* These programs handle the organization, storage, and retrieval of information on mass storage devices such as the disks, CDs, DVDs, and tapes described in Section 5.2.2. They allow you to organize your information in an efficient hierarchical manner, using directories, folders, and files.

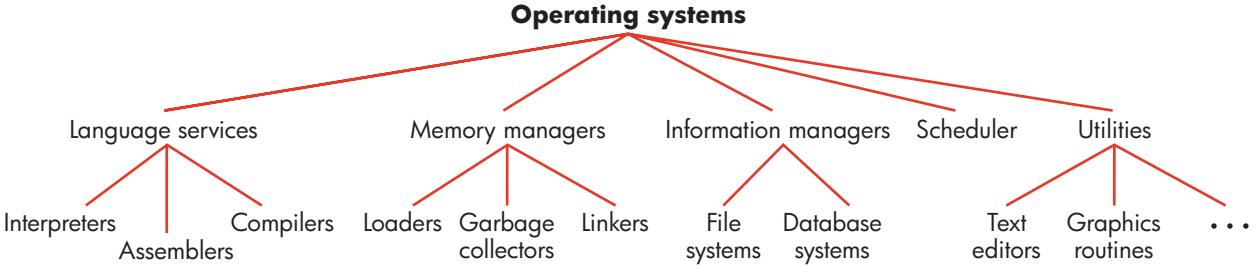


FIGURE 6.2
Types of System Software

- *I/O systems*. These software packages allow you to easily and efficiently use the many different types of input and output devices that exist on a modern computer system.
- *Scheduler*. This system program keeps a list of programs ready to run on the processor, and it selects the one that will execute next. The scheduler allows you to have a number of different programs active at a single time, for instance, to surf the Web while you are waiting for a file to finish printing.
- *Utilities*. These collections of library routines provide useful services either to a user or to other system routines. Text editors, online help routines, drawing programs, and control panels are examples of utility routines. Sometimes these utilities are organized into collections called **program libraries**.

These system routines are used during every phase of problem solving on a computer, and it would be virtually impossible to get anything done without them. Let's go back to the problem described at the beginning of this chapter—the job of writing a program, loading it into memory, running it, and printing the results. On a naked machine this job would be formidable. On the virtual machine created by system software, it is much simpler:

Step	*Task*
1	Use a *text editor* to create program P written in a high-level, English-like notation rather than binary.
2	Use the *file system* to store program P on the hard disk in your home directory.
3	Use a *language translator* to translate program P from a high-level language into a machine language program M.
4	Use the *scheduler* to load, schedule, and run program M. The scheduler itself uses the *memory manager* to obtain memory space for program M.
5	Use the *I/O system* to print the output of your program on printer R.
6	If the program did not complete successfully, use a *debugger* to help locate the error. Use the text editor to correct the program and the file system to store the newly modified program.

Furthermore, most of these operations are easily invoked via mouse clicks and the graphical user interface provided by the operating system.

On a virtual machine, the details of machine operation are no longer visible, and a user can concentrate on higher-level issues: writing the program, executing the program, and saving and analyzing results.

There are many types of system software, and it is impossible to cover them all in this section of the text. Instead, we will investigate two types of system software, and use these as representatives of the entire group. Section 6.3 examines assemblers, and Section 6.4 looks at the design and construction of operating systems. These two packages create a friendly and usable virtual machine. In Chapter 7, we extend that virtual environment from a single computer to a collection of computers by looking at the system software required to create one of the most important and widely used virtual environments—a computer network. Finally, in Chapter 8 we investigate one of the most important services provided by the operating system—system security.

6.3.1 *Assembly Language*

One of the first places where we need a more friendly virtual environment is in our choice of programming language. Machine language, which is designed from a machine's point of view, not a person's, is complicated and difficult to understand. What specifically are the problems with machine language?

- It uses binary. There are no natural language words, mathematical symbols, or other convenient mnemonics to make the language more readable.
- It allows only numeric memory addresses. A programmer cannot name an instruction or a piece of data and refer to it by name.
- It is difficult to change. If we insert or delete an instruction, all memory addresses following that instruction will change. For example, if we place a new instruction into memory location 503, then the instruction previously in location 503 is now in 504. All references to address 503 must be updated to point to 504. There may be hundreds of such references.
- It is difficult to create data. If a user wishes to store a piece of data in memory, he or she must compute the internal binary representation for that data item. These conversion algorithms are complicated and time consuming.

Programmers working on early first-generation computers quickly realized the shortcomings of machine language. They developed a new language, called **assembly language**, designed for people as well as computers. Assembly languages created a more productive, user-oriented environment, and assemblers were one of the first pieces of system software to be widely used. When assembly languages first appeared in the early 1950s, they were one of the most important new developments in programming—so important, in fact, that they were considered an entirely new generation of language, analogous to the new generations of hardware described in Section 1.4.3. Assembly languages were termed **second-generation languages** to distinguish them from machine languages, which were viewed as **first-generation languages**.

Today, assembly languages are more properly called **low-level programming languages**, which means they are closely related to the machine language of Chapter 5. Each symbolic assembly language instruction is translated into exactly *one* binary machine language instruction.

This contrasts with languages like C++, Java, and Python, which are **high-level programming languages**. High-level languages are more user oriented, they are not machine specific, and they use both natural language and mathematical notation in their design. A single high-level language instruction is typically translated into *many* machine language instructions, and the virtual environment created by a high-level language is much more powerful than the one produced by an assembly language. We discuss high-level languages in detail in Chapters 9 and 10.

Figure 6.3 shows a "continuum of programming languages," from the lowest level (closest to the hardware) to the highest level (most abstract, farthest from the hardware).

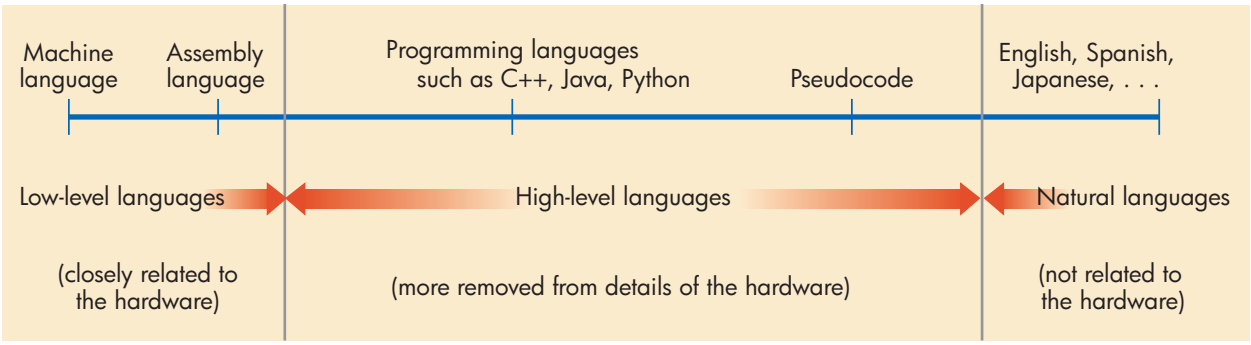


FIGURE 6.3

The Continuum of Programming Languages

The machine language of Chapter 5 is the most primitive; it is the language of the hardware itself. Assembly language, the topic of this section, represents the first step along the continuum from machine language. High-level programming languages like C++, Java, and Python are much closer in style and structure to natural languages and are quite distinct from assembly language. Natural languages, such as English, Spanish, and Japanese, are the highest level; they are totally unrelated to hardware design.

A program written in assembly language is called the **source program**; it uses the features and services provided by the language. However, the processor does not "understand" assembly language instructions, in the sense of being able to fetch, decode, and execute them as described in Chapter 5. The source program must be translated into a corresponding machine language program, called the **object program**. This translation is carried out by a piece of system software called an **assembler**. (Translators for high-level languages are called **compilers**. They are discussed in Chapter 11.) Once the object program has been produced, its instructions can be loaded into memory and executed by the processor exactly as described in Section 5.3. The complete translation/loading/execution process is diagrammed in Figure 6.4.

There are three major advantages to writing programs in assembly language rather than machine language:

- Use of symbolic operation codes rather than numeric (binary) ones
- Use of symbolic memory addresses rather than numeric (binary) ones
- Pseudo-operations that provide useful user-oriented services such as data generation

This section describes a simple, but realistic, assembly language that demonstrates these three advantages.

Our hypothetical assembly language is composed of instructions in the following format:

label: op code mnemonic address field --comment

The **comment** field, preceded in our notation by a double dash (--), is not really part of the instruction. It is a helpful explanation added to the instruction by a programmer and intended for readers of the program. It is ignored during translation and execution.

Assembly languages allow the programmer to refer to op codes using a symbolic name, called the **op code mnemonic**, rather than by a number. We can

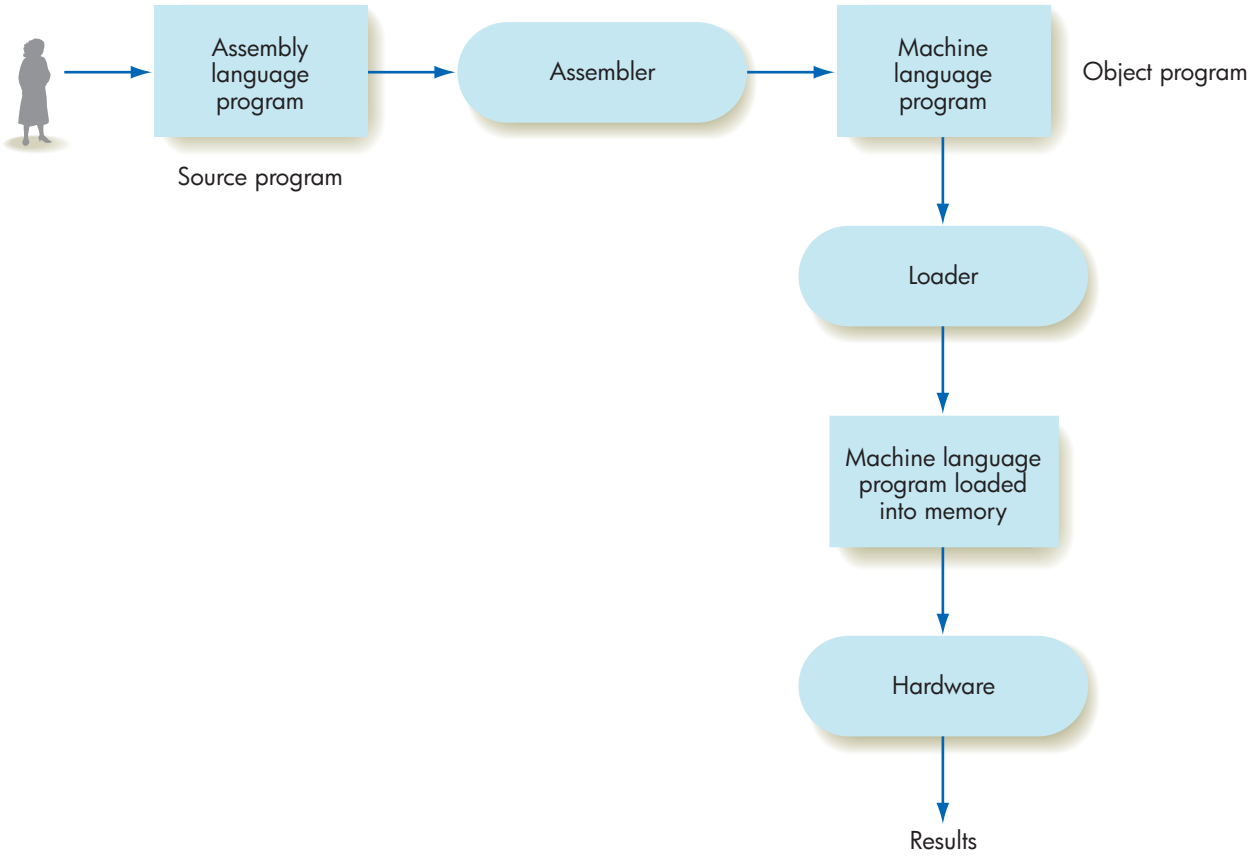


FIGURE 6.4
The Translation/Loading/ Execution Process

write op codes using meaningful words like LOAD, ADD, and STORE rather than obscure binary codes like 0000, 0011, and 0001. Figure 6.5 shows an assembly language instruction set for a Von Neumann machine that has a single ALU register R and three condition codes GT, EQ, and LT. Each numeric op code, its assembly language mnemonic, and its meaning are listed. This table is identical to Figure 5.19, which summarizes the language used in Chapter 5 to introduce the Von Neumann architecture and explains how instructions are executed. (However, Chapter 5 describes binary machine language and uses symbolic names only for convenience. In this chapter we are describing assembly language, where symbolic names such as LOAD and ADD are actually part of the language.)

Another advantage of assembly language is that it lets programmers use **symbolic addresses** instead of numeric addresses. In machine language, to jump to the instruction in location 17, you must refer directly to address 17; that is, you must write JUMP 17 (in binary, of course). This is cumbersome, because if a new instruction is inserted anywhere within the first 17 lines of the program, the jump location changes to 18. The old reference to 17 is incorrect, and the address field must be changed. This makes modifying programs very difficult, and even small changes become big efforts. It is not unlike identifying yourself in a waiting line by position—as, say, the tenth person in line. As soon as someone in front of you leaves (or someone cuts in line ahead of you), that number changes. It is far better to identify yourself using a characteristic that does not change as people enter or exit the line.

FIGURE 6.5

Typical Assembly Language Instruction Set

Binary Op Code	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON(X)
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	if CON(X) > R then GT = 1 else 0 if CON(X) = R then EQ = 1 else 0 if CON(X) < R then LT = 1 else 0
1000	JUMP X	Get the next instruction from memory location X.
1001	JUMPGT X	Get the next instruction from memory location X if GT = 1.
1010	JUMPEQ X	Get the next instruction from memory location X if EQ = 1.
1011	JUMPLT X	Get the next instruction from memory location X if LT = 1.
1100	JUMPNEQ X	Get the next instruction from memory location X if EQ = 0.
1101	IN X	Input an integer value from the standard input device and store into memory cell X.
1110	OUT X	Output, in decimal notation, the value stored in memory cell X.
1111	HALT	Stop program execution.

