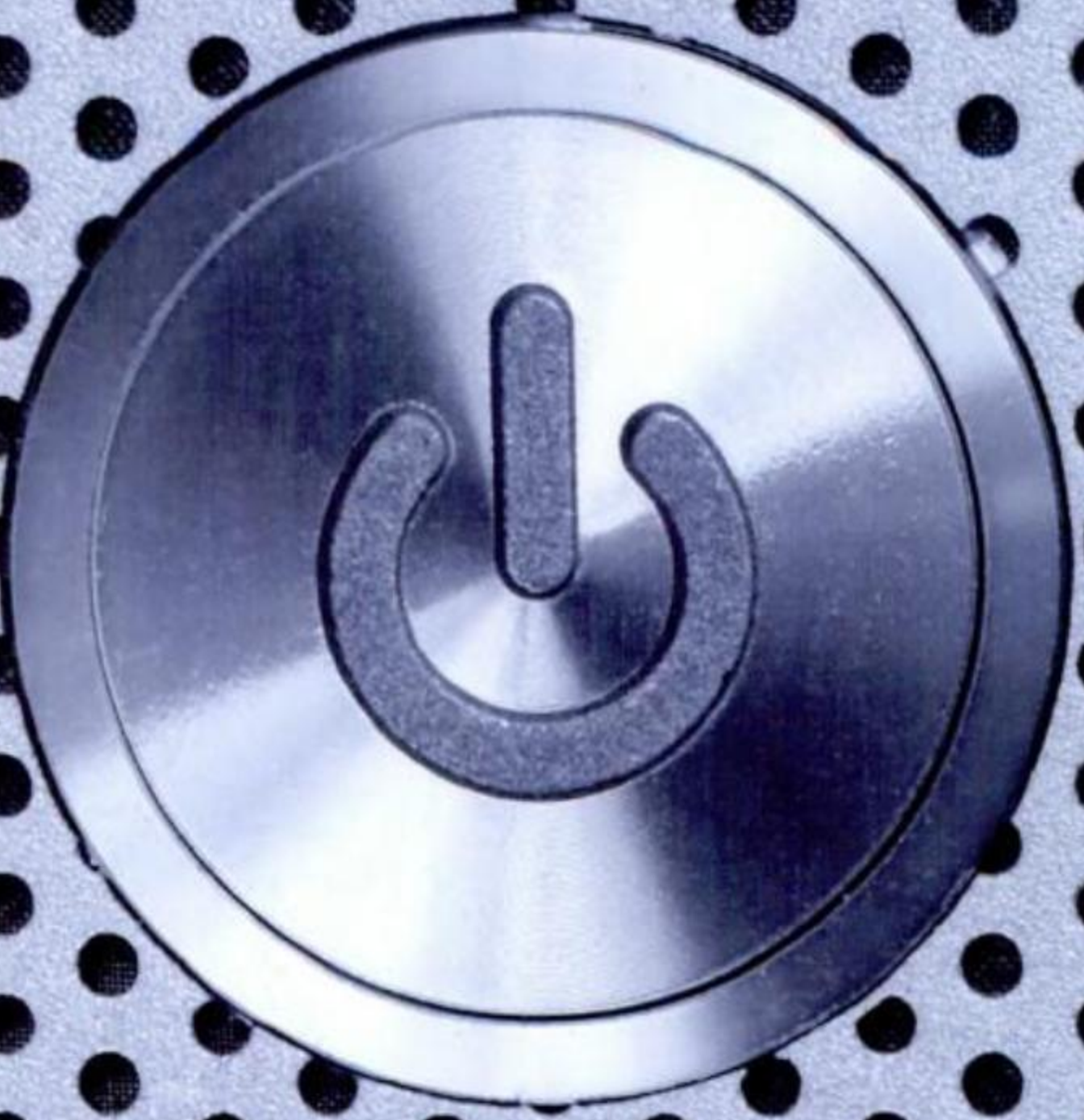
THOMSON



foundations of computer science

Behrouz Forouzan and Firouz Mosharraf

second edition

Visit the website at: **www.thomsonlearning.co.uk/forouzan**

Copyrighted material

1.1 TURING MODEL

The idea of a universal computational device was first described by Alan Turing in 1937. He proposed that all computation could be performed by a special kind of a machine, now called a *Turing machine*. Although Turing presented a mathematical description of such a machine, he was more interested in the philosophical definition of computation than in building the actual machine. He based the model on the actions that people perform when involved in computation. He abstracted these actions into a model for a computational machine that has really changed the world.

Data processors

Before discussing the Turing model, let us define a computer as a **data processor**. Using this definition, a computer acts as a black box that accepts input data, processes the data, and creates output data (Figure 1.1). Although this model can define the functionality of a computer today, it is too general. In this model, a pocket calculator is also a computer (which it is, in a literal sense).

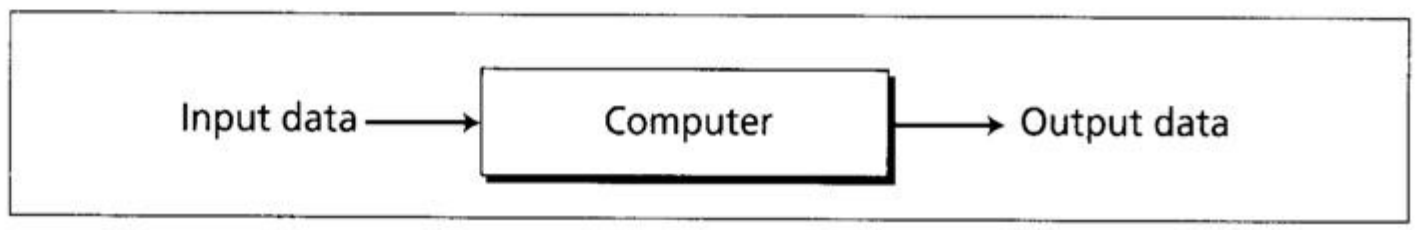


Figure 1.1 A single purpose computing machine

Another problem with this model is that it does not specify the type of processing, or whether more than one type of processing is possible. In other words, it is not clear how many types or sets of operations a machine based on this model can perform. Is it a specific-purpose machine or a general-purpose machine?

This model could represent a specific-purpose computer (or processor) that is designed to do a single job, such as controlling the temperature of a building or controlling the fuel usage in a car. However, computers, as the term is used today, are *general-purpose* machines. They can do many different types of tasks. This implies that we need to change this model into the Turing model to be able to reflect the actual computers of today.

Programmable data processors

The Turing model is a better model for a general-purpose computer. This model adds an extra element to the specific computing machine: the *program*. A **program** is a set of instructions that tells the computer what to do with data. Figure 1.2 shows the Turing model.

In the Turing model, the **output data** depends on the combination of two factors: the **input data** and the program. With the same input data, we can generate different outputs if we change the program. Similarly, with the same program, we can generate different outputs if we change the input data. Finally, if the input data

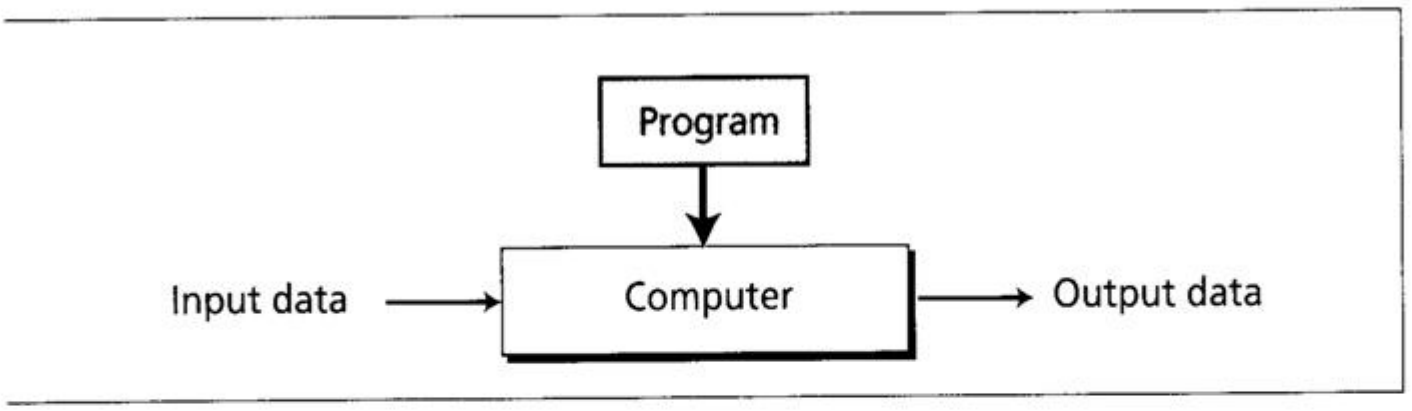


Figure 1.2 A computer based on the Turing model: programmable data processor

and the program remain the same, the output should be the same. Let us look at three cases.

Figure 1.3 shows the same sorting program with different input data. Although the program is the same, the outputs are different, because different input data is processed.