For example, you are the person wearing the green suit with the orange and pink shirt. Those characteristics won't change (though maybe they should).

In assembly language we can attach a symbolic **label** to any instruction or piece of data in the program. The label becomes a permanent identification for this instruction or data, regardless of where it appears in the program or where it may be moved in memory. A label is a name (followed by a colon to identify it as a label) placed at the beginning of an instruction.

```
LOOPSTART:    LOAD X
```

The label LOOPSTART has been attached to the instruction LOAD X. This means that the name LOOPSTART is *equivalent to* the address of the memory cell that holds the instruction LOAD X. If, for example, the LOAD X instruction is stored in memory cell 62, then the name LOOPSTART is equivalent to address 62. Any use of the name LOOPSTART in the address field of an instruction is treated as though the user had written the numeric address 62. For example, to jump to the load instruction shown above, we do not need to know that it is stored in location 62. Instead, we need only write the instruction

```
JUMP LOOPSTART
```

Symbolic labels have two advantages over numeric addresses. The first is **program clarity**. As with the use of mnemonics for op codes, the use of meaningful symbolic names can make a program much more readable. Names like

LOOPSTART, COUNT, and ERROR carry a good deal of meaning and help people to understand what the code is doing. Memory addresses such as 73, 147, and 2001 do not. A second advantage of symbolic labels is **maintainability**. When we refer to an instruction via a symbolic label rather than an address, we no longer need to modify the address field when instructions are added to or removed from the program. Consider the following example:

```
        JUMP    LOOP
        ⋮       ←point A
LOOP:   LOAD    X
```

Say a new instruction is added to the program at point A. When the modified program is translated by the assembler into machine language, all instructions following point A are placed in a memory cell whose address is 1 higher than it was before (assuming that each instruction occupies one memory cell). However, the JUMP refers to the LOAD instruction only by the name LOOP, not by the address where it is stored. Therefore, neither the JUMP nor the LOAD instruction needs to be changed. We need only retranslate the modified program. The assembler determines the new address of the LOAD X instruction, makes the label LOOP equivalent to this new address, and places this new address into the address field of the JUMP LOOP instruction. The assembler does the messy bookkeeping previously done by the machine language programmer.

The final advantage of assembly language programming is **data generation**. In Section 4.2.1 we showed how to represent in binary data types such as unsigned and signed integers, floating point values, and characters. When writing in machine language, the programmer must do these conversions. In assembly language, however, the programmer can ask the assembler to do them by using a special type of assembly language op code called a **pseudo-op**.

A pseudo-op (preceded in our notation by a period to indicate its type) does not generate a machine language instruction like other operation codes. Instead, it invokes a service of the assembler. One of these services is generating data in the proper binary representation for this system. There are typically assembly language pseudo-ops to generate integer, character, and (if the hardware supports it) real data values. In our example language, we will limit ourselves to one data generation pseudo-op called .DATA that builds signed integers. This pseudo-op converts the signed decimal integer in the address field to the proper binary representation. For example, the pseudo-op

```
FIVE:   .DATA   +5
```

tells the assembler to generate the binary representation for the integer +5, puts it into memory, and makes the label "FIVE" equivalent to the address of that cell. If a memory cell contains 16 bits, and the next available memory cell is address 53, then this pseudo-op produces

address	*contents*
53	0000000000000101

and the name FIVE is equivalent to memory address 53. Similarly, the pseudo-op

```
NEGSEVEN: .DATA    –7
```

might produce the following 16-bit quantity, assuming sign/magnitude representation:

address	*contents*
54	1000000000000111

and the symbol NEGSEVEN is equivalent to memory address 54.

We can now refer to these data items by their attached label. For example, to load the value +5 into register R, we can say

```
LOAD      FIVE
```

This is equivalent to writing LOAD 53, which loads register R with the contents of memory cell 53—that is, the integer +5. Note that if we had incorrectly said

```
LOAD      5
```

the *contents* of memory cell 5 would be loaded into register R. This is not what we intended, and the program would be wrong. This is a good example of why it is so important to distinguish between the address of a cell and its contents.

To add the value –7 to the current contents of register R, we write

```
ADD       NEGSEVEN
```

The contents of R (currently +5) and the contents of address NEGSEVEN (address 54, whose contents are –7) are added together, producing –2. This becomes the new contents of register R.

When generating data values, we must be careful not to place them in memory locations where they can be misinterpreted as instructions. In Chapter 4 we said that the only way a computer can tell that the binary value 01000001 is the letter *A* rather than the decimal value 65 is by the context in which it appears. The same is true for instructions and data. They are indistinguishable from each other, and the only way a Von Neumann machine can determine whether a sequence of 0s and 1s is an instruction or a piece of data is by how we use it. If we attempt to execute a value stored in memory, then that value *becomes* an instruction whether we meant it to be or not.

For example, if we incorrectly write the sequence

```
LOAD      X
.DATA     +1
```

then, after executing the LOAD X command, the processor fetches, decodes,

and attempts to execute the "instruction" +1. This may sound meaningless, but to a processor, it is not. The representation of +1, using 16 bits, is

```
0000000000000001
```

Because this value is being used as an instruction, some of the bits will be interpreted as the op code and some as the address field. If we assume a 16-bit, one-address instruction format, with the first 4 bits being the op code and the last 12 bits being the address field, then these 16 bits will be interpreted as follows:

```
0000            000000000001
op code         address
```

The "op code" is 0, which is a LOAD on our hypothetical machine (see Figure 6.5), and the "address field" contains a 1. Thus, the data value +1 has accidentally turned into the following instruction:

```
LOAD    1       --Load the contents of memory cell 1 into register R
```

PRACTICE PROBLEMS

1. Assume that register R and memory cells 80 and 81 contain the following values:

   R: 20 memory cell 80: 43 memory cell 81: 97

   Using the instruction set shown in Figure 6.5, determine what value ends up in register R and memory cells 80 and 81 after each of the following instructions is executed. Assume that each question begins with the values shown above.

   a. LOAD 80
   b. STORE 81
   c. COMPARE 80
   d. ADD 81
   e. IN 80
   f. OUT 81

2. Assume that memory cell 50 contains a 4 and label L is equivalent to memory location 50. What value does each of the following LOAD instructions load into register R?

   a. LOAD 50
   b. LOAD 4
   c. LOAD L
   d. LOAD L+1 (Assume that this is legal.)

3. Explain why both the HALT operation code described in Figure 6.5 and the .END pseudo-op mentioned at the end of this section are needed in an assembly language program and what might happen if one or both were omitted.

FIGURE 6.6

Structure of a Typical Assembly Language Program

```
.BEGIN      --This must be the first line of the program.
    ⋮       --Assembly language instructions like those in Figure 6.5.
  HALT      --This instruction terminates execution of the program
    ⋮       --Data generation pseudo-ops such as
            --.DATA are placed here, after the HALT.
.END        --This must be the last line of the program.
```

This is obviously incorrect, but how is the problem solved? The easiest way is to remember to place all data at the end of the program in a section where they cannot possibly be executed. One convenient place that meets this criterion is after a HALT instruction, because the HALT prevents any further execution. The data values can be referenced, but they cannot be executed.

A second service provided by pseudo-ops is **program construction**. Pseudo-ops that mark the beginning (.BEGIN) and end (.END) of the assembly language program specify where to start and stop the translation process, and they do not generate any instructions or data. Remember that it is the HALT instruction, not the .END pseudo-op, that terminates execution of the program. The .END pseudo-op ends the translation process. Figure 6.6, which shows the organization of a typical assembly language program, helps explain this distinction.

6.3.2 *Examples of Assembly Language Code*

This section describes how to use assembly language to translate algorithms into programs that can be executed on a Von Neumann computer. Today, software development is rarely performed in assembly language except for special-purpose tasks; most programmers use one of the higher-level languages mentioned in Figure 6.3 and described in Chapters 9 and 10. Our purpose in offering these examples is to demonstrate how system software, in this case an assembler, can create a user-oriented virtual environment that supports effective and productive problem solving.

One of the most common operations in any algorithm is the evaluation of arithmetic expressions. For example, the sequential search algorithm of Figure 2.13 contains the following arithmetic operations:

Set the value of *i* to 1 (line 2).

⋮

Add 1 to the value of *i* (line 7).

These algorithmic operations can be translated quite easily into assembly language as follows:

```
        LOAD        ONE     --Put a 1 into register R.
        STORE       I       --Store the constant 1 into i.
        ⋮
        INCREMENT I         --Add 1 to memory location i.
        ⋮                   --These data should be placed after the HALT.
I:      .DATA       0       --The index value. Initially it is 0.
ONE:    .DATA       1       --The integer constant 1.
```

Note how readable this code is, compared to machine language, because of such op code mnemonics as LOAD and STORE and the use of descriptive labels such as I and ONE.

Another example is the following assembly language translation of the arithmetic expression $A = B + C - 7$. (Assume that B and C have already been assigned values.)

```
        LOAD      B       --Put the value B into register R.
        ADD       C       --R now holds the sum (B + C).
        SUBTRACT  SEVEN   --R now holds (B + C - 7).
        STORE     A       --Store the result into A.
        ⋮                 --These data should be placed after the HALT.
A:      .DATA     0
B:      .DATA     0
C:      .DATA     0
SEVEN:  .DATA     7       --The integer constant 7.
```

Another important algorithmic operation involves testing and comparing values. The comparison of values and the subsequent use of the outcome to decide what to do next are termed a **conditional** operation, which we first saw in Section 2.2.3. Here is a conditional that outputs the larger of two values x and y. Algorithmically, it is expressed as follows:

```
Input the value of x
Input the value of y
If x ≥ y then
  Output the value of x
Else
  Output the value of y
```

In assembly language, this conditional operation can be translated as follows:

```
        IN        X       --Read the first data value
        IN        Y       --and now the second.
        LOAD      Y       --Load the value of Y into register R.
        COMPARE   X       --Compare X to Y and set the condition codes.
        JUMPLT    PRINTY  --If X is less than Y, jump to PRINTY.
        OUT       X       --We get here only if X ≥ Y, so print X.
        JUMP      DONE    --Skip over the next instruction and continue.
PRINTY: OUT       Y       --We get here if X < Y, so print Y.
DONE:   ⋮                 --The program continues here.
        ⋮
                          --The following data go after the HALT.
X:      .DATA     0       --Space for the two data values.
Y:      .DATA     0
        ⋮
```

Another important algorithmic primitive is **looping**, which was also introduced in Section 2.2.3. The following algorithmic example contains a while loop that executes 10,000 times.

Step	*Operation*	*Explanation*
1	Set *i* to 0	Start the loop counter at 0.
2	While the value of *i* < 10,000 do lines 3 through 9.	
3–8	⋮	Here is the loop body that is to be done 10,000 times.
9	Add 1 to the value of *i*	Increment the loop counter.
10	End of the loop	
11	Stop	

This looping construct is easily translated into assembly language.

```
            LOAD      ZERO      --Initialize the loop counter to 0.
            STORE     I         --This is step 1 of the algorithm.
LOOP:       LOAD      MAXVALUE  --Put 10,000 into register R.
            COMPARE   I         --Compare I against 10,000.
            JUMPEQ    DONE      --If I = 10,000 we are done (step 2).
            ⋮                   --Here is the loop body (steps 3–8).
            INCREMENT I         --Add 1 to I (step 9).
            JUMP      LOOP      --End of the loop body (step 10).
DONE:       HALT                --Stop execution (step 11).
ZERO:       .DATA     0         --This is the constant 0.
I:          .DATA     0         --The loop counter. It goes to 10,000.
MAXVALUE:   .DATA     10000     --Maximum number of executions.
            ⋮
```

As a final example, we will show a complete assembly language program (including all necessary pseudo-ops) to solve the following problem:

> Read in a sequence of nonnegative numbers, one number at a time, and compute a running sum. When you encounter a negative number, print out the sum of the nonnegative values and stop.

Thus, if the input is

```
8
31
7
5
–1
```

then the program should output the value 51, which is the sum (8 + 31 + 7 + 5). An algorithm to solve this problem is shown in Figure 6.7, using the pseudocode notation of Chapter 2.

Our next task is to convert the algorithmic primitives of Figure 6.7 into assembly language instructions. A program that does this is shown in Figure 6.8.

FIGURE 6.7

Algorithm to Compute the Sum of Numbers

Step	Operation
1	Set the value of Sum to 0
2	Input the first number *N*
3	While *N* is not negative do
4	Add the value of *N* to Sum
5	Input the next data value *N*
6	End of the loop
7	Print out Sum
8	Stop

Of all the examples in this chapter, the program in Figure 6.8 demonstrates best what is meant by the phrase *user-oriented virtual environment*. Although it is not as clear as natural language or the pseudocode of Figure 6.7, this program can be read and understood by humans as well as computers. Tasks such as modifying the program and locating an error are significantly easier on the code of Figure 6.8 than on its machine language equivalent.

The program in Figure 6.8 is an important milestone in that it represents a culmination of the algorithmic problem-solving process. Earlier chapters introduced algorithms and problem-solving (Chapters 1, 2, 3), discussed how to build computers to execute algorithms (Chapters 4, 5), and introduced system software that enables us to code algorithms into a language that computers can translate and execute (Chapter 6). The program in Figure 6.8 is the end product of this discussion: It can be input to an assembler, translated into machine language, loaded into a Von Neumann computer, and executed to produce answers to our problem. This **algorithmic problem-solving cycle** is one of the central themes of computer science.

FIGURE 6.8

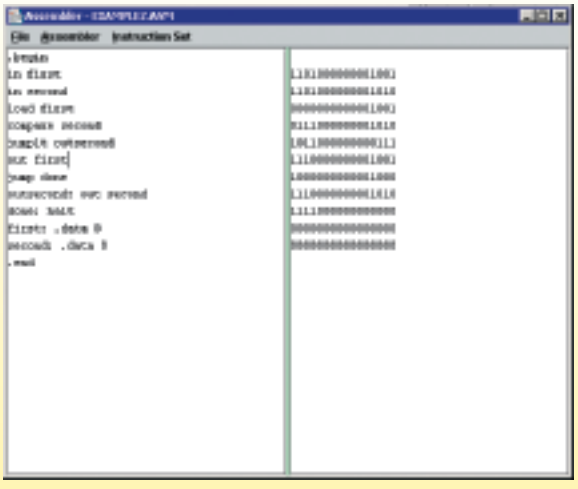
Assembly Language Program to Compute the Sum of Nonnegative Numbers

```
        .BEGIN                  --This marks the start of the program.
        CLEAR   SUM             --Set the running sum to 0 (line 1).
        IN      N               --Input the first number N (line 2).
--The next three instructions test whether N is a negative number (line 3).
AGAIN:  LOAD    ZERO            --Put 0 into register R.
        COMPARE N               --Compare N and 0.
        JUMPLT  NEG             --Go to NEG if N < 0.
--We get here if N ≥ 0. We add N to the running sum (line 4).
        LOAD    SUM             --Put SUM into R.
        ADD     N               --Add N. R now holds (N + SUM).
        STORE   SUM             --Put the result back into SUM.
--Get the next input value (line 5).
        IN      N
--Now go back and repeat the loop (line 6).
        JUMP    AGAIN
--We get to this section of the program only when we encounter a negative value.
NEG:    OUT     SUM             --Print the sum (line 7)
        HALT                    --and stop (line 8).
--Here are the data generation pseudo-ops
SUM:    .DATA   0               --The running sum goes here.
N:      .DATA   0               --The input data are placed here.
ZERO:   .DATA   0               --The constant 0.
--Now we mark the end of the entire program.
        .END
```

PRACTICE PROBLEMS

1. Using the instruction set in Figure 6.5, translate the following algorithmic operations into assembly code. Show all necessary .DATA pseudo-ops.
   a. Add 1 to the value of *x*
   b. Add 50 to the value of *x*
   c. Set *x* to the value *y* + *z* – 2
   d. If *x* > 50 then output the value of *x*, otherwise input a new value of *x*
   e.
   ```
   sum = 0
   I = 0
   While I < 50 do
      sum = sum + I;
      I = I + 1;
   End of the loop
   ```
2. Using the instruction set in Figure 6.5, write a complete assembly language program (including all necessary pseudo-ops) that reads in numbers and counts how many inputs it reads until it encounters the first negative value. The program then prints out that count and stops. For example, if the input data is 42, 108, 99, 60, 1, 42, 3, –27, then your program outputs the value 7 because there are seven nonnegative values before the appearance of the negative value –27.
3. Now modify your program from Question 2 above so that if you have not encountered a negative value after 100 inputs your program stops and outputs the value 100.

LABORATORY EXPERIENCE 10



This section of Chapter 6 introduced assembly language instructions and programming techniques. However, as mentioned before, you do not learn programming and problem solving by reading and watching but rather by doing and trying. In this laboratory experience you will program in an assembly language that is virtually identical to the one shown in Figure 6.5. You will be able to design and write programs like the one shown in Figure 6.8 and execute them on a simulated Von Neumann computer. You will observe the effect of individual instructions on the functional units of this machine and produce results.

This experience should give you a much deeper understanding of the concepts of assembly language programming and the Von Neumann architecture. It will also tie together the hardware concepts of Level 2 (Chapters 4 and 5) and the virtual machine system software concepts of Level 3. This lab shows how an assembly language program is written, translated, and loaded into a Von Neumann machine and executed by that machine using the ideas presented in the previous chapters. The figure here shows an example of the information that will be displayed during this laboratory.

6.3.3 *Translation and Loading*

What must happen in order for the assembly language program in Figure 6.8 to be executed on a processor? Figure 6.4 shows that before our source program can be run, we must invoke two system software packages—an **assembler** and a **loader**.

An **assembler** translates a symbolic assembly language program, such as the one in Figure 6.8, into machine language. We usually think of translation as an extremely difficult task. In fact, if two languages differ greatly in vocabulary, grammar, and syntax, it can be quite formidable. (This is why a translator for a high-level programming language is a very complex piece of software.) However, machine language and assembly language are very similar, and therefore an assembler is a relatively simple piece of system software.

An assembler must perform the following four tasks, none of which is particularly difficult.

1. Convert symbolic op codes to binary.
2. Convert symbolic addresses to binary.
3. Perform the assembler services requested by the pseudo-ops.
4. Put the translated instructions into a file for future use.

Let's see how these operations are carried out using the hypothetical assembly language of Figure 6.5.

The conversion of symbolic op codes such as LOAD, ADD, and SUBTRACT to binary makes use of a structure called the **op code table**. This is an alphabetized list of all legal assembly language op codes and their binary equivalents. An op code table for the instruction set of Figure 6.5 is shown in Figure 6.9. (The table assumes that the op code field is 4 bits wide.)

The assembler finds the operation code mnemonic in column 1 of the table and replaces the characters with the 4-bit binary value in column 2. (If the mnemonic is not found, then the user has written an illegal op code, which results in an error message.) Thus, for example, if we use the mnemonic SUBTRACT in our program, the assembler converts it to the binary value 0101.

To look up the code in the op code table, we could use the sequential search algorithm introduced in Chapter 2 and shown in Figure 2.13. However, using this algorithm may significantly slow the translation of our program. The analysis of the sequential search algorithm in Chapter 3 showed that locating a single item in a list of N items takes, on the average, $N/2$ comparisons if the item is in

FIGURE 6.9

Structure of the Op Code Table

Operation	Binary Value
ADD	0011
CLEAR	0010
COMPARE	0111
DECREMENT	0110
HALT	1111
⋮	
STORE	0001
SUBTRACT	0101

the table and N comparisons if it is not. In Chapter 5 we stated that modern computers may have as many as 500 machine language instructions in their instruction set, so the size of the op code table of Figure 6.9 could be as large as $N = 500$. This means that using sequential search, we perform an average of $N/2$, about 250, comparisons for every op code in our program. If our assembly language program contains 10,000 instructions (not an unreasonably large number), the op code translation task requires a total of 10,000 instructions $\times$ 250 comparisons/instruction = 2.5 million comparisons. That is a lot of searching, even for a computer.

Because the op code table of Figure 6.9 is sorted alphabetically, we can instead use the more efficient **binary search** algorithm discussed in Section 3.4.2 and shown in Figure 3.18. On the average, the number of comparisons needed to find an element using binary search is not $N/2$ but (lg N), the logarithm of N to the base 2. [*Note:* (lg N) is the value k such that $2^k = N$.] For a table of size $N = 500$, $N/2$ is 250, whereas (lg N) is approximately 9 ($2^9 = 512$). This says that on the average, we find an op code in the table in about 9 comparisons rather than 250. If our assembly language program contains 10,000 instructions, then the op code translation task requires only about 10,000 $\times$ 9 = 90,000 comparisons rather than 2.5 million, a reduction of 2,410,000. By selecting a better algorithm, we achieve an increase in speed of about 96%—quite a significant improvement!

This example demonstrates why algorithm analysis, introduced in Chapter 3, is such a critically important part of the design and implementation of system software. Replacing a slow algorithm by a faster one can turn an "insoluble" problem into a solvable one and a worthless solution into a highly worthwhile one. Remember that, in computer science, we are looking not just for correct solutions but for efficient ones as well.

After the op code has been converted into binary, the assembler must perform a similar task on the address field. It must convert the address from a symbolic value, such as X or LOOP, into the correct binary address. This task is more difficult than converting the op code, because the assembler itself must determine the correct numeric value of all symbols used in the label field. There is no "built-in" address conversion table equivalent to the op code table of Figure 6.9.

In assembly language a symbol is defined when it appears in the label field of an instruction or data pseudo-op. Specifically, the symbol is given the value of the address of the instruction to which it is attached. Assemblers usually make two passes over the source code, where a **pass** is defined as the process of examining and processing every assembly language instruction in the program, one instruction at a time. During the **first pass** over the source code, the assembler looks at every instruction, keeping track of the memory address where this instruction will be stored when it is translated and loaded into memory. It does this by knowing where the program begins in memory and knowing how many memory cells are required to store each machine language instruction or piece of data. It also determines whether there is a symbol in the label field of the instruction. If there is, it enters the symbol and the address of this instruction into a special table that it is building called a **symbol table**.

We can see this process more clearly in Figure 6.10. The figure assumes that each instruction and data value occupies one memory cell and that the first instruction of the program will be placed into address 0.

The assembler looks at the first instruction in the program, IN X, and determines that when this instruction is loaded into memory, it will go into memory cell 0. Because the label LOOP is attached to that instruction, the name LOOP is made equivalent to address 0. The assembler enters the (name,

FIGURE 6.10

Generation of the Symbol Table

Label	Code		Location Counter
LOOP:	IN	X	0
	IN	Y	1
	LOAD	X	2
	COMPARE	Y	3
	JUMPGT	DONE	4
	OUT	X	5
	JUMP	LOOP	6
DONE:	OUT	Y	7
	HALT		8
X:	.DATA	0	9
Y:	.DATA	0	10

(a)

Symbol	Address Value
LOOP	0
DONE	7
X	9
Y	10

(b)

value) pair (LOOP, 0) into the symbol table. This process of associating a symbolic name with a physical memory address is called **binding**, and the two primary purposes of the first pass of an assembler are (1) to bind all symbolic names to address values, and (2) to enter those bindings into the symbol table. Now, any time the programmer uses the name LOOP in the address field, the assembler can look up that symbol in column 1 of the symbol table and replace it with the address value in column 2, in this case address 0. (If it is not found, the programmer has used an undefined symbol, which produces an error message.)

The next six instructions of Figure 6.10(a), from IN Y to JUMP LOOP, do not contain labels, so they do not add new entries to the symbol table. However, the assembler must still update the counter it is using to determine the address where each instruction will ultimately be stored. The variable used to determine the address of a given instruction or piece of data is called the **location counter**. The location counter values are shown in the third column of Figure 6.10(a). Using the location counter, the assembler can determine that the address values of the labels DONE, X, and Y are 7, 9, and 10, respectively. It binds these symbolic names and addresses and enters them in the symbol table, as shown in Figure 6.10(b). When the first pass is done, the assembler has constructed a symbol table that it can use during pass 2. The algorithm for pass 1 of a typical assembler is shown (using an alternative form of algorithmic notation called a **flowchart**) in Figure 6.11.

During the **second pass**, the assembler translates the source program into machine language. It has the op code table to translate mnemonic op codes to binary, and it has the symbol table to translate symbolic addresses to binary. Therefore, the second pass is relatively simple, involving two table look-ups and the generation of two binary fields. For example, if we assume that our instruction format is a 4-bit op code followed by a single 12-bit address, then given the instruction

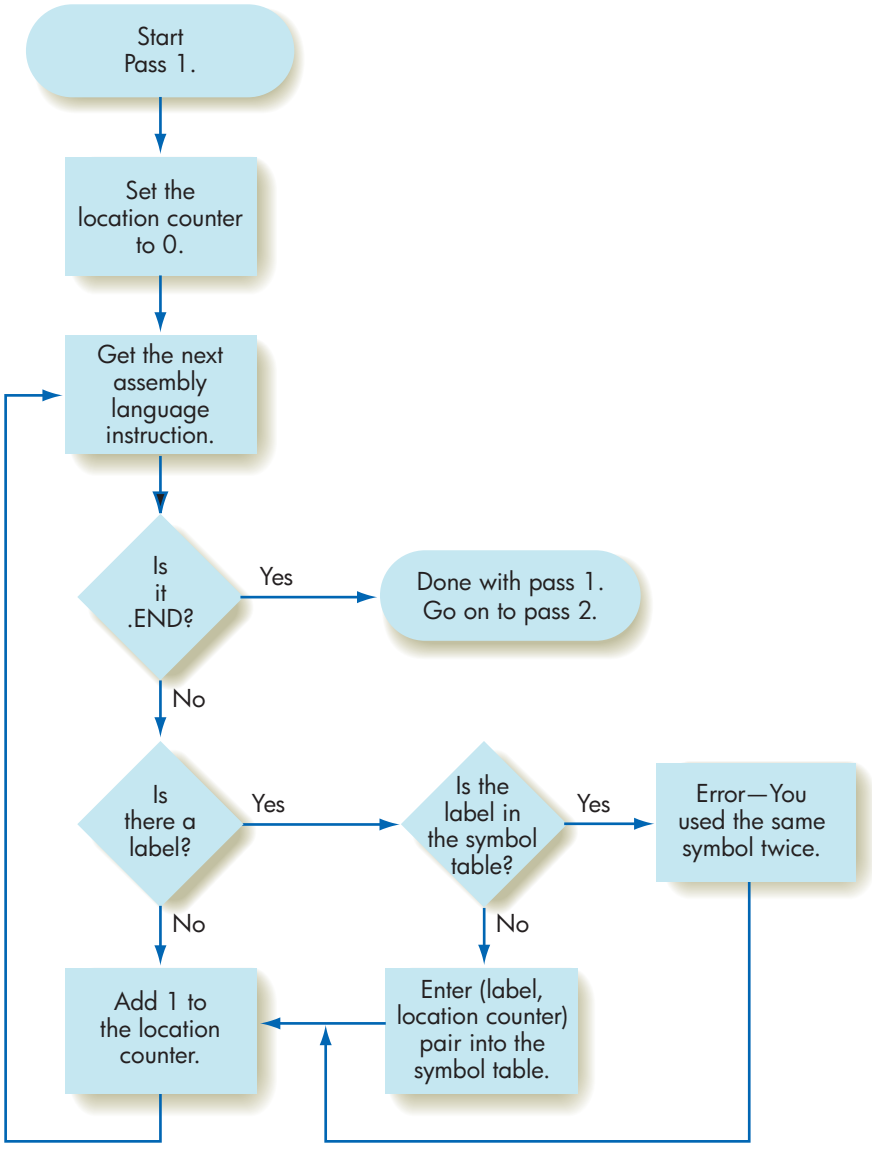
```
SUBTRACT    X
```

the assembler

1. Looks up SUBTRACT in the op code table of Figure 6.9 and places the 4-bit binary value 0101 in the op code field.

FIGURE 6.11

Outline of Pass 1 of the Assembler



2. Looks up the symbol X in the symbol table of Figure 6.10(b) and places the binary address value 0000 0000 1001 (decimal 9) into the address field.

After these two steps, the assembler produces the 16-bit instruction

0101 0000 0000 1001

which is the correct machine language equivalent of the assembly language statement SUBTRACT X.

When it is done with one instruction, the assembler moves on to the next and translates it in the same fashion. This continues until it sees the pseudo-op .END, which terminates translation.