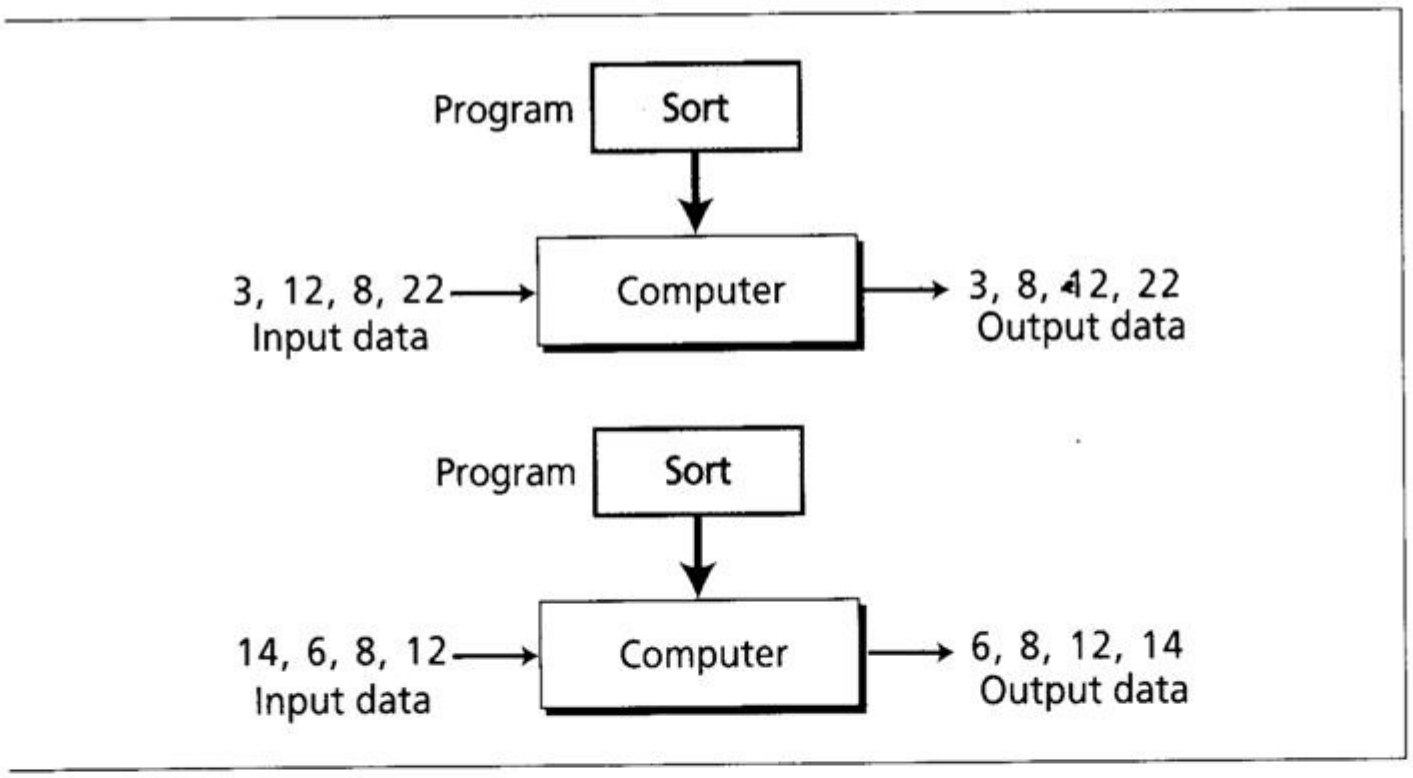


Figure 1.3 The same program, different data

Figure 1.4 shows the same input data with different programs. Each program makes the computer perform different operations on the input data. The first program sorts the data, the second adds the data, and the third finds the smallest number.

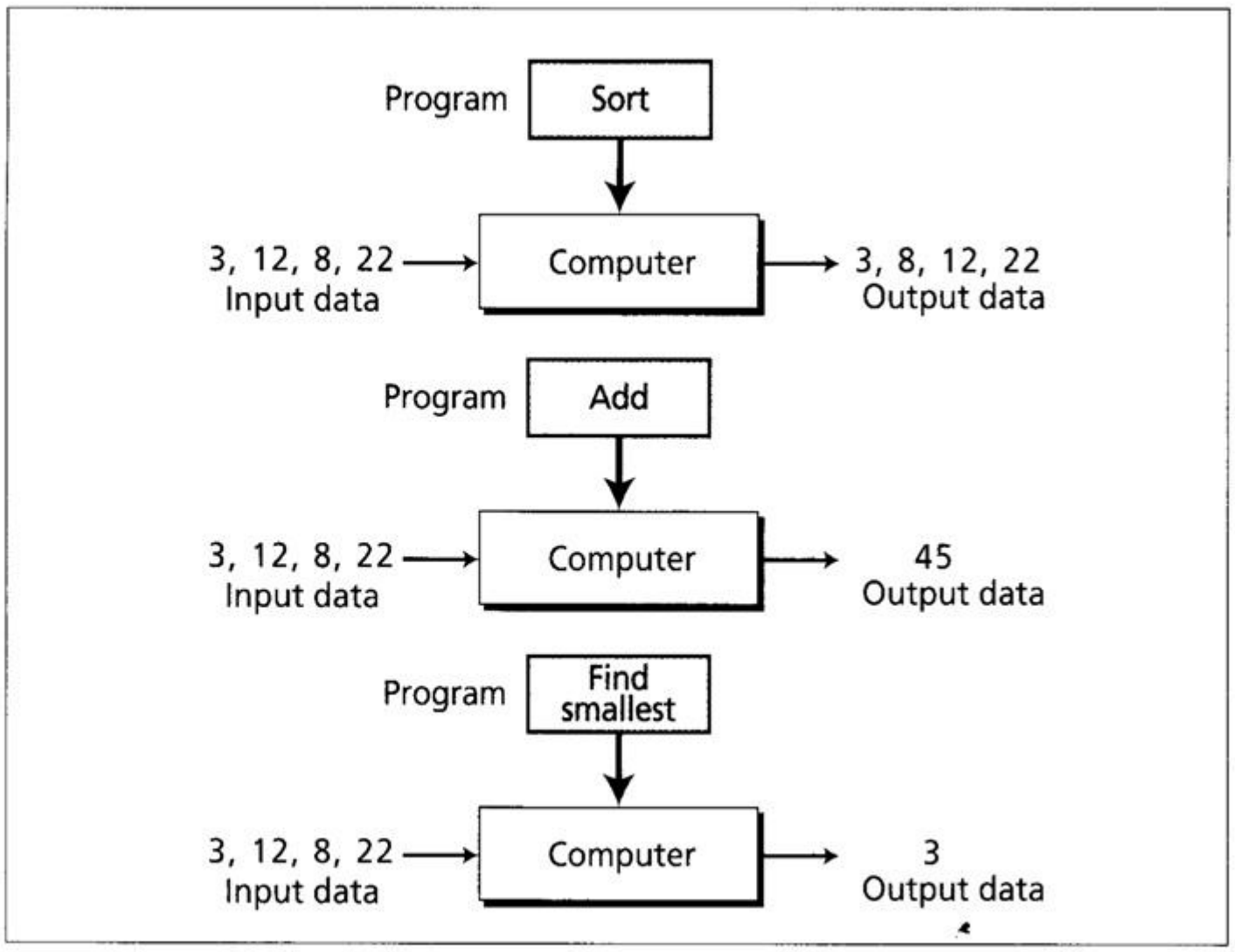


Figure 1.4 The same data, different programs

Same input data, same program

We expect the same result each time if both input data and the program are the same, of course. In other words, when the same program is run with the same input data, we expect the same output.

The universal Turing machine

A *universal Turing machine,* a machine that can do any computation if the appropriate program is provided, was the first description of a modern computer. It can be proved that a very powerful computer and a universal Turing machine can compute the same thing. We need only provide the data and the program—the description of how to do the computation—to either machine. In fact, a universal Turing machine is capable of computing anything that is computable.

1.2 VON NEUMANN MODEL

Computers built on the Turing universal machine store data in their memory. Around 1944–1945, John von Neumann proposed that, since program and data are logically the same, programs should also be stored in the memory of a computer.

Four subsystems

Computers built on the **von Neumann model** divide the computer hardware into four subsystems: memory, arithmetic logic unit, control unit, and input/output (Figure 1.5).

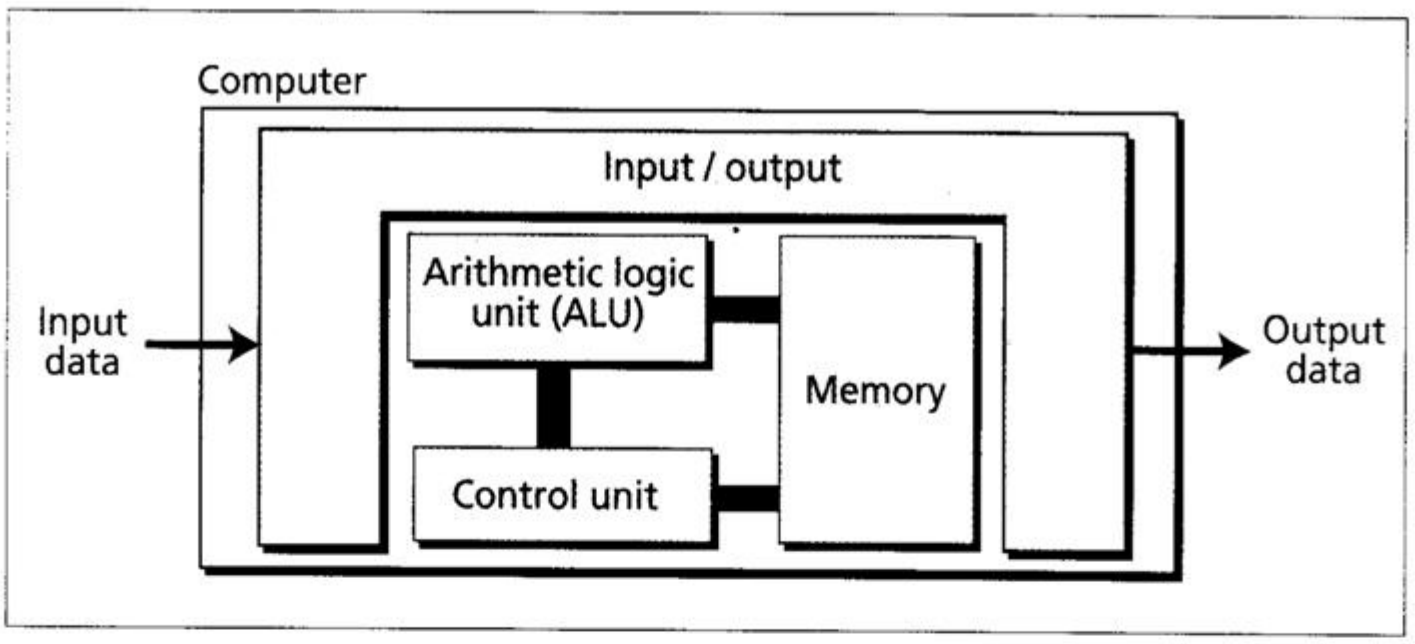


Figure 1.5 The von Neumann model

Memory

Memory is the storage area. This is where programs and data are stored during processing. We discuss the reasons for storing programs and data later in the chapter.

Arithmetic logic unit

The **arithmetic logic unit (ALU)** is where calculation and logical operations take place. For a computer to act as a data processor, it must be able to do arithmetic operations on data (such as adding a list of numbers). It should also be able to do logical operations on data, as we will see in Chapter 4.

Control unit

The **control unit** controls the operations of the memory, ALU, and the input/output subsystem.

Input / output

The input subsystem accepts input data and the program from outside the computer, while the output subsystem sends the results of processing to the outside world. The definition of the input/output subsystem is very broad: it also includes secondary storage devices such as disk or tape that store data and programs for processing. When a disk stores data that results from processing, it is considered an output device: when data is read from the disk, it is considered an input device.

The stored program concept

The von Neumann model states that the program must be stored in memory. This is totally different from the architecture of early computers in which only the data was stored in memory: the programs for their tasks were implemented by manipulating a set of switches or by changing the wiring system.

The memory of modern computers hosts both a program and its corresponding data. This implies that both the data and programs should have the same format, because they are stored in memory. In fact, they are stored as *binary* patterns in memory—a sequence of 0s and 1s.

Sequential execution of instructions

A program in the von Neumann model is made of a finite number of **instructions**. In this model, the control unit fetches one instruction from memory, decodes it,

then executes it. In other words, the instructions are executed one after another. Of course, one instruction may request the control unit to jump to some previous or following instruction, but this does not mean that the instructions are not executed sequentially. Sequential execution of a program was the initial requirement of a computer based on the von Neumann model. Today's computers execute programs in the order that is most efficient.

1.3 COMPUTER COMPONENTS

We can think of a computer as being made up of three components: computer hardware, data, and computer software.

Computer hardware

Computer hardware today has four components under the von Neumann model, although we can have different types of memory, different types of input/output subsystems, and so on. We discuss computer hardware in more detail in Chapter 5.

Data

The von Neumann model clearly defines a computer as a data processing machine that accepts the input data, processes it, and outputs the result.

Storing data

The von Neumann model does not define how data must be stored in a computer. If a computer is an electronic device, the best way to store data is in the form of an electrical signal, specifically its presence or absence. This implies that a computer can store data in one of two states.

Obviously, the data we use in daily life is not just in one of two states. For example, our numbering system uses digits that can take one of ten states (0 to 9). We cannot (as yet) store this type of information in a computer: it needs to be changed to another system that uses only two states (0 and 1). We also need to be able to process other types of data (text, image, audio, video). These also cannot be stored in a computer directly, but need to be changed to the appropriate form (0s and 1s).

In Chapters 3, we will learn how to store different types of data as a binary pattern, a sequence of 0s and 1s. In Chapter 4, we show how data is manipulated, as a binary pattern, inside a computer.

Organizing data

Although data should be stored in only one form inside a computer, a binary pattern, data outside a computer can take many forms. In addition, computers (and the notion of data processing) have created a new field of study known as *data organization*, which asks the question: can we organize our data into different entities and formats before storing it inside a computer? Today, data is not treated as a flat sequence of information. Instead, data is organized into small units, small units are organized into larger units, and so on. We will look at data from this point of view in Chapters 11–14.

Computer software

The main feature of the Turing or von Neumann models is the concept of the *program*. Although early computers did not store the program in the computer's memory, they did use the concept of programs. *Programming* those early computers meant changing the wiring systems or turning a set of switches on or off. Programming was therefore a task done by an operator or engineer before the actual data processing began.

Programs must be stored

In the von Neumann model programs are stored in the computer's memory. Not only do we need memory to hold data, but we also need memory to hold the program (Figure 1.6).

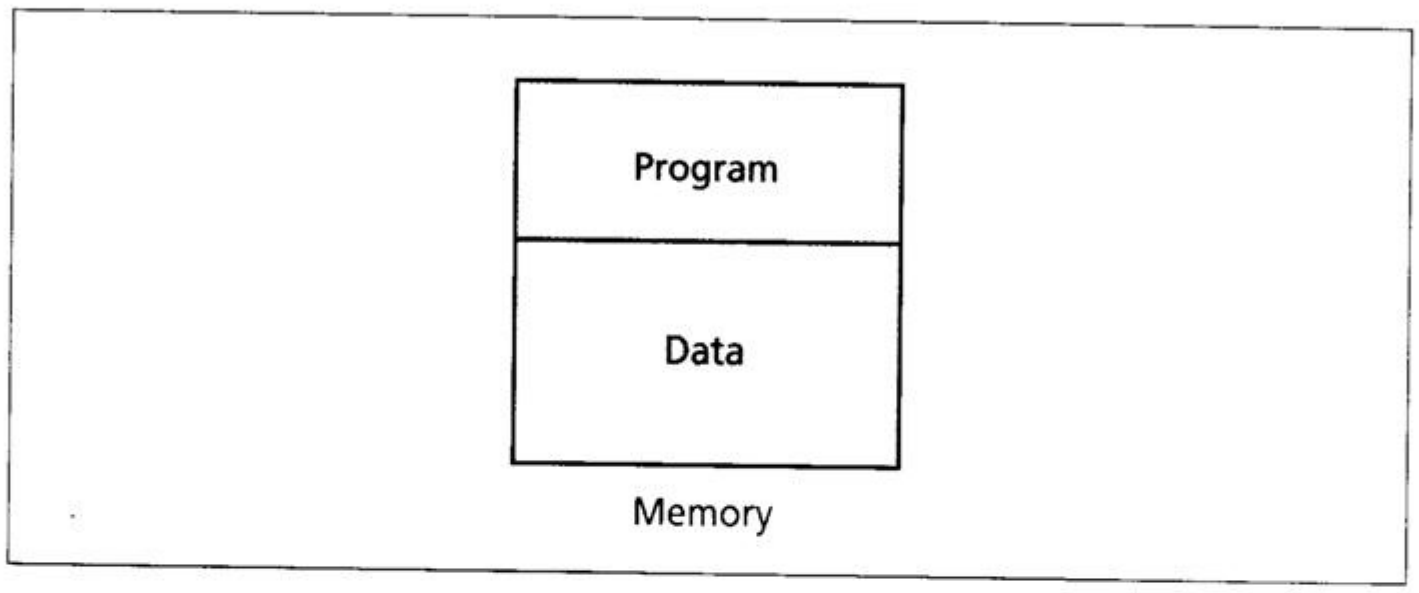


Figure 1.6 Program and data in memory

A sequence of instructions

Another requirement of the model is that the program must consist of a sequence of instructions. Each instruction operates on one or more data items. Thus, an instruction can change the effect of a previous instruction. For example, Figure 1.7 shows a program that inputs two numbers, adds them, and prints the result. This program consists of four individual instructions.

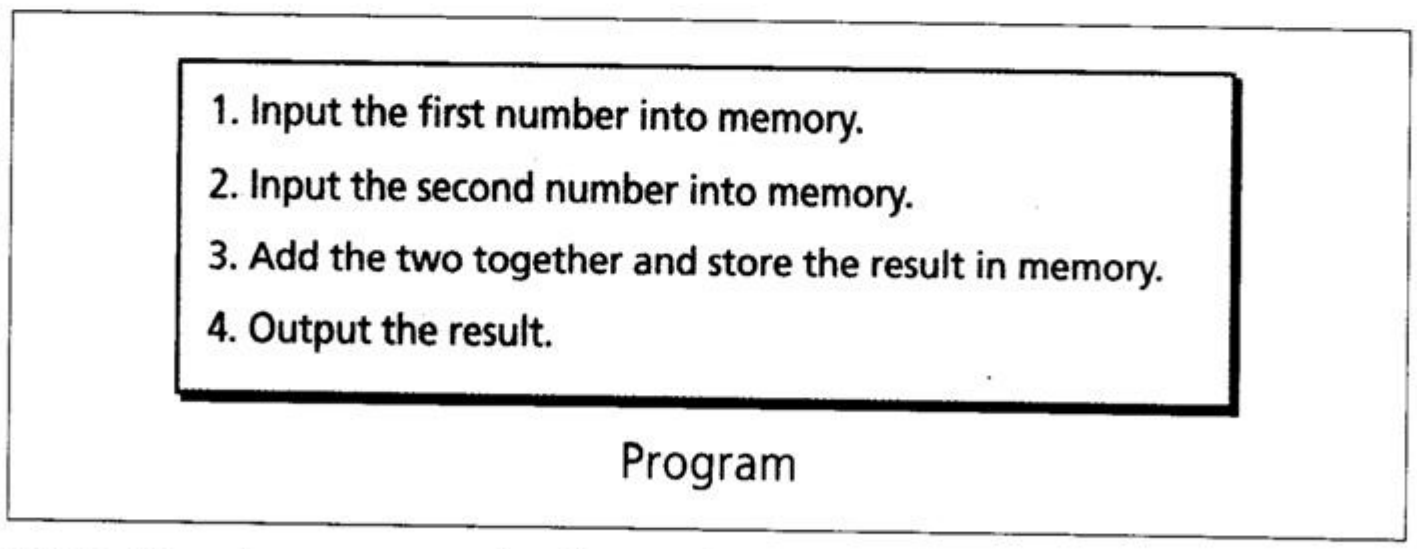


Figure 1.7 A program made of instructions

We might ask why a program must be composed of instructions. The answer is reusability. Today, computers do millions of tasks. If the program for each task was

an independent entity without anything in common with other programs, programming would be difficult. The Turing and von Neumann models make programming easier by defining the different instructions that can be used by computers. A programmer can then combine these instructions to make any number of programs. Each program can be a different combination of different instructions.

Algorithms

The requirement for a program to consist of a sequence of instructions made programming possible, but it brought another dimension to using a computer. A programmer must not only learn the task performed by each instruction, but also learn how to combine these instructions to do a particular task. Looking at this issue differently, a programmer must first solve the problem in a step-by-step manner, then try to find the appropriate instruction (or series of instructions) to implement those steps. This step-by-step solution is called an **algorithm**. Algorithms play a very important role in computer science and are discussed in Chapter 8.

Languages

At the beginning of the computer age there was only one computer language, *machine language*. Programmers wrote instructions (using binary patterns) to solve a problem. However, as programs became larger, writing long programs using these patterns became tedious. Computer scientists came up with the idea of using symbols to represent binary patterns, just as people use symbols (words) for commands in daily life. Of course, the symbols used in daily life are different than those used in computers. So the concept of **computer languages** was born. A natural language such as English is rich and has many rules to combine words correctly: a computer language, on the other hand, has a more limited number of symbols and also a limited number of words. We will study computer languages in Chapter 9.