The other responsibilities of pass 2 are also relatively simple:

- Handling data generation pseudo-ops (only .DATA in our example).
- Producing the object file needed by the loader.

The .DATA pseudo-op asks the assembler to build the proper binary representation for the signed decimal integer in the address field. To do this, the assembler must implement the sign/magnitude integer representation algorithms described in Section 4.2.

Finally, after all the fields of an instruction have been translated into binary, the newly built machine language instruction and the address of where it is to be loaded are written out to a file called the **object file**. (On Windows machines, this is referred to as an .EXE file.) The algorithm for pass 2 of the assembler is shown in Figure 6.12.

After completion of pass 1 and pass 2, the object file contains the translated machine language **object program**, referred to in Figure 6.4. One possible

PRACTICE PROBLEMS

1. Translate the following algorithm into assembly language using the instructions in Figure 6.5.

Step	Operation
1	Set *Negative Count* to 0
2	Set *i* to 1
3	While $i \leq 50$ do lines 4 through 6
4	Input a number *N*
5	If $N < 0$ then increment *Negative Count* by 1
6	Increment *i* by 1
7	End of the loop
8	Output the value of *Negative Count*
9	Stop

2. What is the machine language representation of each of the following instructions? Assume the symbol table values are as shown in Figure 6.10(b) and the instruction format is that of Figure 6.13.

   a. COMPARE Y
   b. JUMPNEQ DONE
   c. DECREMENT LOOP

3. What is wrong or inconsistent with the instruction that is shown in Problem 2(c)?

4. Take the assembly language program that you developed in Problem 1 and determine the physical memory address associated with each label in the symbol table. (Assume the first instruction is loaded into address 0 and that each instruction occupies one cell.)

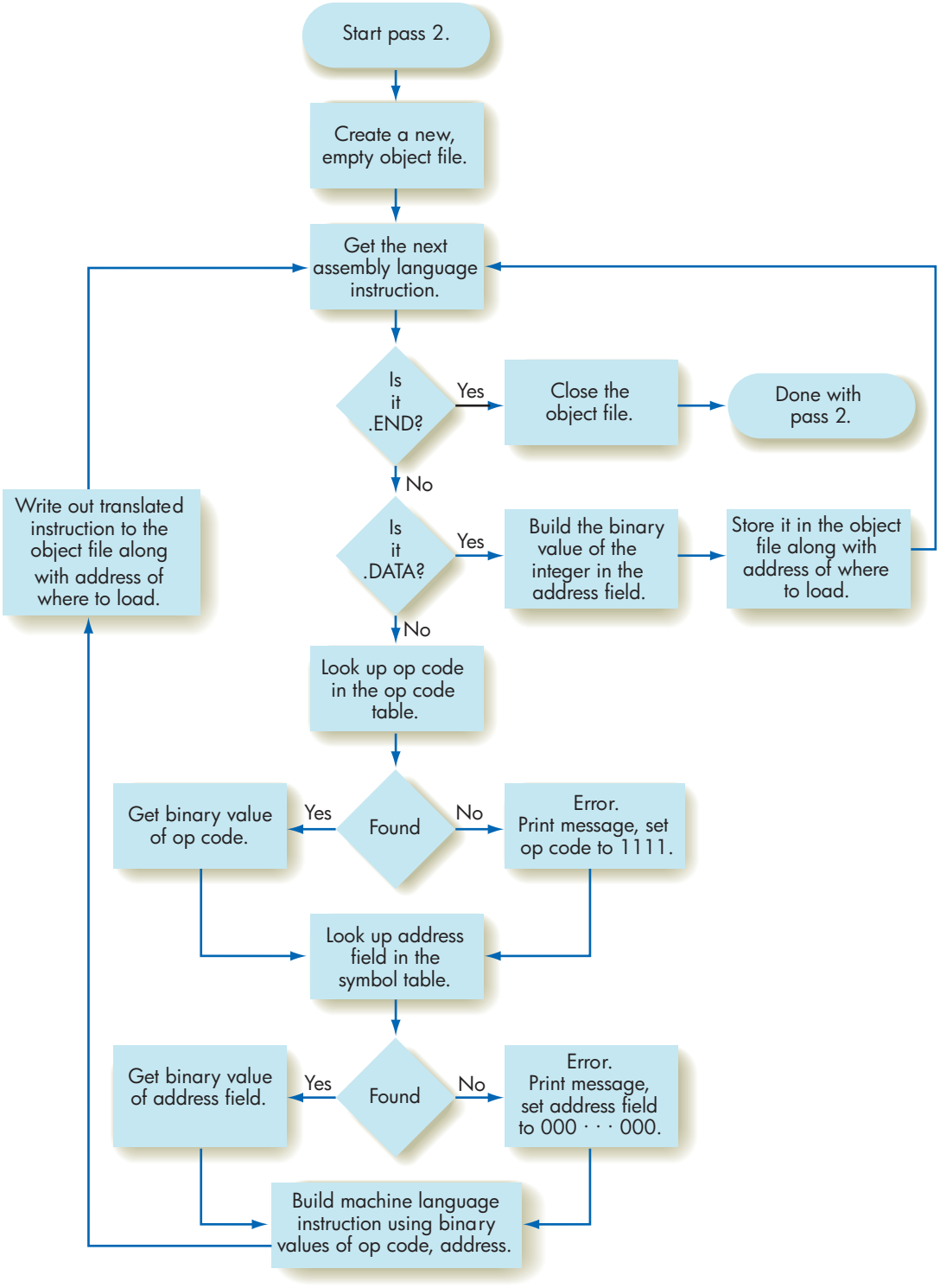


FIGURE 6.12

Outline of Pass 2 of the Assembler

FIGURE 6.13

Example of an Object Program

INSTRUCTION FORMAT:

OP CODE	ADDRESS
4 bits	12 bits

OBJECT PROGRAM:

Address	*Machine Language Instruction*	*Meaning*
0000	1101 000000001001	IN X
0001	1101 000000001010	IN Y
0010	0000 000000001001	LOAD X
0011	0111 000000001010	COMPARE Y
0100	1001 000000000111	JUMPGT DONE
0101	1110 000000001001	OUT X
0110	1000 000000000000	JUMP LOOP
0111	1110 000000001010	OUT Y
1000	1111 000000000000	HALT
1001	0000 000000000000	The constant 0
1010	0000 000000000000	The constant 0

object program for the assembly language program of Figure 6.10(a) is shown in Figure 6.13. (Note that a real object file contains only the address and instruction fields. The meaning field is included here for clarity only.)

The object program shown in Figure 6.13 becomes input to yet another piece of system software called a **loader**. It is the task of the loader to read instructions from the object file and store them into memory for execution. To do this, it reads an address value—column 1 of Figure 6.13—and a machine language instruction—column 2 of Figure 6.13—and stores that instruction into the specified memory address. This operation is repeated for every instruction in the object file. When loading is complete, the loader places the address of the first instruction (0 in this example) into the program counter (PC) to initiate execution. The hardware, as we learned in Chapter 5, then begins the fetch, decode, and execute cycle starting with the instruction whose address is located in the PC, namely the beginning of this program.

6.4 Operating Systems

To carry out the services just described (translate a program, load a program, and run a program), a user must issue **system commands**. These commands may be lines of text typed at a terminal, such as

>assemble MyProg	(Invoke the assembler to translate a program called MyProg.)
>run MyProg	(Load the translated MyProg and start execution.)

or they may be represented by icons displayed on the screen and selected with a mouse and a button, using a technique called **point-and-click**.

Regardless of how the process is initiated, the important question is: what program examines these commands? What piece of system software waits for requests and activates other system programs like a translator or loader to service these requests? The answer is the **operating system**, and, as shown in Figure 6.2, it is the "top-level" system software component on a computer.

6.4.1 *Functions of an Operating System*

An operating system is an enormously large and complex piece of software that has many responsibilities within a computer system. This section examines five of the most important tasks that it performs.

THE USER INTERFACE. The operating system is executing whenever no other piece of user or system software is using the processor. Its most important task is to wait for a user command delivered via the keyboard, mouse, or other input device. If the command is legal, the operating system activates and schedules the appropriate software package to process the request. In this sense, the operating system acts like the computer's *receptionist* and *dispatcher*.

Operating system commands usually request access to hardware resources (processor, printer, communication lines), software services (translator, loader, text editor, application program), or information (data files, date, time). Examples of typical operating system commands are shown in Figure 6.14. Modern operating systems can execute dozens or even hundreds of different commands.

After a user enters a command, the operating system determines which software package needs to be loaded and put on the schedule for execution. When that package completes execution, control returns to the operating system, which waits for a user to enter the next command. This **user interface** algorithm is diagrammed in Figure 6.15.

The user interfaces on the operating systems of the 1950s, 1960s, and 1970s were text oriented. The system displayed a **prompt character** on the screen to indicate that it was waiting for input, and then it waited for something to happen. The user entered commands in a special, and sometimes quite complicated, **command language**. For example, on the UNIX operating system, widely used on personal computers and workstations, the following command asks the system to list the names and access privileges of the files contained in the home directory of a user called mike.

```
> ls -al /usr/mike/home    (">"is the prompt character)
```

As you can see, commands were not always easy to understand, and learning the command language of the operating system was a major stumbling block for new users. Unfortunately, without first learning some basic commands, no useful work could be done.

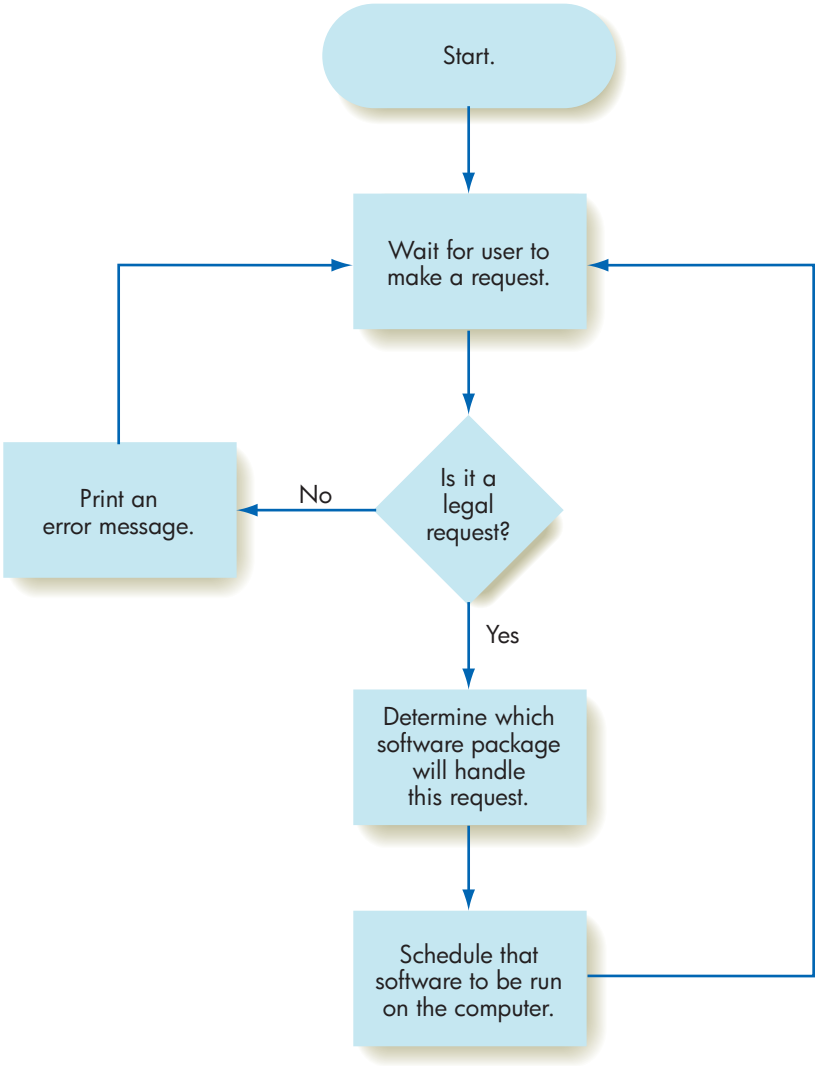
FIGURE 6.14

Some Typical Operating System Commands

- Translate a program
- Run a program
- Save information in a file
- Retrieve a file previously stored
- List all the files for this user
- Print a file on a specified device
- Delete or rename a file
- Copy a file from one I/O device to another
- Let the user set or change a password
- Establish a network connection
- Tell me the current time and date

FIGURE 6.15

User Interface Responsibility of the Operating System



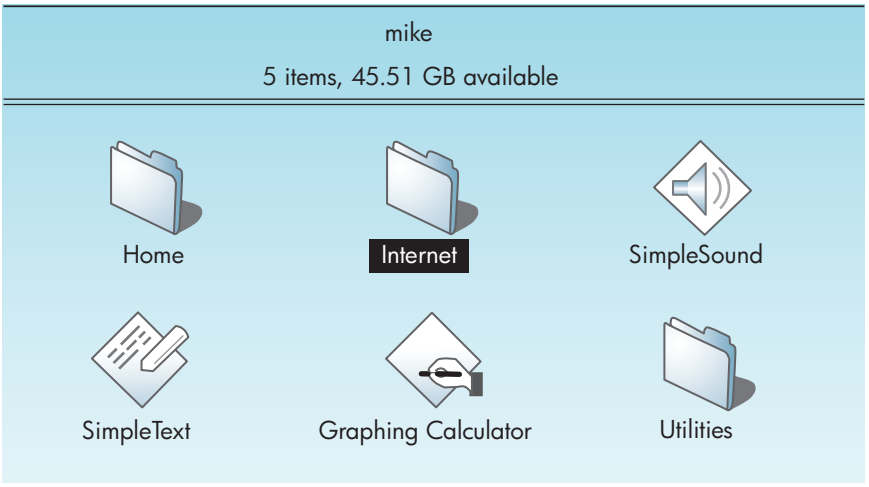
Because users found text-oriented command languages very cumbersome, virtually all modern operating systems now include a graphical user interface, or **GUI**. To communicate with a user, a GUI supports visual aids and point-and-click operations, rather than textual commands. The interface uses **icons**, **pull-down menus**, **scrolling windows**, and other visual elements and graphical metaphors that make it much easier for a user to formulate requests.