Software engineering

Something that was not defined in the von Neumann model is **software engineering**, which is the design and writing of *structured programs*. Today it is not acceptable just to write a program that does a task: the program must follow strict rules and principles. We discuss these principles, collectively known as *software engineering*, in Chapter 10.

Operating systems

During the evolution of computers, scientists became aware that there was a series of instructions common to all programs. For example, instructions to tell a computer where to receive data and where to send data are needed by almost all programs. It is more efficient to write these instructions only once for the use of all programs. Thus the concept of the **operating system** emerged. An operating system originally worked as a manager to facilitate access to the computer's components by a program, although today operating systems do much more. We will learn about them in Chapter 7.

1.4 HISTORY

In this section we briefly review the history of computing and computers. We divide this history into three periods.

Mechanical machines (before 1930)

During this period, several computing machines were invented that bear little resemblance to the modern concept of a computer.

- In the 17th century, Blaise Pascal, a French mathematician and philosopher, invented Pascaline, a mechanical calculator for addition and subtraction operations. In the 20th century, when Niklaus Wirth invented a structured programming language, he called it Pascal to honor the inventor of the first mechanical calculator.
- In the late 17th century, German mathematician Gottfried Leibnitz invented a more sophisticated mechanical calculator that could do multiplication and division as well as addition and subtraction. It was called Leibnitz' Wheel.
- The first machine that used the idea of storage and programming was the Jacquard loom, invented by Joseph-Marie Jacquard at the beginning of the 19th century. The loom used punched cards (like a stored program) to control the raising of the warp threads in the manufacture of textiles.
- In 1823, Charles Babbage invented the Difference Engine, which could do more than simple arithmetic operations—it could solve polynomial equations too. Later, he invented a machine called the Analytical Engine that, to some extent, parallels the idea of modern computers. It had four components: a mill (corresponding to a modern ALU), a store (memory), an operator (control unit), and output (input/output).
- In 1890, Herman Hollerith, working at the US Census Bureau, designed and built a programmable machine that could automatically read, tally, and sort data stored on punched cards.

The birth of electronic computers (1930–1950)

Between 1930 and 1950, several computers were invented by scientists who could be considered the pioneers of the electronic computer industry.

Early electronic computers

The early computers of this period did not store the program in memory—all were programmed externally. Five computers were prominent during these years:

- The first special-purpose computer that encoded information electrically was invented by John V. Atanasoff and his assistant Clifford Berry in 1939. It was called the ABC (Atanasoff Berry Computer) and was specifically designed to solve a system of linear equations.
- At the same time, a German mathematician called Konrad Zuse designed a general-purpose machine called Z1.
- In the 1930s, the US Navy and IBM sponsored a project at Harvard University under the direction of Howard Aiken to build a huge computer called Mark I. This computer used both electrical and mechanical components.
- In England, Alan Turing invented a computer called Colossus that was designed to break the German Enigma code.

- The first general-purpose, totally electronic computer was made by John Mauchly and J. Presper Eckert and was called ENIAC (Electronic Numerical Integrator and Calculator). It was completed in 1946. It used 18,000 vacuum tubes, was 100 feet long by 10 feet high, and weighed 30 tons.

Computers based on the von Neumann model

The preceding five computers used memory only for storing data, and were programmed externally using wires or switches. John von Neumann proposed that the program and the data should be stored in memory. That way, every time we use a computer to do a new task, we need only change the program instead of rewiring the machine or turning hundreds of switches on and off.

The first computer based on von Neumann's ideas was made in 1950 at the University of Pennsylvania and was called EDVAC. At the same time, a similar computer called EDSAC was built by Maurice Wilkes at Cambridge University in England.

Computer generations (1950–present)

Computers built after 1950 more or less follow the von Neumann model. They have become faster, smaller, and cheaper, but the principle is almost the same. Historians divide this period into generations, with each generation witnessing some major change in hardware or software (but not in the model).

First generation

The first generation (roughly 1950–1959) is characterized by the emergence of commercial computers. During this time, computers were used only by professionals. They were locked in rooms with access limited only to the operator or computer specialist. Computers were bulky and used vacuum tubes as electronic switches. At this time, computers were affordable only by big organizations.

Second generation

Second-generation computers (roughly 1959–1965) used transistors instead of vacuum tubes. This reduced the size of computers, as well as their cost, and made them affordable to small and medium-size corporations. Two high-level programming languages, FORTRAN and COBOL (see Chapter 9), were invented and made programming easier. These two languages separated the programming task from the computer operation task. A civil engineer, for example could write a FORTRAN program to solve a problem without being involved in the electronic details of computer architecture.

Third generation

The invention of the **integrated circuit** (transistors, wiring, and other components on a single chip) reduced the cost and size of computers even further. *Minicomputers* appeared on the market. Canned programs, popularly known as *software packages*, became available. A small corporation could buy a package, for example for accounting, instead of writing its own program. A new industry, the software industry, was born. This generation lasted roughly from 1965 to 1975.

Fourth generation

The fourth generation (approximately 1975–1985) saw the appearance of *microcomputers*. The first desktop calculator, the Altair 8800, became available in 1975. Advances in the electronics industry allowed whole computer subsystems to fit on

a single circuit board. This generation also saw the emergence of computer networks (see Chapter 6).

Fifth generation

This open-ended generation started in 1985. It has witnessed the appearance of laptop and palmtop computers, improvements in secondary storage media (CD-ROM, DVD and so on), the use of multimedia, and the phenomenon of virtual reality.

1.5 SOCIAL AND ETHICAL ISSUES

Computer science has created some peripheral issues, the most prevalent of which can be categorized as social and ethical issues.

Social issues

Computers have created some controversy. We introduce some of these arguments here.

Dependency

Computer science has definitely changed our society. Based on some surveys, more than half of the households in the US use a computer to access the Internet. Does this mean that a society in which the majority of people have access to a computer is considered a *better* society? Some people think that the use of computers is inevitable in everyone's life: life is more difficult without it, while others think that computers have created a kind of dependency. The latter group think that this new dependency, like others, makes people's lives more difficult.

Social justice

Social justice is another issue we often hear about. The advocates of this issue argue that using computers at home is a luxury that not all people can afford. The cost of a computer, peripheral devices, and a monthly charge for Internet access is an extra burden on low-income people. This means that computers have created a service only for middle or high-income people, while low-income people are deprived of it. The opponents of this idea argue that the same issue was applied when other modern communication services were created, such as telephone and television. This group argue that, with the advance of technology, one day everyone will be able to afford a computer and access to the Internet.

Digital divide

The concept of a **digital divide** covers both the issues of dependency and social justice discussed above. The concept divides society into two groups: those who are electronically connected to the rest of society and those who are not. Someone in the first group communicates with people in the group through e-mail and uses the Internet for business and entertainment. Someone in the second group uses "snail mail" and the telephone for communication, and enjoys sitting in a cinema to see a movie instead of downloading it from a cable network. This digital divide is gradually disappearing in industrial countries, but experts believe that it will stay for a long time in developing countries.

1.9 KEY TERMS

This chapter has introduced the following key terms, which are listed here with the pages on which they first occur:

algorithm 8
computer languages 8
data processor 2
input data 2
integrated circuit 10
operating system 8
program 2
Turing model 1
arithmetic logic unit (ALU) 5
control unit 5
digital divide 11
instruction 5
memory 5
output data 2
software engineering 8
von Neumann model 5

1.10 SUMMARY

- The idea of a universal computational device was first given by Alan Turing in 1937. He proposed that all computation can be performed by a special kind of a machine, now called a Turing machine.
- The von Neumann model defines a computer as four subsystems: memory, arithmetic logic unit, control unit, and input/output. The von Neumann model states that the program must be stored in memory.
- We can think of a computer as made up of three components: computer hardware, data, and computer software.
- The history of computing and computers can be divided into three periods: the period of mechanical machines (before 1930), the period of electronic computers (1930–1950), and the period that includes the five modern computer generations.
- Computer science has created some peripheral issues, the most prevalent of which can be categorized as social and ethical issues.
- With the invention of computers a new discipline has evolved, *computer science*, which is now divided into several areas.

1.11 PRACTICE SET

Review questions

1. Define a computer based on the Turing model.
2. Define a computer based on the von Neumann model.
3. What is the role of a program in a computer that is based on the Turing model?
4. What is the role of a program in a computer based on the von Neumann model?
5. What are the various subsystems of a computer?
6. What is the function of the memory subsystem in a computer?
7. What is the function of the ALU subsystem in a computer?
8. What is the function of the control unit subsystem in a computer?
9. What is the function of the input/output subsystem in a computer?
10. Briefly describe the five generations of computers.

Multiple-choice questions

11. The _____ model is the basis for today's computers.
    a. Leibnitz
    b. von Neumann
    c. Pascal
    d. Charles Babbage
12. In a computer, the _____ subsystem stores data and programs.
    a. ALU
    b. input/output
    c. memory
    d. control unit
13. In a computer, the _____ subsystem performs calculations and logical operations.
    a. ALU
    b. input/output
    c. memory
    d. control unit
14. In a computer, the _____ subsystem accepts data and programs and sends processing results to output devices.
    a. ALU
    b. input/output
    c. memory
    d. control unit
15. In a computer, the _____ subsystem serves as a manager of the other subsystems.
    a. ALU
    b. input/output
    c. memory
    d. control unit
16. According to the von Neumann model, _____ stored in memory.
    a. only data is
    b. only programs are
    c. data and programs are
    d. (none of the above)
17. A step-by-step solution to a problem is called _____.
    a. hardware
    b. an operating system
    c. a computer language
    d. an algorithm
18. FORTRAN and COBOL are examples of _____.
    a. hardware
    b. operating systems
    c. computer languages
    d. algorithms
19. A 17th-century computing machine that could perform addition and subtraction was the _____.
    a. Pascaline
    b. Jacquard loom
    c. Analytical Engine
    d. Babbage machine
20. _____ is a set of instructions in a computer language that tells the computer what to do with data.
    a. An operating system
    b. An algorithm
    c. A data processor
    d. A program
21. _____ is the design and writing of a program in structured form.
    a. Software engineering
    b. Hardware engineering
    c. Algorithm development
    d. Instructional architecture
22. The first electronic special-purpose computer was called _____.
    a. Pascal
    b. Pascaline
    c. ABC
    d. ENIAC
23. One of the first computers based on the von Neumann model was called _____.
    a. Pascal
    b. Pascaline
    c. ABC
    d. EDVAC
24. The first computing machine to use the idea of storage and programming was called _____.
    a. the Madeline
    b. EDVAC
    c. the Babbage machine
    d. the Jacquard loom

25. _____ separated the programming task from computer operation tasks.
    a. Algorithms
    b. Data processors
    c. High-level programming languages
    d. Operating systems

Exercises

26. Explain why a computer cannot solve a problem for which there is no solution outside the computer.
27. If a small cheap computer can do the same thing as a large expensive computer, why do people need to have a large one.
28. Do some research and find out whether the Pascaline calculator is a computer according to the Turing model.
29. Do some research and find out whether Leibnitz's Wheel is a computer according to the Turing model.
30. Do some research and find out whether the Jacquard Loom is a computer according to the Turing model.
31. Do some research and find out whether Babbage's Analytical Engine is a computer according to the von Neumann model.
32. Do some research about the ABC computer and find out whether this computer followed the von Neumann model.
33. Do some research and find out in which computer generation keyboards originated.
34. Do some research and find out in which computer generation printers originated.
35. According to the von Neumann model, can the hard disk of today be used as input or output? Explain.
36. A programming language has ten different instructions. How many five-instruction programs can be written in this language if no instruction is repeated? How many seven-instruction programs?
37. Which is more valuable today to an organization: hardware, software, or data?

2.1 INTRODUCTION

A **number system** (or numeral system) defines how a number can be represented using distinct symbols. A number can be represented differently in different systems. For example, the two numbers $(2A)_{16}$ and $(52)_8$ both refer to the same quantity, $(42)_{10}$, but their representations are different. This is the same as using the words *cheval* (French) and *equus* (Latin) to refer to the same entity, a horse.

As we use symbols (characters) to create words in a language, we use symbols (digits) to represent numbers. However, we know that the number of symbols (characters) in any language is limited. We need to repeat characters and combine them to create words. It is the same for numbers: we have a limited number of symbols (digits) to represent numbers, which means that the digits need to be repeated.

Several number systems have been used in the past and can be categorized into two groups: positional and non-positional systems. Our main goal is to discuss the positional number systems, but we also give examples of non-positional systems.

2.2 POSITIONAL NUMBER SYSTEMS

In a **positional number system**, the position a symbol occupies in the number determines the value it represents. In this system, a number represented as:

$$\pm (S_{k-1} \ldots S_2 S_1 S_0 . S_{-1} S_{-2} \ldots S_{-l})_b$$

has the value of:

$$n = \pm \quad S_{k-1} \times b^{k-1} + \ldots + S_1 \times b^1 + S_0 \times b^0 \quad + \quad S_{-1} \times b^{-1} + S_{-2} \times b^{-2} + \ldots + S_{-l} \times b^{-l}$$

in which S is the set of symbols, b is the **base** (or **radix**), which is equal to the total number of the symbols in the set S, and S_i is the symbol in position i. Note that we have used an expression that can be extended from the right or from the left. In other words, the power of b can be 0 to $k - 1$ in one direction and -1 to $-l$ in the other direction. The terms with non-negative powers of b are related to the integral part of the number, while the terms with negative power of b are related to the fractional part of the number. The ± sign shows that the number can be either positive or negative. We will study several positional number systems in this chapter.

The decimal system (base 10)

The first positional number system we discuss in this chapter is the **decimal system**. The word *decimal* is derived from the Latin root *decem* (ten). In this system the base b = 10 and we use ten symbols to represent a number. The set of symbols is S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. As we know, the symbols in this system are often referred to as **decimal digits** or just digits. In this chapter, we use ± to show that a

number can be positive or negative, but remember that these signs are not stored in computers—computers handle the sign differently, as we discuss in Chapter 3.

Computers store positive and negative numbers differently.

In the decimal system, a number is written as:

$$\pm (S_{k-1} \ldots S_2 S_1 S_0 . S_{-1} S_{-2} \ldots S_{-l})_{10}$$

but for simplicity, we often drop the parentheses, the base, and the plus sign (if the number is positive). For example, we write the number $+(552.23)_{10}$ as 552.23—the base and plus signs are implicit.

Integers

An **integer** (an integral number with no fractional part) in the decimal system is familiar to all of us—we use integers in our daily life. In fact, we have used them so much that they are intuitive. We represent an integer as $\pm\ S_{k-1} \ldots S_1\ S_0$. The value is calculated as:

$$N = \pm \quad S_{k-1} \times 10^{k-1} + S_{k-2} \times 10^{k-2} + \ldots + S_2 \times 10^2 + S_1 \times 10^1 + S_0 \times 10^0$$

in which S_i is a digit, b = 10 is the base, and k is the number of digits.

Another way to show an integer in a number system is to use **place values**, which are powers of 10 (10^0, 10^1, ... 10^{k-1}) for decimal numbers. Figure 2.1 shows an integer in the decimal system using place values.

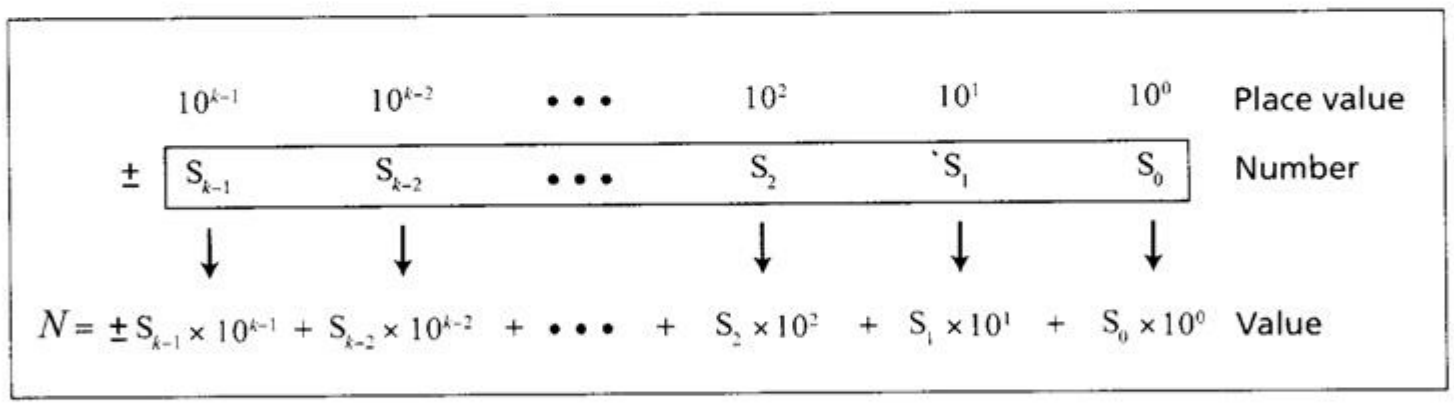


Figure 2.1 Place values for an integer in the decimal system

Example 2.1

The following shows the place values for the integer +224 in the decimal system.

		10^2		10^1		10^0	Place value
		2		2		4	Number
N	= +	2×10^2	+	2×10^1	+	4×10^0	Value

Note that the digit 2 in position 1 has the value 20, but the same digit in position 2 has the value 200. Also note that we normally drop the plus sign, but it is implicit.

Example 2.2

The following shows the place values for the decimal number −7508. We have used 1, 10, 100, and 1000 instead of powers of 10.

		1000		100		10		1	Place value
		7		5		0		8	Number
N =	−	(7 × 1000	+	5 × 100	+	0 × 10	+	8 × 1)	Value

Maximum value

Sometimes we need to know the maximum value of a decimal integer that can be represented by k digits. The answer is $N_{max} = 10^k - 1$. For example, if $k = 5$, then the maximum value is $N_{max} = 10^5 - 1 = 99{,}999$.

Reals

A **real** (a number with a fractional part) in the decimal system is also familiar. For example, we use this system to show dollars and cents ($23.40). We can represent a real as $\pm\ S^{k-1} \ldots S_1\ S_0 \cdot S_{-1} \ldots\ S_{-l}$. The value is calculated as:

$$R = \pm \quad \underbrace{S_{k-1} \times 10^{k-1} + \ldots + S_1 \times 10^1 + S_0 \times 10^0}_{\text{Integral part}} \quad + \quad \underbrace{S_{-1} \times 10^{-1} + \ldots + S_{-l} \times 10^{-l}}_{\text{Fractional part}}$$

in which S_i is a digit, b = 10 is the base, k is the number of digits in the integral part, and l is the number of digits in the fractional part. The decimal point we use in our representation separates the fractional part from the integral part.

Example 2.3

The following shows the place values for the real number +24.13.

		10^1		10^0		10^{-1}		10^{-2}	Place value
		2		4	•	1		3	Number
R =	+	(2 × 10	+	4 × 1	+	1 × 0.1	+	3 × 0.01)	Value

The binary system (base 2)

The second positional number system we discuss in this chapter is the **binary system**. The word *binary* is derived from the Latin root *bini* (or two by two). In this system the base b = 2 and we use only two symbols, S = {0, 1}. The symbols in this system are often referred to as **binary digits** or **bits** (binary digit). As we will see in Chapter 3, data and programs are stored in the computer using binary patterns, a string of bits. This is because the computer is made of electronic switches that can have only two states, on and off. The bit 1 represents one of these two states and the bit 0 the other.

We can represent an integer as $\pm\ (S_{k-1} \ldots S_1\ S_0)_2$. The value is calculated as:

$$N = \pm\ S_{k-1} \times 2^{k-1} + S_{k-2} \times 2^{k-2} + \ldots + S_2 \times 2^2 + S_1 \times 2^1 + S_0 \times 2^0$$

in which S_i is a digit, b = 2 is the base, and k is the number of bits. Another way to show a binary number is to use place values (2^0, 2^1, ... 2^{k-1}). Figure 2.2 shows a number in the binary number system using place values:

	2^{k-1}	2^{k-2}	• • •	2^2	2^1	2^0	Place value
$\pm$	S_{k-1}	S_{k-2}	• • •	S_2	S_1	S_0	Number
	↓	↓		↓	↓	↓	
$N = \pm$	$S_{k-1} \times 2^{k-1}$	$+\ S_{k-2} \times 2^{k-2}$	$+$ • • •	$+\ S_2 \times 2^2$	$+\ S_1 \times 2^1$	$+\ S_0 \times 2^0$	Value

Figure 2.2 Place values for an integer in the binary system

Example 2.4

The following shows that the number $(11001)_2$ in binary is the same as 25 in decimal. The subscript 2 shows that the base is 2.