For example, Figure 6.16 shows a window listing the folders on the hard disk called mike. One of these is a folder called home. To list all the files contained in this folder, a user points-and-clicks on it, and the list of its files appears in a new window. Compare the clarity of that operation with the preceding UNIX command that does the same thing.

Graphical interfaces are a good example of the high-level virtual machine created by the operating system. A GUI hides a great deal of the underlying hardware and software, and it makes the computer appear very easy to use. In fact, the computer that produces the elegant windowing environment shown in Figure 6.16 is the same Von Neumann machine described in Chapters 4 and 5.

FIGURE 6.16

Example of a Graphical User Interface



SYSTEM SECURITY AND PROTECTION. In addition to being a receptionist, the operating system also has the responsibilities of a *security guard*—controlling access to the computer and its resources. It must prevent unauthorized users from accessing the system and prevent authorized users from doing unauthorized things.

At a minimum, the operating system must not allow people to access the computer if they have not been granted permission. In the "olden days" of computing (the 1950s and 1960s), security was implemented by physical means—walls and locked doors around the computer and security guards at the door to prevent unauthorized access. However, when telecommunications networks

A Machine for the Rest of Us

In January 1984, Apple Computer launched its new line of Macintosh computers with a great deal of showmanship: a TV commercial at the 1984 NFL Superbowl Game. The company described the Macintosh as a computer that anyone could understand and use—"a machine for the rest of us." People who saw and used it quickly agreed, and in the early days, its major selling point was that "a Macintosh is much easier to use than an IBM PC." However, the Macintosh and IBM PC were extremely similar in terms of hardware, and they both used something like the architecture of Figure 5.18. Both systems used Von Neumann–type processors, and these processors executed similar sets of machine language instructions exactly as described in Chapter 5. In fact, in 2006 Apple began using the same type of Intel processors as in the IBM PC and its clones. It certainly was not the underlying hardware that created these huge differences in ease of use.

What made the Macintosh easier to use was its radically new graphical user interface, created by two system software packages called the *Finder* and the *System*. They produced a sophisticated visual environment that most users found much easier to understand than the text-oriented interface of *MS-DOS*, the most popular PC-based operating system of the 1980s and early 1990s. IBM users quickly realized the importance of having a powerful user interface, and in the early and mid-1990s began to switch to Microsoft *Windows*, which provided a windowing environment similar to the Macintosh. Newer versions of these systems, such as *Mac OS X, Windows NT, Windows XP*, and *Windows Vista* all represent attempts at creating an even more powerful and easy to use virtual environment.

We can see now that it was wrong for Apple to say that "a Macintosh is easier to use than a PC." What they should have said is that "the virtual machine environment created by the Macintosh operating system is easier to use than the virtual machine environment created by the Windows operating system." However, maybe that was just a little too wordy!

appeared on the scene in the late 1960s and 1970s (we will discuss them in detail in Chapter 7), access to computers over telephone lines became possible from virtually anywhere in the world, and responsibility for access control migrated from the guard at the door to the operating system inside the machine.

In most operating systems, access control means requiring a user to enter a legal **user name** and **password** before any other requests are accepted. For example, here is what a user sees when logging on to the network server at Macalester College:

Welcome to the Macalester College Computing Center.
Please enter your User Name and Password in the appropriate boxes:
User Name: ☐
Password: ☐

If an incorrect user name or password is entered, the operating system does not allow access to the computer.

It is also the operating system's responsibility to safeguard the **password file** that stores all valid user name/password combinations. It must prevent this file from being accessed by any unauthorized users, because that would compromise the security of the entire system. This is analogous to putting a lock on your door but also making sure that you don't lose the key. (Of course, some privileged users, called **superusers**—usually computer center employees or system administrators—must be able to access and maintain this file.) To provide this security, the operating system may choose to **encrypt** the password file using an encoding algorithm that is extremely difficult to crack. A thief must steal not only the encrypted text but also the algorithm to change the encrypted text back to the original clear text. Without this information the stolen password file is useless. Operating systems use encryption algorithms whenever they must provide a high degree of security for sensitive information. We will learn more about these encryption algorithms in Chapter 8.

Even when valid users gain access to the system, there are things they should not be allowed to do. The most obvious is that they should access only their own personal information. They should not be able to look at the files or records of other users. Therefore, when the operating system gets a request such as

> open filename (Open a file and allow this user to access it.)

(Or click Open in the File menu.)

it must determine who is the owner of the file—that is, who created it. If the individual accessing the file is not the owner, then it usually rejects the request. However, most operating systems allow the owner of a file to provide a list of additional authorized users or a general class of authorized users, such as all students or all faculty. Like the password file, these **authorization lists** are highly sensitive files, and operating systems generally store them in an encrypted format.

Most modern operating systems not only determine whether you are allowed to access a file, they also check what operations you are permitted to do on that file. The following hierarchically ordered list shows the different types of operations that users may be permitted to do on a file:

- Read the information in the file but not change it
- Append new information to the end of the file but not change existing information

- Change existing information in the file
- Delete the file from the system

For example, the grade file GRADES of a student named Smith could have the authorization list shown in Figure 6.17.

This authorization list says that Smith, the student whose grades are in the file, has the right to access his or her own file, but only to read the information.

Jones, a clerk in the administration center, can read the file and can append new grades to the end of the file at the completion of the term. Adams, the school's registrar, can read and append information and is also allowed to change the student's grades if an error was made. Doe, the director of the computer center, can do all of these operations as well as delete the file and all its information.

Permission to look at information can be given to a number of people. However, changing information in a file is a sensitive operation (think about changing a payroll file), and permission to make changes must be limited. Deleting information is the most powerful and potentially damaging operation of all, and its use must be restricted to people at the highest level. It is the operating system's responsibility to help ensure that individuals are authorized to carry out the operation they request.

Computers today play such a critical role in the storage and management of economic and personal data that this security responsibility has taken on an increasingly important role. We investigate this topic in detail in Chapter 8.

EFFICIENT ALLOCATION OF RESOURCES. Section 5.2.2 described the potentially enormous difference in speed between a processor and an I/O unit: up to 5 orders of magnitude. A hardware device called an I/O controller (Figure 5.9) frees the processor to do useful work while the I/O operation is being completed. What useful work can a processor do in this free time? What ensures that this valuable resource is used efficiently? Again, it is the operating system's responsibility to see that the resources of a computer system are used efficiently as well as correctly.

To ensure that a processor does not sit idle if there is useful work to do, the operating system keeps a **queue** (a waiting line) of programs that are ready to run. Whenever the processor is idle, the operating system picks one of these jobs and assigns it to the processor. This guarantees that the processor always has something to do.

To see how this algorithm might work, let's define the following three classes of programs:

Running	The one program currently executing on the processor
Ready	Programs that are loaded in memory and ready to run but are not yet executing
Waiting	Programs that cannot run because they are waiting for an I/O operation (or some other time-consuming event) to finish

FIGURE 6.17

Authorization List for the File GRADES

File: GRADES

NAME	PERMITTED OPERATIONS
Smith	R (R = Read only)
Jones	RA (A = Append)
Adams	RAC (C = Change)
Doe	RACD (D = Delete)

Here is how these three lists might look at some instant in time:

Waiting	*Ready*	*Running*
	B	A
	C	
	D	

There are four programs, called A, B, C, and D, in memory. Program A is executing on the processor; B, C, and D are ready to run and are in line waiting their turn. Assume that program A performs the I/O operation "read a sector from the disk." (Maybe it is a word processor, and it needs to get another piece of the document on which you are working.) We saw in Section 5.2.2 that, relative to processing speeds, this operation takes a long time, about 10 msec or so. While it is waiting for this disk I/O operation to finish, the processor has nothing to do, and system efficiency plummets.

To solve this problem, the operating system can do some shuffling. It first moves program A to the waiting list, because it must wait for its I/O operation to finish before it can continue. It then selects one of the ready programs (say B) and assigns it to the processor, which starts executing it. This leads to the following situation:

Waiting	*Ready*	*Running*
A	C	B
	D	

Instead of sitting idle while A waits for I/O, the processor works on program B and gets something useful done. This is equivalent to working on another project while waiting for your secretary to fetch a document, instead of waiting and doing nothing.

Perhaps B also does an I/O operation. If so, then the operating system repeats the same steps. It moves B to the waiting list, picks any ready program (say C) and starts executing it, producing the following situation:

Waiting	*Ready*	*Running*
A	D	C
B		

As long as there is at least one program that is ready to run, the processor always has something useful to do.

At some point, the I/O operation that A started finishes, and the "I/O completed interrupt signal" described in Section 5.2.2 is generated. The appearance of that signal indicates that program A is now ready to run, but it cannot do so immediately because the processor is currently assigned to C. Instead, the operating system moves A to the ready list, producing the following situation:

Waiting	*Ready*	*Running*
B	D	C
	A	

Programs cycle from running to waiting to ready and back to running, each one using only a portion of the resources of the processor.

In Chapter 5 we stated that the execution of a program is an unbroken repetition of the fetch/decode/execute cycle from the first instruction of the program to the HALT. Now we see that this view may not be completely

accurate. For reasons of efficiency, the running history of a program may be a sequence of starts and stops—a cycle of execution, waits for I/O operations, waits for the processor, followed again by execution. By having many programs loaded in memory and sharing the processor, the operating system can use the processor to its fullest capability and run the overall system more efficiently.

PRACTICE PROBLEM

Assume that programs spend about 25% of their time waiting for I/O operations to complete. If there are two programs loaded into memory, what is the likelihood that both programs will be blocked waiting for I/O and there will be nothing for the processor to do? What percentage of time will the processor be busy? (This value is called **processor utilization**.) By how much does processor utilization improve if we have four programs in memory instead of two?

THE SAFE USE OF RESOURCES. Not only must resources be used *efficiently*, they must also be used *safely*. That doesn't mean an operating system must prevent users from sticking their fingers in the power supply and getting electrocuted! The job of the operating system is to prevent programs or users from attempting operations that cause the computer system to enter a state where it is incapable of doing any further work—a "frozen" state where all useful work comes to a grinding halt.

To see how this can happen, imagine a computer system that has one laser printer, one data file called D, and two programs A and B. Program A wants to load data file D and print it on the laser printer. Program B wants to do the same thing. Each of them makes the following requests to the operating system:

Program A	*Program B*
Get data file D.	Get the laser printer.
Get the laser printer.	Get data file D.
Print the file.	Print the file.

If the operating system satisfies the first request of each program, then A "owns" data file D and B has the laser printer. When A requests ownership of the laser printer, it is told that the printer is being used by B. Similarly, B is told that it must wait for the data file until A is finished with it. Each program is waiting for a resource to become available that will never become free. This situation is called a **deadlock**. Programs A and B are in a permanent waiting state, and if there is no other program ready to run, all useful work on the system ceases.

More formally, deadlock means that there is a set of programs, each of which is waiting for an event to occur before it may proceed, but that event can be caused only by another waiting program in the set. Another example is a telecommunication system in which program A sends messages to program B, which acknowledges their correct receipt. Program A cannot send another message to B until it knows that the last one has been correctly received.

Program A	*Program B*
Message →	
	← Acknowledge
Message →	
	← Acknowledge
Message →	

Suppose B now sends an acknowledgment, but it gets lost. (Perhaps there was static on the line, or a lightning bolt jumbled the signal.) What happens? Program A stops and waits for receipt of an acknowledgment from B. Program B stops and waits for the next message from A. Deadlock! Neither side can proceed, and unless something is done, all communication between the two will cease.

How does an operating system handle deadlock conditions? There are two basic approaches, called **deadlock prevention** and **deadlock recovery**. In deadlock prevention, the operating system uses resource allocation algorithms that prevent deadlock from occurring in the first place. In the example of the two programs simultaneously requesting the laser printer and the data file, the problem is caused by the fact that each program has a portion of the resources needed to solve its problem, but neither has all that it requested. To prevent this, the operating system can use the following algorithm:

> *If a program cannot get all the resources that it needs, it must give up all the resources it currently owns and issue a completely new request.*

Essentially, this resource allocation algorithm says, If you cannot get everything you need, then you get nothing. If we had used this algorithm, then after program A acquired the laser printer but not the data file, it would have had to relinquish ownership of the printer. Now B could get everything it needed to execute, and no deadlock would occur. (It could also work in the reverse direction, with B relinquishing ownership of the data file and A getting the needed resources. Which scenario unfolds depends on the exact order in which requests are made.)

In the telecommunications example, one deadlock prevention algorithm is to require that messages and acknowledgments never get garbled or lost. Unfortunately, that is impossible. Real-world communication systems (telephone, microwave, satellite) do make errors, so we are powerless to guarantee that deadlock conditions can never occur. Instead we must detect them and recover from them when they do occur. This is typical of the class of methods called **deadlock recovery algorithms**.

For example, here is a possible algorithmic solution to our telecommunications problem:

> *Sender:* Number your messages with the nonnegative integers 0, 1, 2, . . . and send them in numerical order. If you send message number *i* and have not received an acknowledgment for 30 seconds, send message *i* again.
>
> *Receiver:* When you send an acknowledgment, include the number of the message you received. If you get a duplicate copy of message *i,* send another acknowledgment and discard the duplicate.

Using this algorithm, here is what might happen:

Program A	*Program B*
Message (1) →	
	← Acknowledge (1)
Message (2) →	
	← Acknowledge (2) (Assume this acknowledgment is lost.)

At this point we have exactly the same deadlock condition described earlier. However, this time we are able to recover in a relatively short period. For 30 seconds nothing happens. However, after 30 seconds A sends message (2) a second time. B acknowledges it and discards it (because it already has a copy), and communication continues:

(Wait 30 seconds.)	
Message (2) →	(Discard this duplicate copy but acknowledge it.)
	← Acknowledge (2)
Message (3) →	

We have successfully recovered from the error, and the system is again up and running.

Regardless of whether we prevent deadlocks from occurring or recover from those that do occur, it is the responsibility of the operating system to create a virtual machine in which the user never sees deadlocks and does not worry about them. The operating system should create the illusion of a smoothly functioning, highly efficient, error-free environment—even if, as we know from our glimpse behind the scenes, that is not always the case.