	2^4	2^3	2^2	2^1	2^0	Place value
	1	1	0	0	1	Number
$N =$	1×2^4	$+\ 1 \times 2^3$	$+\ 0 \times 2^2$	$+\ 0 \times 2^1$	$+\ 1 \times 2^0$	Decimal

Note that the equivalent decimal number is $N = 16 + 8 + 0 + 0 + 1 = 25$.

Maximum value

The maximum value of a binary integer with k digits is $N_{max} = 2^k - 1$. For example, if $k = 5$, then the maximum value is $N_{max} = 2^5 - 1 = 31$.

A real—a number with an optional fractional part—in the binary system can be made of k bits on the left and l bits on the right, $\pm\ (S^{k-1} \ldots S_1\ S_0 \cdot S_{-1} \ldots\ S_{-l})_2$. The value can be calculated as:

	Integral part		Fractional part
$R = \pm$	$S_{k-1} \times 2^{k-1} + \ldots + S_1 \times 2^1 + S_0 \times 2^0$	$+$	$S_{-1} \times 2^{-1} + \ldots + S_{-l} \times 2^{-l}$

in which S_i is a digit, b = 2 is the base, k is the number of bits to the left, and l is the number of bits to the right of the decimal point. Note that k starts from 0, but l starts from −1. The highest power is $k - 1$ and the lowest power is $-l$.

Example 2.5

The following shows that the number $(101.11)_2$ in binary is equal to the number 5.75 in decimal.

		2^2		2^1		2^0		2^{-1}		2^{-2}	Place value
		1		0		1	•	1		1	Number
R	=	1×2^2	+	0×2^1	+	1×2^0	+	1×2^{-1}	+	1×2^{-2}	Value

Note that the value in the decimal system is $R = 4 + 0 + 1 + 0.5 + 0.25 = 5.75$.

The hexadecimal system (base 16)

Although the binary system is used to store data in computers, it is not convenient for representation of numbers outside the computer, as a number in binary notation is much longer than the corresponding number in decimal notation. However, the decimal system does not show what is stored in computer as binary directly—there is no obvious relationship between the number of bits in binary and the number of decimal digits. Conversion from one to the other is not fast, as we will see shortly.

To overcome this problem, two positional systems were devised: hexadecimal and octal. We first discuss the **hexadecimal system**, which is more common. The word *hexadecimal* is derived from the Greek root *hex* (six) and the Latin root *decem* (ten). To be consistent with decimal and binary, it should really have been called *sexadecimal*, from the Latin roots *sex* and *decem*. In this system the base b = 16 and we use sixteen symbols to represent a number. The set of symbols is S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. Note that the symbols A, B, C, D, E, F (uppercase or lowercase) are equivalent to 10, 11, 12, 13, 14, and 15 respectively. The symbols in this system are often referred to as **hexadecimal digits**.

Integers

We can represent an integer as $\pm\ S_{k-1} \ldots S_1\ S_0$. The value is calculated as:

$$N = \pm S_{k-1} \times 16^{k-1} + S_{k-2} \times 16^{k-2} + \ldots + S_2 \times 16^2 + S_1 \times 16^1 + S_0 \times 16^0$$

in which S_i is a digit, b = 16 is the base, and k is the number of digits.

Another way to show a hexadecimal number is to use place values (16^0, 16^1, ... 16^{k-1}). Figure 2.3 shows a number in the hexadecimal number system using place values.

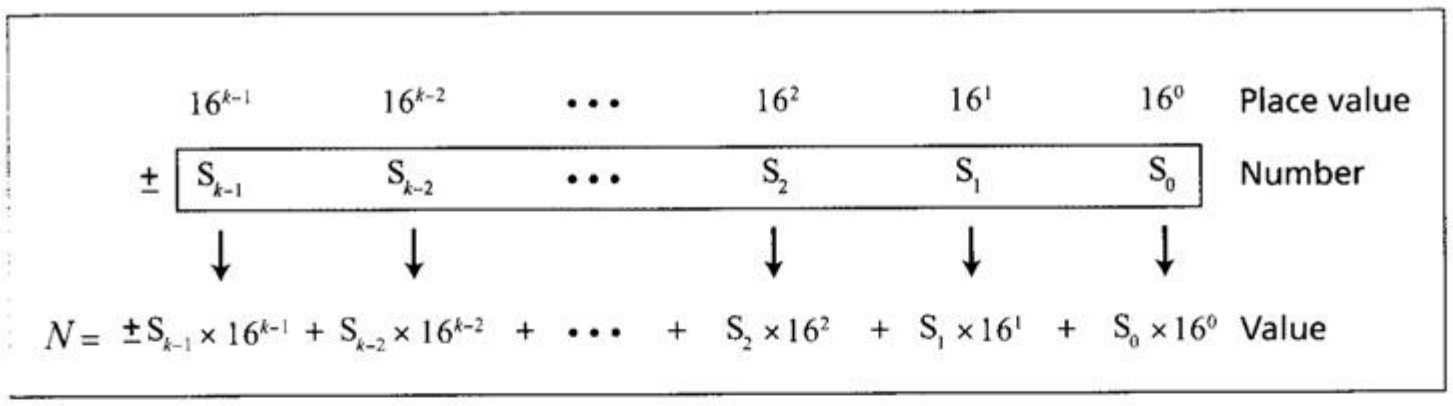


Figure 2.3 Place values for an integer in the hexadecimal system

Example 2.6

The following shows that the number $(2AE)_{16}$ in hexadecimal is equivalent to 686 in decimal.

	16^2		16^1		16^0	Place value
	2		A		E	Number
N =	2×16^2	+	10×16^1	+	14×16^0	Value

Maximum value

The maximum value of a hexadecimal integer with k digits is $N_{max} = 16^k - 1$. For example, if $k = 5$, then the maximum value is $N_{max} = 16^5 - 1 = 1{,}048{,}575$.

Although a real number can be also represented in the hexadecimal system, it is not very common. We leave this as an exercise.

The octal system (base 8)

The second system that was devised to show the equivalent of the binary system outside the computer is the **octal system**. The word *octal* is derived from the Latin root *octo* (eight). In this system the base b = 8 and we use eight symbols to represent a number. The set of symbols is S = {0, 1, 2, 3, 4, 5, 6, 7}. The symbols in this system are often referred to as **octal digits**.

We can represent an integer as $\pm\ S_{k-1} \ldots S_1\ S_0$. The value is calculated as:

$$N = \pm S_{k-1} \times 8^{k-1} + S_{k-2} \times 8^{k-2} + \ldots + S_2 \times 8^2 + S_1 \times 8^1 + S_0 \times 8^0$$

in which S_i is a digit, b = 8 is the base, and k is the number of digits.

Another way to show an octal number is to use place values (8^0, 8^1, ... 8^{k-1}). Figure 2.4 shows a number in the octal number system using place values.

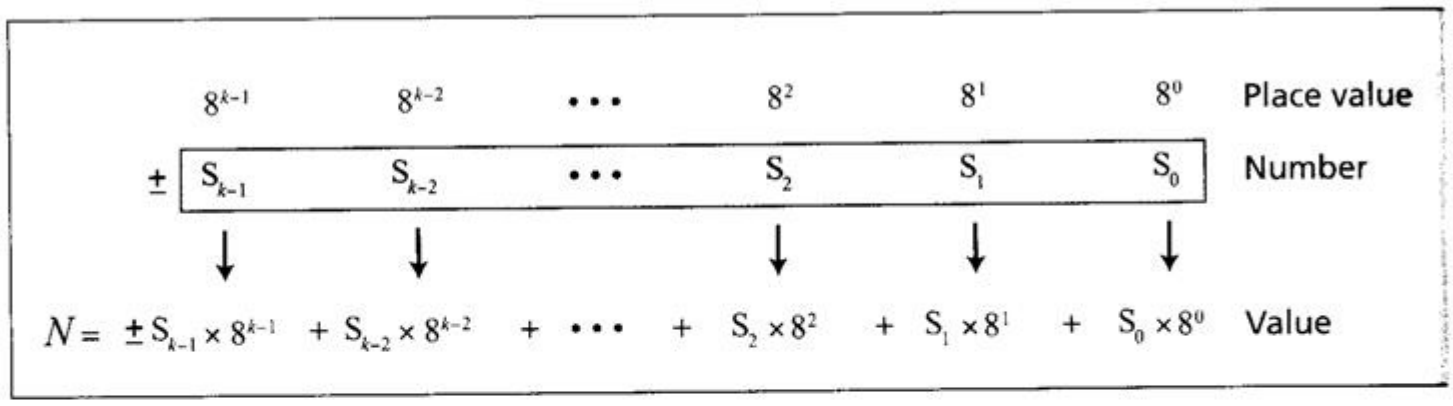


Figure 2.4 Place values for an integer in the octal system

Example 2.7

The following shows that the number $(1256)_8$ in octal is the same as 686 in decimal.

	8^3		8^2		8^1		8^0	Place value
	1		2		5		6	Number
N =	1×8^3	+	2×8^2	+	5×8^1	+	6×8^0	Value

Note that the decimal number is $N = 512 + 128 + 40 + 6 = 686$.

Maximum value

The maximum value of an octal integer with k digits is $N_{max} = 8^k - 1$. For example, if $k = 5$, then the maximum value is $N_{max} = 8^5 - 1 = 32767$.

Reals

Although a real number can be also represented in the octal system, it is not very common. We leave this as an exercise.

Summary of the four positional systems

Table 2.1 shows a summary of the four positional number systems discussed in this chapter.

Table 2.1 Summary of the four positional number systems

System	*Base*	*Symbols*	*Examples*
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	2345.56
Binary	2	0, 1	$(1001.11)_2$
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	$(156.23)_8$
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	$(A2C.A1)_{16}$

Table 2.2 shows how the number 15 is represented with two digits in decimal, four digits in binary, two digits in octal, and only one digit in hexadecimal. The hexadecimal representation is definitely the shortest.

Table 2.2 Comparison of numbers in the four systems

Decimal	*Binary*	*Octal*	*Hexadecimal*
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Conversion

We need to know how to convert a number in one system to the equivalent number in another system. Since the decimal system is more familiar than the other systems, we first show how to covert from any base to decimal. Then we show how to convert from decimal to any base. Finally, we show how we can easily convert from binary to hexadecimal or octal and vice versa.

This type of conversion is easy and fast. We multiply each digit with its place value in the source system and add the results to get the number in the decimal system. Figure 2.5 shows the idea.

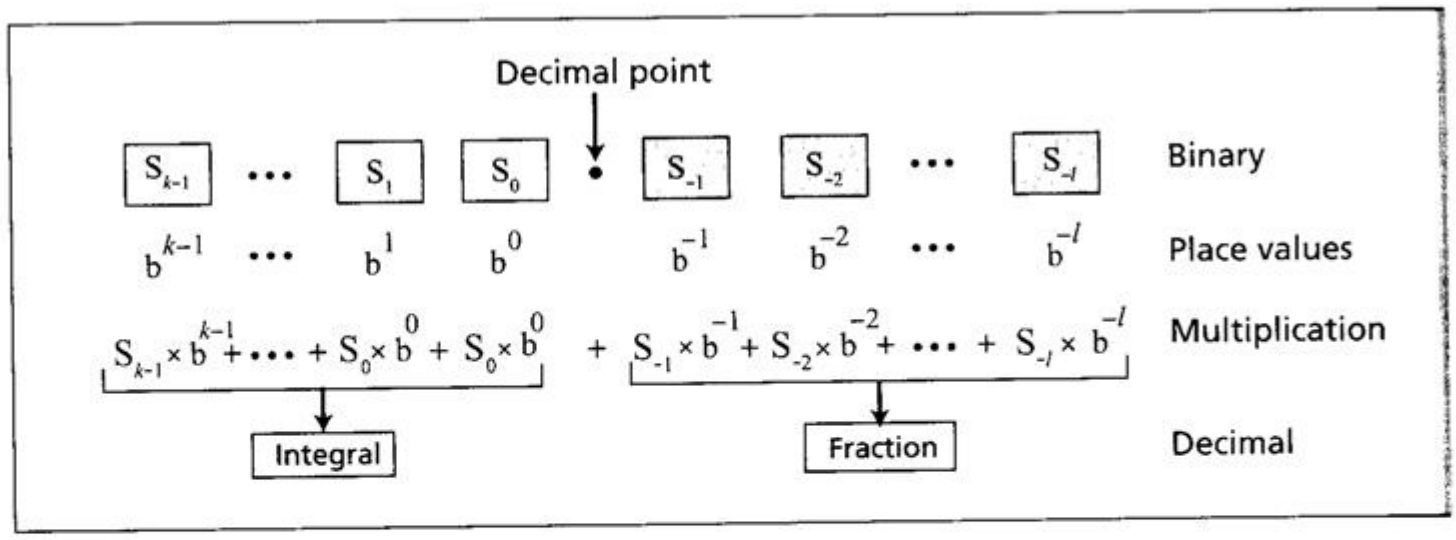


Figure 2.5 Converting other bases to decimal

Example 2.8

The following shows how to convert the binary number $(110.11)_2$ to decimal: $(110.11)_2 = 6.75$.

Binary	1		1		0	•	1		1
Place value	2^2		2^1		2^0		2^{-1}		2^{-2}
Partial result	4	+	2	+	0	+	0.5	+	0.25

Decimal: 6.75

Example 2.9

The following shows how to convert the hexadecimal number $(1A.23)_{16}$ to decimal.

Hexadecimal	1		A	•	2		3
Place value	16^1		16^0		16^{-1}		16^{-2}
Partial result	16	+	10	+	0.125	+	0.012

Decimal: 26.137

Note that the result in the decimal notation is not exact, because $3 \times 16^{-2} = 0.01171875$. We have rounded this value to three digits (0.012). In other words, $(1A.23)_{16} \approx 26.137$. When we convert a number in decimal to hexadecimal, we need to specify how many digits we allow to the right of the decimal point.

Example 2.10

The following shows how to convert $(23.17)_8$ to decimal.

Octal	2		3	•	1		7
Place value	8^1		8^0		8^{-1}		8^{-2}
Partial result	16	+	3	+	0.125	+	0.109

Decimal: 19.234

This means that $(23.17)_8 \approx 19.234$ in decimal. Again, we have rounded up $7 \times 8^{-2} = 0.109375$.

ecimal any base

We can convert a decimal number to its equivalent in any base. We need two procedures, one for the integral part and one for the fractional part.

Converting the integral part

The integral part can be converted using repetitive division. Figure 2.6 shows the UML diagram for the process. We use UML diagrams through the book. For those readers not familiar with UML diagrams, please see Appendix B.

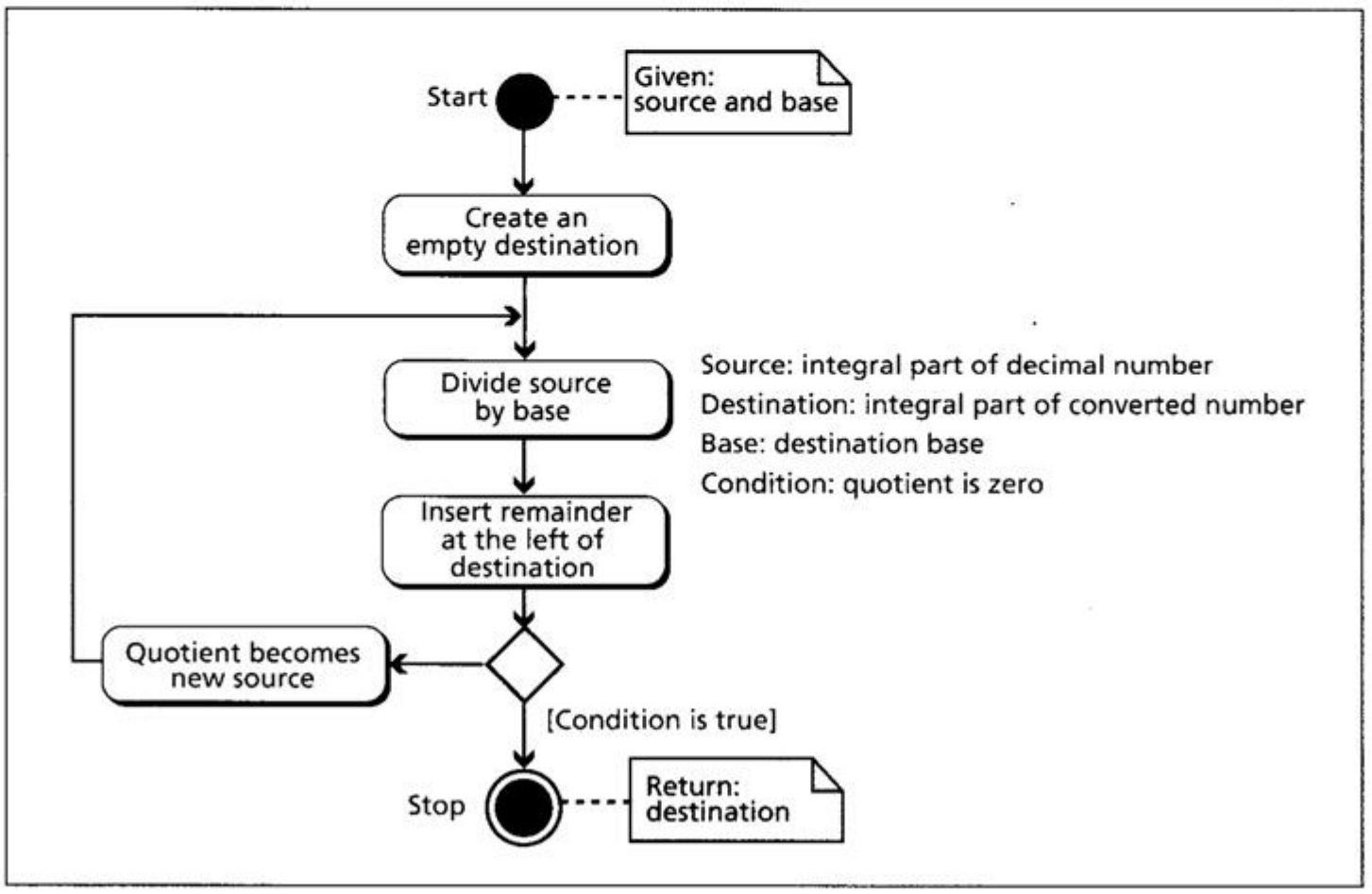


Figure 2.6 Converting the integral part of a number in decimal to other bases

We call the integral part of the decimal number the *source* and the integral part of the converted number the *destination*. We first create an empty destination. We then repeatedly divide the source to get the quotient and the remainder. The

remainder is inserted to the left of the destination. The quotient becomes a new source. Figure 2.7 shows the how the destination is made with each repetition.

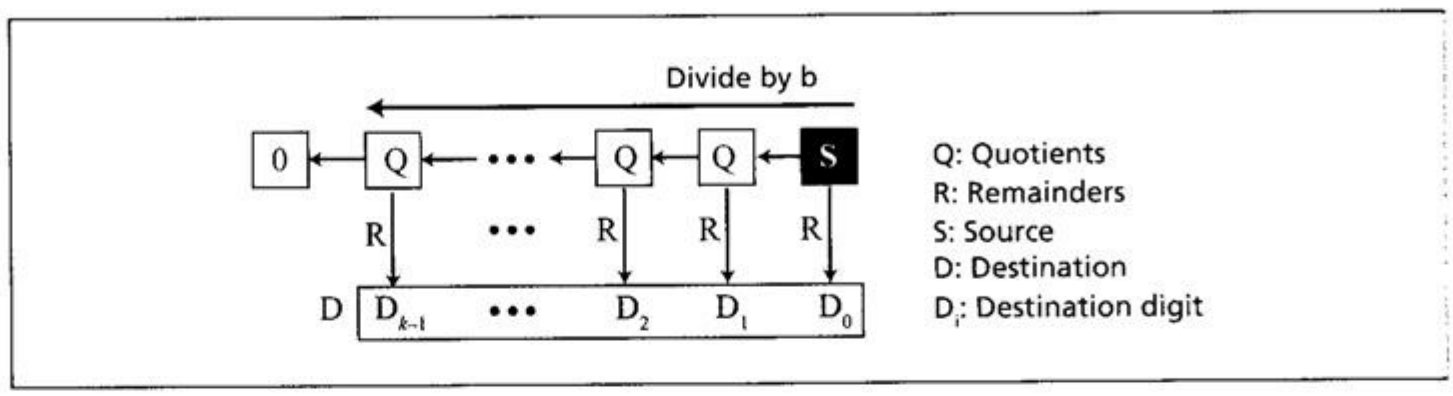
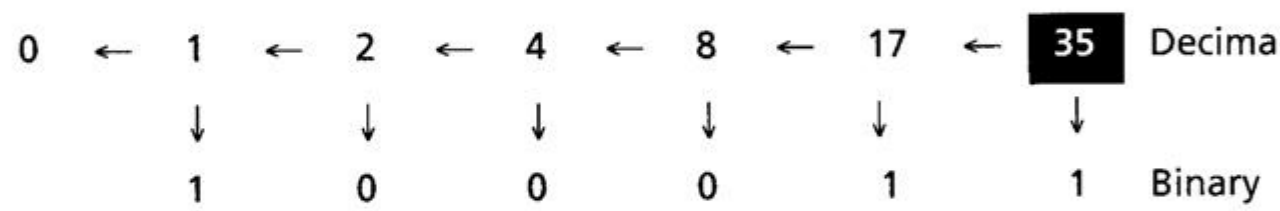


Figure 2.7 Converting the integral part of a number in decimal to other bases

We use Figure 2.7 below to illustrate the process manually with some examples.