SUMMARY. In this section we have highlighted some of the major responsibilities of the critically important software package called the operating system:

- User interface management (a receptionist)
- Control of access to system and files (a security guard)
- Program scheduling and activation (a dispatcher)
- Efficient resource allocation (an efficiency expert)
- Deadlock detection and error detection (a traffic officer)

These are by no means the operating system's only responsibilities, which can also include such areas as input/output processing, allocating priorities to programs, swapping programs in and out of memory, recovering from power failures, managing the system clock, and dozens of other tasks, large and small, essential to keeping the computer system running smoothly.

As you can imagine, given all these responsibilities, an operating system is an extraordinarily complex piece of software. An operating system for a large network of computers can require millions of lines of code, take thousands of person-years to develop, and cost as much to develop as the hardware on which it runs. Even operating systems for personal computers and workstations (e.g., Windows Vista, Linux, Mac OS X) are huge programs developed over periods of years by teams of dozens of computer scientists. Designing and creating a

The Open Source Movement

The design and development of an operating system like Windows Vista or Mac OS X is an enormous undertaking that can take thousands of person-years to complete. Furthermore, the likelihood of getting everything correct is quite small. (We have all had the experience of being frustrated by the freezes, errors, and crashes of our operating system.) One of the ways that people are attempting to address this issue is via the *Open Source Movement*. This is a worldwide movement of people who feel that the best way to develop efficient and bug-free software is to enlist the cooperation of interested, skilled, and altruistic programmers who are willing to work for free. They are inspired simply by the goals of producing high-quality software and of working cooperatively with like-minded individuals. The software is distributed to anyone who wants to use it, and it can be modified, improved, and changed by any user. This is quite different from the proprietary approach to software development within a corporation such as IBM or Microsoft, in which the development process is kept secret, and the source code is not shared with anyone else.

Essentially, the Open Source Movement encourages contributions to the development process from anyone in the world, in the belief that the more open the process and the more eyes examining the code, the more likely it is that errors or invalid assumptions will be located and corrected. Both the Linux operating system and the Apache Web server package were developed using the open source model.

high-level virtual environment is a difficult job, but without it, computers would not be so widely used nor anywhere near as important as they are today.

6.4.2 *Historical Overview of Operating Systems Development*

Like the hardware on which it runs, system software has gone through a number of changes since the earliest days of computing. The functions and capabilities of a modern operating system described in the previous section did not appear all at once but evolved over many years.

During the **first generation** of system software (roughly 1945–1955), there really were no operating systems and there was very little software support of any kind—typically just the assemblers and loaders described in Section 6.3. All machine operation was "hands-on." Programmers would sign up for a block of time and, at the appointed time, show up in the machine room carrying their programs on punched cards or tapes. They had the entire computer to themselves, and they were responsible for all machine operation. They loaded their assembly language programs into memory along with the assembler and, by punching some buttons on the console, started the translation process. Then they loaded their program into memory and started it running. Working with first-generation software was a lot like working on the naked machine described at the beginning of the chapter. It was attempted only by highly trained professionals intimately familiar with the computer and its operation.

System administrators quickly realized that this was a horribly inefficient way to use an expensive piece of equipment. (Remember that these early computers cost millions of dollars.) A programmer would sign up for an hour of computer time, but the majority of that time was spent analyzing results and trying to figure out what to do next. During this "thinking time," the system was idle and doing nothing of value. Eventually, the need to keep machines busy led to the development of a **second generation** of system software called **batch operating systems** (1955–1965).

Now That's Big!

The most widely used measure of program size is **source lines of code** (abbreviated SLOC). This is a count of the total number of non-blank, non-comment lines in a piece of software. According to Wikipedia, a well-known Internet encyclopedia (www.wikipedia.org), the estimated size of Microsoft's *Windows Vista*, one of the most widely used operating systems in the world, is 50 million SLOC. If you were to print out the entire Vista program, at 60 lines per printed page you would generate about 833,000 pages of output, or roughly the number of pages in 2,500 novels. If you were to store that output on a bookshelf, it would stretch for almost the length of an American football field.

It is estimated that the average programmer can produce about 40 lines of correct code per day. If that number is correct, then the Windows Vista operating system represents 1,250,000 person-days, or (at 240 working days per year) about 5,200 person years of effort.

In second-generation batch operating systems, rather than operate the machine directly, a programmer handed the program (typically entered on punched cards) to a trained computer operator, who grouped it into a "batch"— hence the name. After a few dozen programs were collected, the operator carried this batch of cards to a small I/O computer that put these programs on tape. This tape was carried into the machine room and loaded onto the "big" computer that actually ran the users' programs, one at a time, writing the results to yet another tape. During the last stage, this output tape was carried back to the I/O computer to be printed and handed to the programmer. The entire cycle is diagrammed in Figure 6.18.

This cycle may seem cumbersome and, from the programmer's point of view, it was. (Every programmer who worked in the late 1950s or early 1960s has horror stories about waiting many hours—even days—for a program to be returned, only to discover that there was a missing comma.) From the computer's point of view, however, this new batch system worked wonderfully, and system utilization increased dramatically. No longer were there delays while a programmer was setting up to perform an operation. There were no long periods of idleness while someone was mulling over what to do next. As soon as one job was either completed normally or halted because of an error, the computer went to the input tape, loaded the next job, and started execution. As long as there was work to be done, the computer was kept busy.

Because programmers no longer operated the machine, they needed a way to communicate to the operating system what had to be done, and these early batch operating systems were the first to include a **command language**, also called a **job control language**. This was a special-purpose language in which users wrote commands specifying to the operating system (or the human operator) what operations to perform on their programs. These commands were interpreted by the operating system, which initiated the proper action. The "receptionist/dispatcher" responsibility of the operating system had been born. A typical batch job was a mix of programs, data, and commands, as shown in Figure 6.19.

By the mid-1960s, the use of integrated circuits and other new technologies had boosted computational speeds enormously. The batch operating system just described kept only a single program in memory at any one time. If that job paused for a few milliseconds to complete an I/O operation (such as

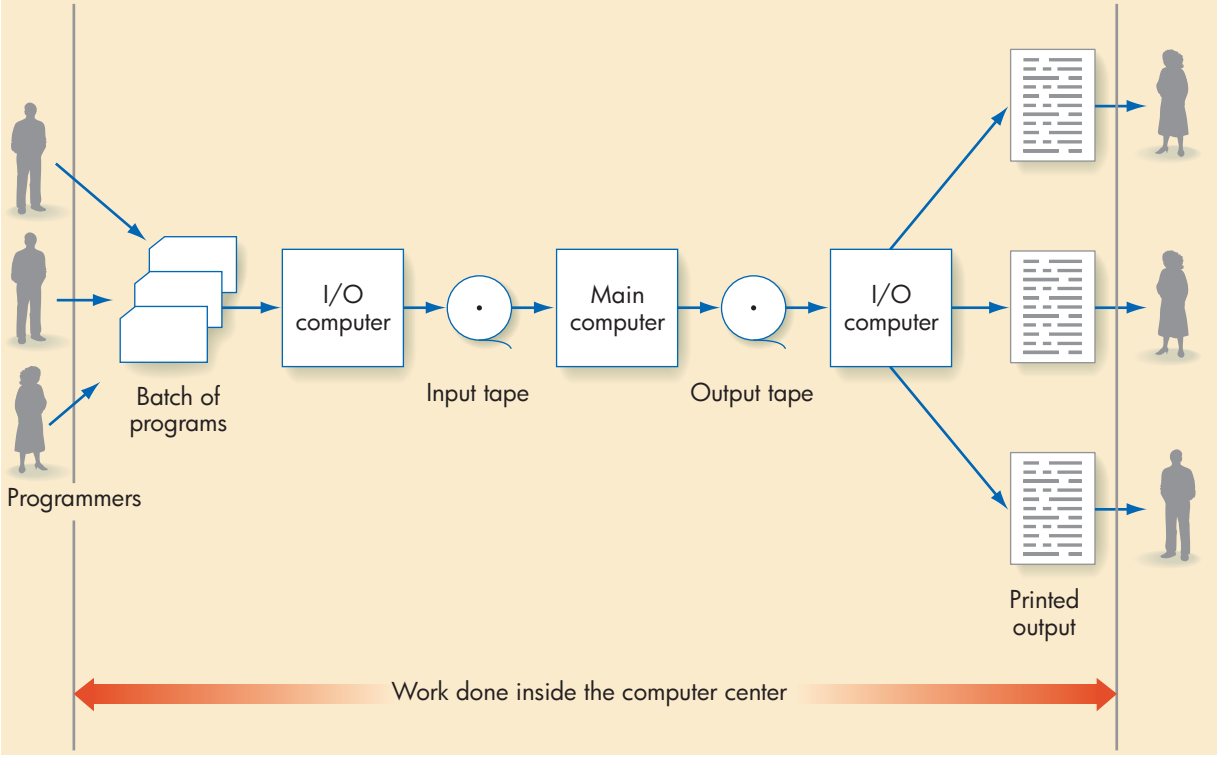
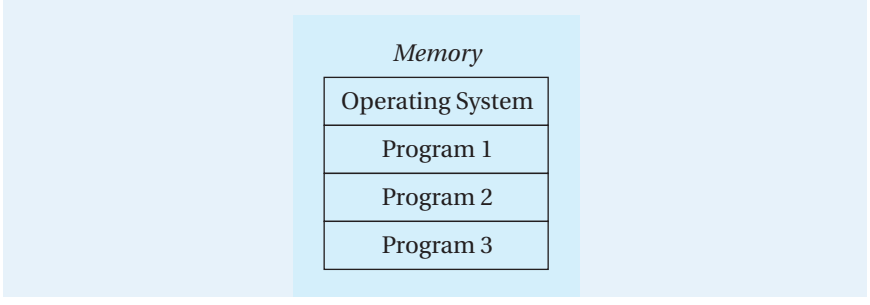


FIGURE 6.18

Operation of a Batch Computer System

read a disk sector or print a file on the printer), the processor simply waited. As computers became faster, designers began to look for ways to use those idle milliseconds. The answer they came up with led to a **third generation** of operating systems called **multiprogrammed operating systems** (1965–1985).

In a multiprogrammed operating system, many user programs are simultaneously loaded into memory, rather than just one:



If the currently executing program pauses for I/O, one of the other ready jobs is selected for execution so that no time is wasted. As we described earlier, this cycle of running/waiting/ready states led to significantly higher processor utilization.

To make this all work properly, the operating system had to protect user programs (and itself) from damage by other programs. When there was a single program in memory, the only user program that could be damaged was

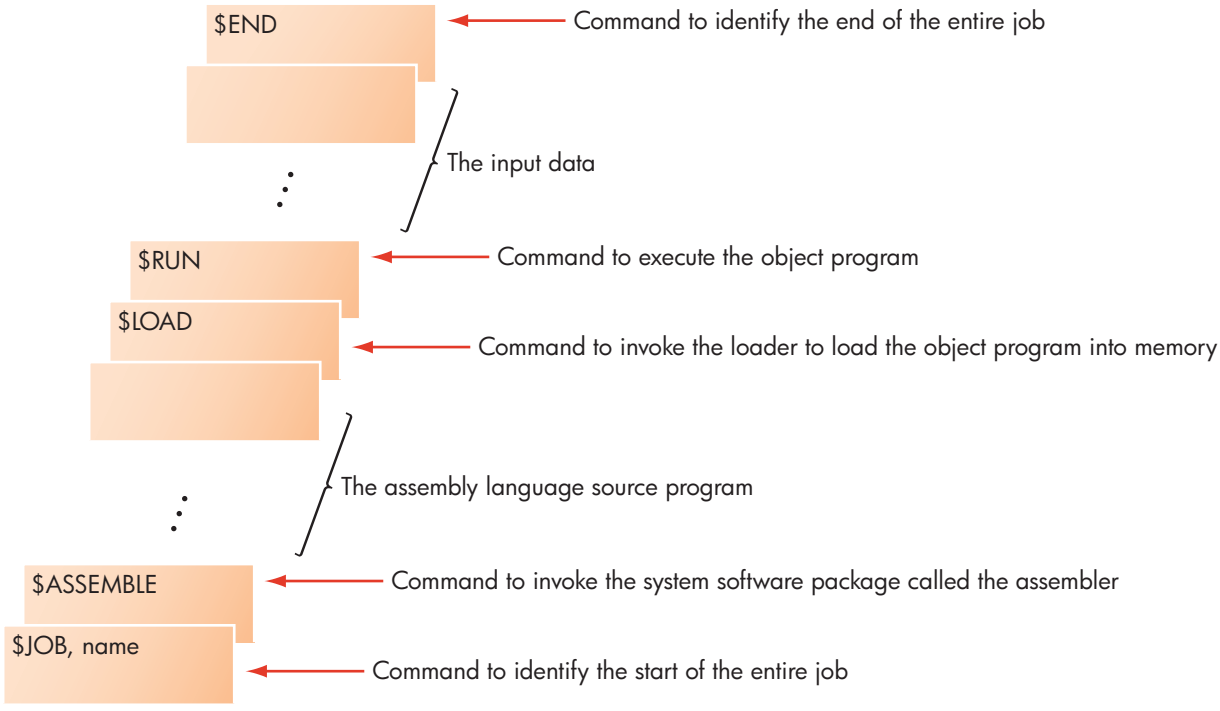


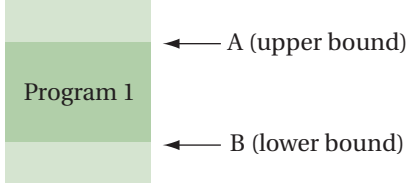
FIGURE 6.19
Structure of a Typical Batch Job

your own. Now, with many programs in memory, an erroneous instruction in one user's program could wreak havoc on any of the others. For example, the seemingly harmless instruction

```
STORE 1000    --Store the contents of register R into memory cell 1000.
```

should not be executed if the physical address 1000 is not located within this user's program. It could wipe out an instruction or piece of data in someone else's program, causing unexpected behavior and (probably) incorrect results.

These third-generation operating systems kept track of the upper and lower address bounds of each program in memory



and ensured that no program ever attempted a memory reference outside this range. If it did, then the system ceased execution of that program, produced an error message, removed that program from memory, and began work on another ready program.

Similarly, the operating system could no longer permit any program to execute a HALT instruction, because that would shut down the processor and prevent it from finishing any other program currently in memory.

These third-generation systems developed the concept of **user operation codes** that could be included in any user program, and **privileged operation codes** whose use was restricted to the operating system or other system software. The HALT instruction became a privileged op code that could be executed only by a system program, not by a user program.

These multiprogrammed operating systems were the first to have extensive protection and error detection capabilities, and the "traffic officer" responsibility began to take on much greater importance than in earlier systems.

During the 1960s and 1970s, computer networks and telecommunications systems (which are discussed in detail in Chapter 7) developed and grew rapidly. Another form of third-generation operating system evolved to take advantage of this new technology. It is called a **time-sharing** system, and it is a variation of the multiprogrammed operating system just described.

In a time-sharing system, many programs can be stored in memory rather than just one. However, instead of requiring the programmer to load all system commands, programs, and data in advance, a time-sharing system allows them to be entered **online**—that is, entered dynamically by users sitting at terminals and communicating interactively with the operating system. This configuration is shown in Figure 6.20.

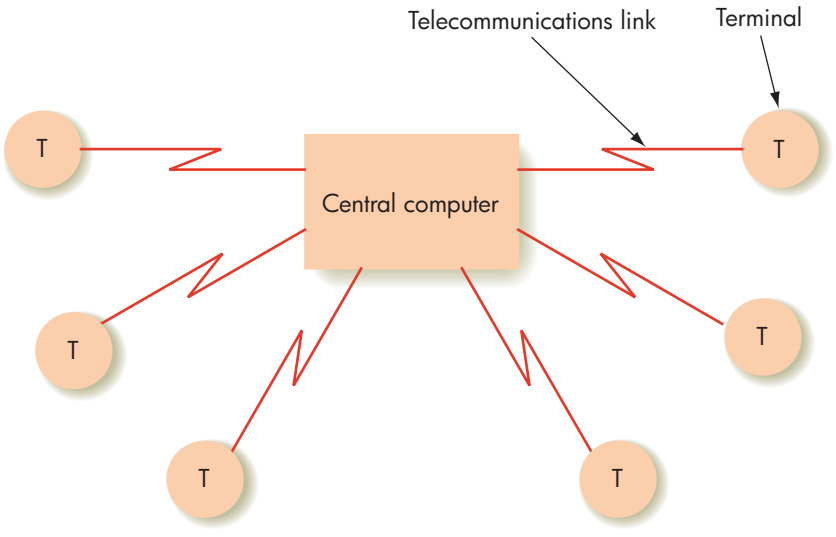
The terminals are connected to the central computer via communication links and can be located anywhere. This new system design freed users from the "tyranny of geography." No longer did they have to go to the computer to hand in their deck of cards; the services of the computer were delivered directly to them via their terminal. However, now that the walls and doors of the computer center no longer provided security and access control, the "security guard/watchman" responsibility became an extremely important part of operating system design. (We discuss the topic of computer security at length in Chapter 8.)

In a time-sharing system, a user would sit down at a terminal, log in, and initiate a program or make a request by entering a command:

```
>run MyJob
```

In this example, the program called MyJob would be loaded into memory and would compete for the processor with all other ready programs. When the

FIGURE 6.20
Configuration of a Time-Shared Computing System



program was finished running, the system would again display a prompt (">") and wait for the next command. The user could examine the results of the last program, think for a while, and decide what to do next, rather than having to determine the entire sequence of operations in advance. For example, say there is a mistake in the program and we want to correct it using a text editor. We can enter the command

```
>edit MyJob
```

which loads the text editor into memory, schedules it for execution, and causes the file system to load the file called MyJob.

However, one minor change was needed to make this new system work efficiently. In a "true" multiprogrammed environment, the only event, other than termination, that causes a program to be **suspended** (taken off the processor) is the execution of a slow I/O operation. What if the program currently executing is heavily **compute-bound**? That is, it does mostly computation and little or no I/O (for example, computing the value of π to a million decimal places). It could run for minutes or even hours before it is suspended and the processor is given to another program. During that time, all other programs would have to sit in the ready queue, waiting their turn. This is analogous to being in line at a bank behind someone depositing thousands of checks.

In a noninteractive environment this situation may be acceptable because no one is sitting at a terminal waiting for output. In fact, it may even be desirable, because a compute-bound job keeps the processor heavily utilized. In a time-sharing system, however, this waiting would be disastrous. There *are* users sitting at terminals communicating directly with the system and expecting an immediate response. If they do not get some type of response soon after entering a command, they may start banging on the keyboard and, eventually, give up. (Isn't that what you would do if the party at the other end of a telephone did not respond for several minutes?)

Therefore, to design a time-sharing system, we must make the following change to the multiprogrammed operating system described earlier. A program can keep the processor until *either* of the following two events occurs:

- It initiates an I/O operation.
- It has run for a maximum length of time, called a **time slice**.

Typically, this time slice is on the order of about a tenth of a second. This may seem like a minuscule amount of time, but it isn't. As we saw in Chapter 5, a typical time to fetch and execute a machine language instruction is about 1 nsec. Thus, in the 0.1-second time slice allocated to a program, a modern processor could execute roughly 100 million machine language instructions.

The basic idea in a time-sharing system is to service many users in a circular, round-robin fashion, giving each user a small amount of time and then moving on to the next. If there are not too many users on the system, the processor can get back to a user before he or she even notices any delay. For example, if there are five users on a system and each one gets a time slice of 0.1 second, a user will wait no more than 0.5 second for a response to a command. This delay would hardly be noticed. However, if 40 or 50 users were actively working on the system, they might begin to notice a 4- or 5-second delay and become irritated. (This is an example of the "virtual environment" created by the operating system *not* being helpful and supportive!) The number

of simultaneous users that can be serviced by a time-sharing system depends on (1) the speed of the processor, (2) the time slice given to each user, and (3) the type of operation each user is doing (i.e., how many use the full time slice, and how many stop before that).

Time sharing was the dominant form of operating system during the 1970s and 1980s, and time-sharing terminals appeared throughout government offices, businesses, and campuses.

The early 1980s saw the appearance of the first personal computers, and in many business and academic environments the "dumb" terminal began to be replaced by these newer PCs. Initially, the PC was viewed as simply another type of terminal, and during its early days it was used primarily to access a central time-sharing system. However, as PCs became faster and more powerful, people soon realized that much of the computing being done on the centralized machine could be done much more conveniently and cheaply by the microcomputers sitting on their desktops.

During the late 1980s and the 1990s, computing rapidly changed from the centralized environment typical of batch, multiprogramming, and timesharing systems to a **distributed environment** in which much of the computing was done remotely in the office, laboratory, classroom, and factory. Computing moved from the computer center out to where the real work was actually being done. The operating systems available for early personal computers were simple **single-user operating systems** that gave one user total access to the entire system. Because personal computers were so cheap, there was really no need for many users to share their resources, and the time-sharing and multiprogramming designs of the third generation became less important.

Although personal computers were relatively cheap (and were becoming cheaper all the time), many of the peripherals and supporting gear—laser printers, large disk drives, tape back-up units, and specialized software packages—were not. In addition, electronic mail was growing in importance, and standalone PCs were unable to communicate easily with other users and partake in this important new application. The personal computer era required a new approach to operating system design. It needed a virtual environment that supported both *local computation* and *remote access* to other users and shared resources.

This led to the development of a **fourth-generation** operating system called a **network operating system** (1985–present). A network operating system manages not only the resources of a single computer, but also the capabilities of a telecommunications system called a **local area network**, or **LAN** for short. (We will take a much closer look at these types of networks in Chapter 7.) A LAN is a network that is located in a geographically contiguous area such as a room, a building, or a campus. It is composed of personal computers (workstations), and special shared resources called **servers**, all interconnected via a high-speed link made of **coaxial** or **fiber-optic cable**. A typical LAN configuration is shown in Figure 6.21.

The users of the individual computers in Figure 6.21, called **clients**, can perform local computations without regard to the network. In this mode, the operating system provides exactly the same services described earlier: loading and executing programs and managing the resources of this one machine.

However, a user can also access any one of the shared network resources just as though it were local. These resources are provided by a computer called a *server* and can include a special high-quality color printer, a shared file system, or access to an international computer network. The system software

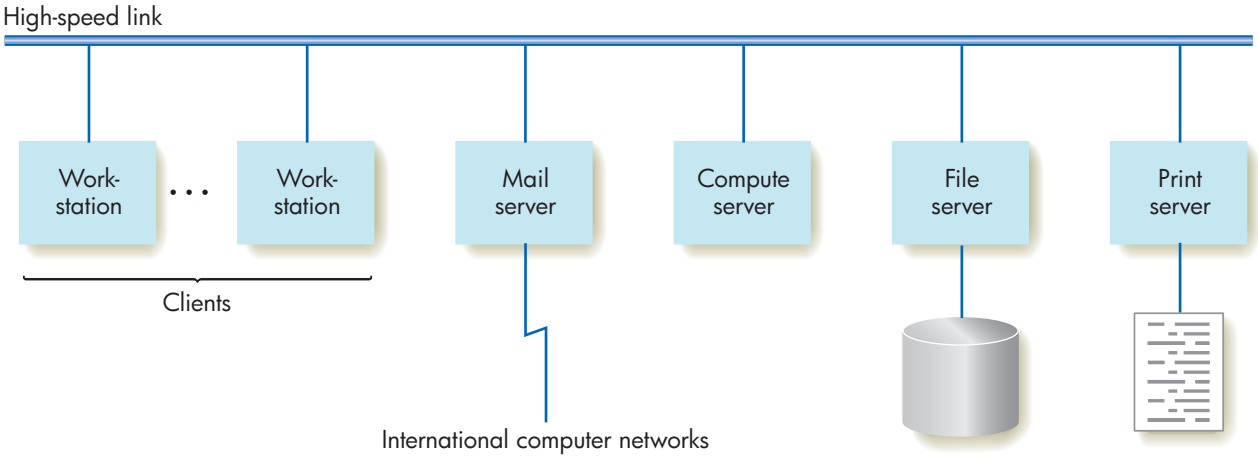


FIGURE 6.21
A Local Area Network

does all the work needed to access those resources, hiding the details of communication and competition with other nodes for this shared resource.

Network operating systems create a virtual machine that extends beyond the boundaries of the local system on which the user is working. They let us access a huge pool of resources—computers, servers, and users—exactly as though they were connected to our own computers. This fourth-generation virtual environment, exemplified by operating systems such as Windows NT, Windows Vista, Mac OS X, and Linux, is diagrammed in Figure 6.22.

One important variation of the network operating system is called a **real-time operating system**. During the 1980s and 1990s, computers got smaller and smaller, and it became common to place them inside other pieces of

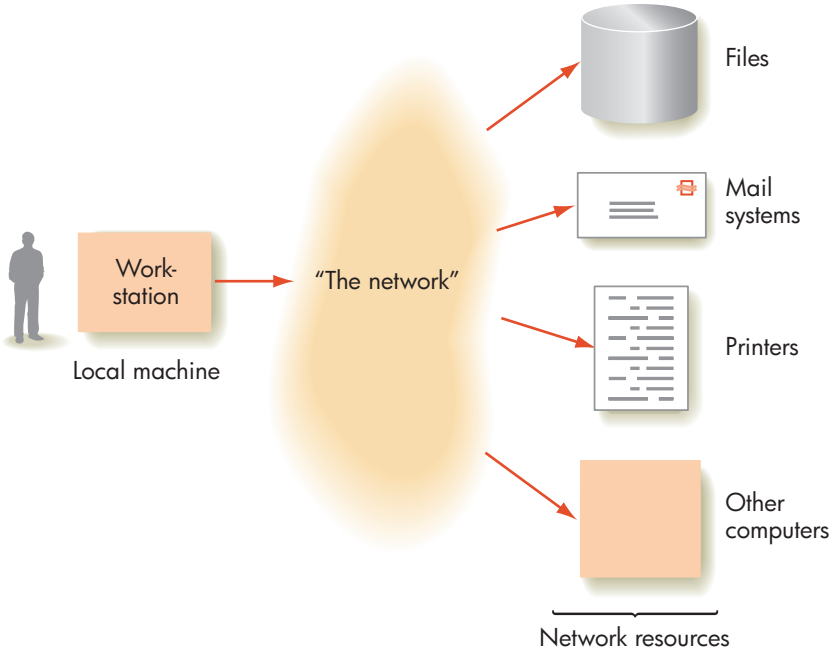


FIGURE 6.22
The Virtual Environment Created by a Network Operating System

equipment to control their operation. These types of computers are called **embedded systems**; examples include computers placed inside automobile engines, microwave ovens, thermostats, assembly lines, airplanes, and watches.

For example, the Boeing 787 Dreamliner jet contains hundreds of embedded computer systems inside its engines, braking system, wings, landing gear, and cabin. The central computer controlling the overall operation of the airplane is connected by a LAN to these embedded computers that monitor system functions and send status information.

In all the operating systems described thus far, we have implied that the system satisfies requests for services and resources in the order received. In some systems, however, certain requests are much more important than others, and when these important requests arrive, we must drop everything else to service them. Imagine that the central computer on our Boeing 787 receives two requests. The first request is from a cabin monitoring sensor that wants the central system to raise the cabin temperature a little for passenger comfort. The second message comes from the on-board collision detection system and says that another plane is approaching on the same flight path, and there is about to be a mid-air collision. It would like the central computer to take evasive action. Which request should be serviced next? Of course, the collision detection message, even though it arrived second.

A real-time operating system manages the resources of embedded computers that are controlling ongoing physical processes and that have requests that must be serviced within fixed time constraints. This type of operating system guarantees that it can service these important requests within that fixed amount of time. For example, it may guarantee that, regardless of what else it is currently doing, if a collision detection message arrives, the software implementing collision avoidance will be activated and executed within 50 milliseconds. Typically, the way that this guarantee is implemented is that all requests to a real-time operating system are **prioritized**. Instead of being handled in first-come, first-served order, they are handled in priority sequence, from most important to least important, where "importance" is defined in terms of the time-critical nature of the request. A real-time operating system lets passengers be uncomfortably cool for a few more seconds while it handles the problem of avoiding a mid-air collision.

6.4.3 *The Future*

The discussions in this chapter demonstrate that, just as there have been huge changes in hardware over the last 50 years, there have been equally huge changes in system software. We have progressed from a first-generation environment in which a user personally managed the computing hardware, using a complicated text-oriented command language, to current fourth-generation systems in which users can request services from anywhere in a network, using enormously powerful and easy-to-use graphical user interfaces.

And just as hardware capabilities continue to improve, there is a good deal of computer science research directed at further improving the high-level virtual environment created by a modern fourth-generation operating system. A fifth-generation operating system is certainly not far off.

These next-generation systems will have even more powerful user interfaces that incorporate not only text and graphics but photography, touch, sound, fax, video, and TV. These **multimedia user interfaces** will interact

with users and solicit requests in a variety of ways. Instead of point-and-click, a fifth-generation system might allow you to speak the command, "Please display my meeting schedule for May 6." The visual display may include separate windows for a verbal reminder about an important event and a digitally encoded photograph of the person with whom you are meeting. Just as text-only systems are now viewed as outmoded, today's text and graphics system may be viewed as too limiting for high-quality user/system interaction.

A fifth-generation operating system will typically be a **parallel processing operating system** that can efficiently manage computer systems containing tens, hundreds, or even thousands of processors. Such an operating system will need to recognize opportunities for parallel execution, send the separate tasks to the appropriate processor, and coordinate their concurrent execution, all in a way that is transparent to the user. On this virtual machine, a user will be unaware that multiple processors even exist except that programs run 10, 100, or 1,000 times faster. Without this type of software support, a massively parallel system would be a "naked parallel processor" just as difficult to work with as the "naked machine" discussed at the beginning of this chapter.

Finally, new fifth-generation operating systems will create a truly **distributed computing environment** in which users do not need to know the location of a given resource within the network. In current network operating systems, the details of how the communication is done are hidden, but the existence of separate nodes in the network are not (see Figure 6.22). The user is aware that a network exists and must specify the network node where the work is to be done. In a typical fourth-generation network operating system, a user issues the following types of commands:

- Access file F on file server S and copy it to my local system.
- Run program P on machine M.
- Save file F on file server T.
- Print file F on print server Q.

Compare these commands with the instructions the manager of a business gives to an assistant: "Get this job done. I don't care how or where. Just do it, and when you are done, give me the results." The details of how and where to get the job done are left to the underling. The manager is concerned only with results.

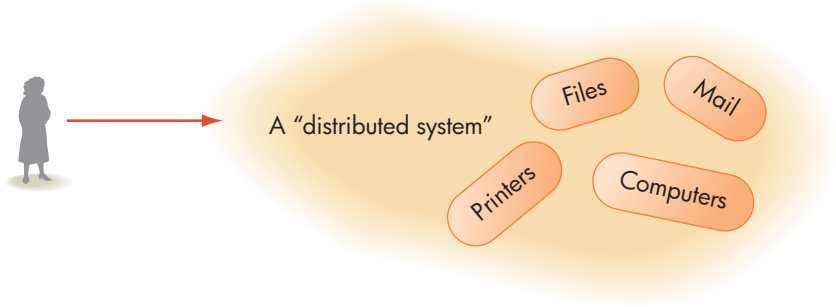
In a truly **distributed operating system**, the user is the manager and the operating system the assistant, and the user does not care where or how the system satisfies a request as long as it gets done correctly. The users of a distributed system do not see a network of distinct sites or "local" and "remote" nodes. Instead, they see a single logical system that provides resources and services. The individual nodes and the boundaries between them are no longer visible to the user, who thinks only in terms of *what* must be done, not *where* it will be done or *which* node will do it. This situation is diagrammed in Figure 6.23.

In a distributed operating system, the commands shown earlier might be expressed as follows:

- Access file F wherever it is located.
- Run program P on any machine currently available.
- Save file F wherever there is sufficient room.
- Print file F on any laser printer with 400 dpi resolution that is not in use.

FIGURE 6.23

Structure of a Distributed System



This is certainly the most powerful virtual environment we have yet described, and an operating system that creates such an environment would significantly enhance the productivity of all its users. These "fifth-generation dashboards" will make using the most powerful and most complex computer system as easy as driving a car—perhaps even easier. Surfing the Web gives us a good indication of what it will be like to work on a distributed system. When we click on a link we have no idea at all where that information is located and, moreover, we don't care. We simply want that page to appear on our screen. To us, the Web behaves like one giant logical system even though it is spread out across hundreds of countries and hundreds of millions of computers.

Figure 6.24 summarizes the historical evolution of operating systems, much as Figure 1.8 summarized the historical development of computer hardware.

FIGURE 6.24

Some of the Major Advances in Operating Systems Development

Generation	Approximate Dates	Major Advances
First	1945–1955	No operating system available Programmers operated the machine themselves
Second	1955–1965	Batch operating systems Improved system utilization Development of the first command language
Third	1965–1985	Multiprogrammed operating systems Time-sharing operating systems Increasing concern for protecting programs from damage by other programs Creation of privileged instructions and user instructions Interactive use of computers Increasing concern for security and access control First personal computer operating systems
Fourth	1985–present	Network operating systems Client-server computing Remote access to resources Graphical user interfaces Real-time operating systems Embedded systems
Fifth	??	Multimedia user interfaces Massively parallel operating systems Distributed computing environments

EXERCISES

1. Describe the user interface in other high-technology devices commonly found in the home or office, such as a DVD player, audio system, television, copier, or microwave oven. Pick one specific device and discuss how well its interface is designed and how easy it is to use. Does the device use the same techniques as computer system interfaces, such as menus and icons?

2. Can you think of situations where you might *want* to see the underlying hardware of the computer system? That is, you want to interact with the actual machine, not the virtual machine. How could you accomplish this? (Essentially, how could you bypass the operating system?)

3. Assume that you write a letter in English and have a friend translate it into Spanish. In this scenario, what is equivalent to the source program of Figure 6.4? The object program? The assembler?

4. Assume that memory cells 60 and 61 and register R currently have the following values:

Register R: 13
60: 472
61: –1

Using the instruction set in Figure 6.5, what is in register R and memory cells 60 and 61 after completion of each of the following operations? Assume that each instruction starts from the above conditions.

a. LOAD 60
b. STORE 60
c. ADD 60
d. COMPARE 61
e. IN 61 (Assume that the user enters a 50.)
f. OUT 61

5. Assume that memory cell 79 contains the value +6. In addition, the symbol Z is equivalent to memory location 79. What is placed in register R by each of the following load commands?

a. LOAD 79
b. LOAD 6
c. LOAD Z
d. LOAD Z + 1 (Assume that this is allowed.)

6. Say we accidentally execute the following piece of data:

```
.DATA 16387
```

Describe exactly what happens. Assume that the format of machine language instructions on this system is the same format shown in Figure 6.13.

7. What is the assembly language equivalent of each of the following binary machine language instructions? Assume the format described in Figure 6.13 and the numeric op code values shown in Figure 6.5.

a. 0101001100001100
b. 0011000000000111

8. Is the following data generation pseudo-op legal or illegal? Why?

```
THREE: .DATA 2
```

9. Using the instruction set shown in Figure 6.5, translate the following algorithmic primitives into assembly language code. Show all necessary .DATA pseudo-ops.

a. Add 3 to the value of *K*
b. Set *K* to (*L* + 1) – (*M* + *N*)
c. If *K* >10 then output the value of *K*
d. If (*K* > *L*) then output the value of *K* and increment *K* by 1
   otherwise output the value of *L* and increment *L* by 1
e. Set *K* to 1
   Repeat the next two lines until *K* > 100
   Output the value of *K*
   Increment *K* by 1
   End of the loop

10. What, if anything, is the difference between the following two sets of instructions?

```
LOAD X        INCREMENT X
ADD TWO       INCREMENT X
.
.
.
TWO: .DATA 2
```

11. Look at the assembly language program in Figure 6.8. Is the statement CLEAR SUM on line 2 necessary? Why or why not? Is the statement LOAD ZERO on line 4 necessary? Why or why not?

12. Modify the program in Figure 6.8 so that it separately computes and prints the sum of all positive numbers and all negative numbers and stops when it sees the value 0. For example, given the input

12, –2, 14, 1, –7, 0

your program should output the two values 27 (the sum of the three positive values 12, 14, and 1) and –9 (the sum of the two negative numbers –2 and –7) and then halt.

13. Write a complete assembly language program (including all necessary pseudo-ops) that reads in a series of integers, one at a time, and outputs the largest and smallest values. The input will consist of a list of integer values containing exactly 100 numbers.

14. Assume that we are using the 16 distinct op codes in Figure 6.5. If we write an assembly language program that contains 100 instructions and our processor can do about 50,000 comparisons per second, what is the maximum time spent doing operation code translation using:

a. Sequential search (Figure 2.9)

b. Binary search (Figure 3.19)

Which one of these two algorithms would you recommend using? Would your conclusions be significantly different if we were programming in an assembly language with 300 op codes rather than 16? If our program contained 50,000 instructions rather than 100?

15. What value is entered in the symbol table for the symbols AGAIN, ANS, X, and ONE in the following program? (Assume that the program is loaded beginning with memory location 0.)

```
            .BEGIN
            --Here is the program.
            IN          X
            LOAD        X
AGAIN:      ADD         ANS
            SUBTRACT    ONE
            STORE       ANS
            OUT         ANS
            JUMP        AGAIN
            --Here are the data.
ANS:        .DATA       0
X:          .DATA       0
ONE:        .DATA       1
            .END
```

16. Look at the assembly language program in Figure 6.8. Determine the physical memory address associated with every label in the symbol table. (Assume that the program is loaded beginning with memory location 0.)

17. Is the following pair of statements legal or illegal? Explain why.

```
LABEL:  .DATA   3
LABEL:  .DATA   4
```

If it is illegal, will the error be detected during pass 1 or pass 2 of the assembly process?

18. What are some drawbacks in using passwords to limit access to a computer system? Describe some other possible ways that an operating system could limit access. In what type of application might these alternative safeguards be appropriate?

19. Why are authorization lists so sensitive that they must be encrypted and protected from unauthorized change? What kind of damage can occur if these files are modified in unexpected or unplanned ways?

20. Assume that any individual program spends about 50% of its time waiting for I/O operations to be completed. What percentage of time is the processor doing useful work (called **processor utilization**) if there are three programs loaded into memory? How many programs should we keep in memory if we want processor utilization to be at least 95%?

21. Here is an algorithm for calling a friend on the telephone:

Step	*Operation*
1.	Dial the phone and wait for either an answer or a busy signal
2.	If the line is not busy then do steps 3 and 4
3.	Talk as long as you want
4.	Hang up the phone, you are done
5.	Otherwise (the line is busy)
6.	Wait exactly 1 minute
7.	Go back to step 1 and try again

During execution this algorithm could get into a situation where, as in the deadlock problem, no useful work can ever get done. Describe the problem, explain why it occurs, and suggest how it could be solved.

22. Explain why a batch operating system would be totally inadequate to handle such modern applications as airline reservations and automated teller machines.

23. In a time-sharing operating system, why is system performance so sensitive to the value that is selected for the time slice? Explain what type of system behavior would occur if the value selected for the time slice were too large? Too small?

24. As hardware (processor/memory) costs became significantly cheaper during the 1980s and 1990s, time-sharing became a much less attractive design for operating systems. Explain why this is the case.

25. Determine whether the computer system on which you are working is part of a local area network. If it is, determine what servers are available and how they are used. Is there a significant difference between the ways you access local resources and remote resources?

26. The following four requests could come in to an operating system as it is running on a computer system:

- The clock in the computer has just "ticked," and we need to update a seconds counter.
- The program running on processor 2 is trying to perform an illegal operation code.
- Someone pulled the plug on the power supply, and the system will run out of power in 50 msec.
- The disk has just read the character that passed under the read/write head, and it wants to store it in memory before the next one arrives.

In what order should the operating system handle these requests?

CHALLENGE WORK

1. In Chapter 2 we wrote a number of algorithms that worked on a list of values rather than a single value. That is, our algorithm contained statements such as

Get values for A_1, A_2, . . . , A_N -the list to be searched

In statements like this we are dealing with individual data items such as A_1 and A_2, but they are also part of a collection of items, the list called A. A collection of related data items is called a **data structure**. High-level programming languages like C++, Java, and Python provide users with a rich collection of data structures that go by such names as arrays, sets, and lists. We can program with these structures just as though they were an inherent part of the hardware of the computer. However, the discussions in the previous two chapters have shown that data structures such as lists of numbers do *not* exist directly in hardware. There are no machine language instructions that can carry out the type of algorithmic command shown in the pseudocode statement above. When you write an instruction that uses a structure such as a list, the language translator (that is, the assembler or compiler) must map it into what is available on the hardware—the machine language instruction set shown in Figure 5.19 and the sequential addresses in our memory. (This is another good example of the virtual environment created by a piece of system software.)

Write an assembly language program to sum up a list of 50 numbers that are read in and stored in memory. Here is the algorithm you are to translate:

Read in 50 numbers A_1, A_2, . . . , A_{50}
Set *Sum* to 0
Set *i* to 1
While the value of *i* is less than or equal to 50
    Sum = *Sum* + A_i
    i = *i* + *1*
End of the loop
Write out the value of Sum
Stop

To implement this algorithm, you must simulate the concept of a list of numbers using the assembly language resources that are available. (*Hint:* Remember that in the Von Neumann architecture there is no distinction between an instruction and a piece of data. Therefore, an assembly language instruction such as LOAD A can be treated as data and modified by other instructions.)

FOR FURTHER READING

Here are some excellent introductory texts on the design and implementation of operating systems. Most of them also include a discussion of some specific modern operating system such as Linux, Mac OS X, or Windows Vista.

Silbershatz, A.; Galvin, P.; and Gagne, G. *The Design of Operating Systems,* 7th ed. New York: Wiley, 2004.

Stallings, W. *Operating Systems: Internals and Design Principles,* 6th ed. Englewood Cliffs, NJ: Prentice-Hall, 2008.

For a discussion of future directions in operating system design, especially network and distributed operating systems, here are a couple of excellent references:

Coulouris, G.; Dollimore, J.; and Kindberg, T. *Distributed Systems: Concepts and Design,* 4th ed. Reading, MA: Addison Wesley, 2005.

Tanenbaum, A.; and VanSteen, M. *Distributed Systems: Principles and Paradigms,* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2006.

Finally, here is an excellent general reference on system software:

Clarke, D., and Merusi, D. *System Software: The Way Things Work*. Englewood Cliffs, NJ: Prentice-Hall, 1998.