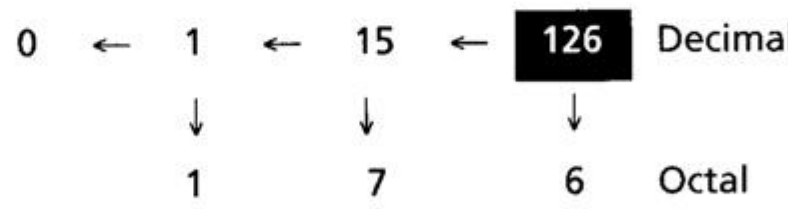
Example 2.11

The following shows how to convert 35 in decimal to binary. We start with the number in decimal, we move to the left while continuously finding the quotients and the remainder of division by 2. The result is $35 = (100011)_2$.



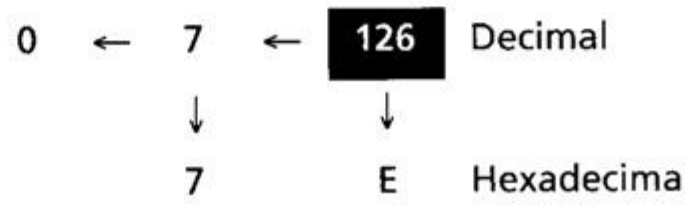
Example 2.12

The following shows how to convert 126 in decimal to its equivalent in the octal system. We move to the right while continuously finding the quotients and the remainder of division by 8. The result is $126 = (176)_8$.



Example 2.13

The following shows how we convert 126 in decimal to its equivalent in the hexadecimal system. We move to the right while continuously finding the quotients and the remainder of division by 16. The result is $126 = (7E)_{16}$.



Converting the fractional part

The fractional part can be converted using repetitive multiplication. We call the fractional part of the decimal number the *source* and the fractional part of the converted number the *destination*. We first create an empty destination. We then repeatedly multiply the source to get the result. The integral part of the result is inserted to the right of the destination, while the fractional part becomes the new source. Figure 2.8 shows the UML diagram for the process.

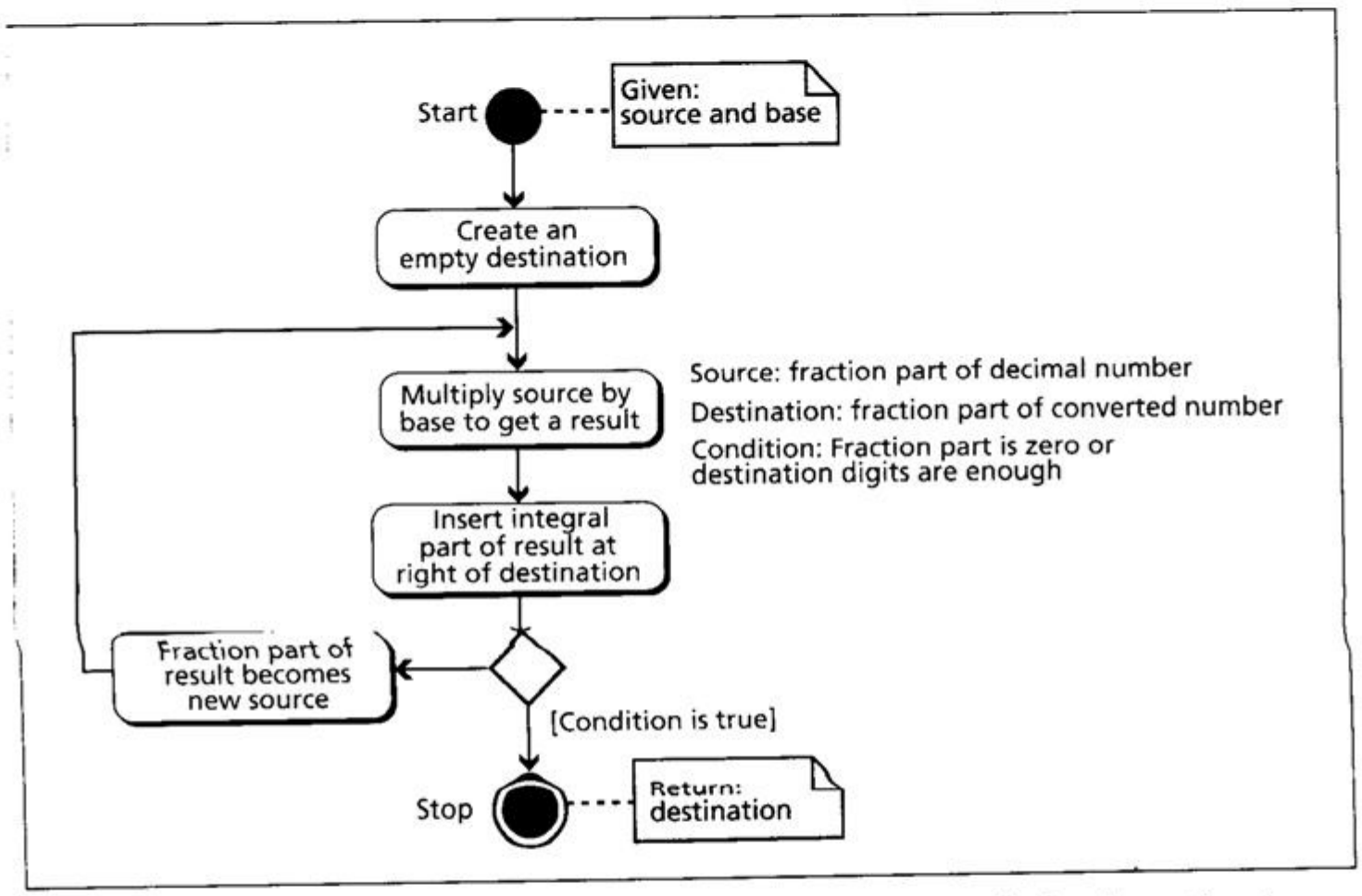


Figure 2.8 Converting the fractional part of a number in decimal to other bases

Figure 2.9 shows how the destination is made in each repetition. We use the figure to illustrate the process manually with some examples.

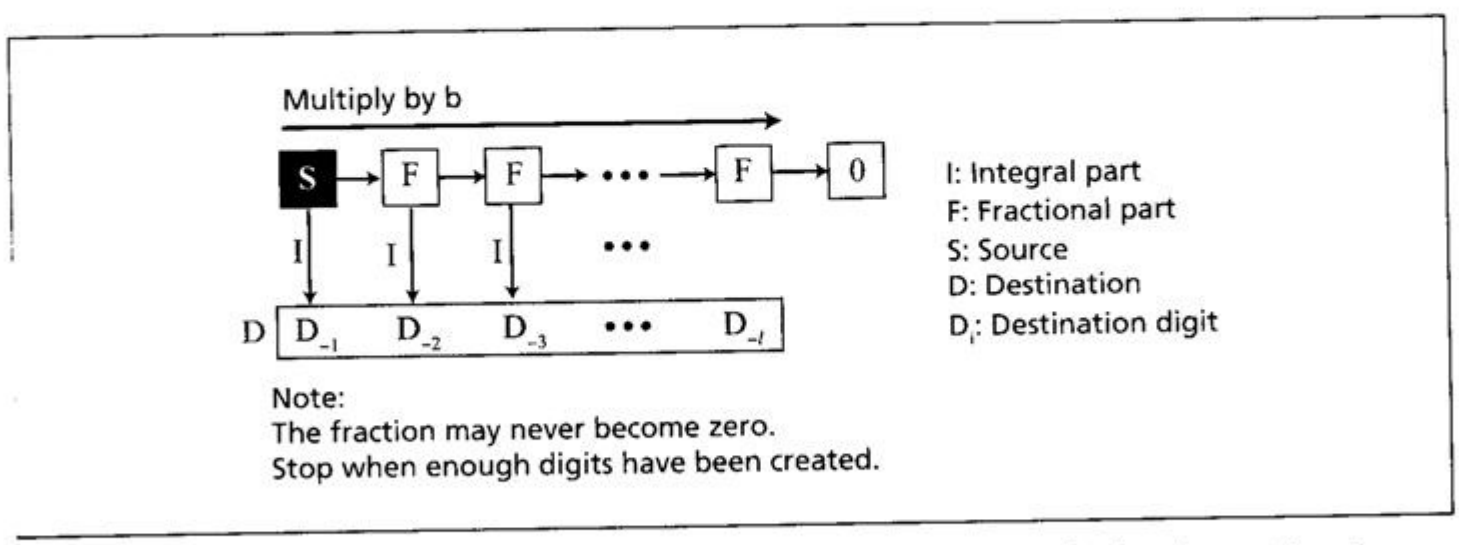


Figure 2.9 Converting the fractional part of a number in decimal to other bases

Example 2.14

Convert the decimal number 0.625 to binary.

Solution

Since the number 0.625 has no integral part, the example shows how the fractional part is calculated. The base here is 2. Write the decimal number at the left corner. Multiply the number continuously by 2 and record the integral and fractional part of the result. The fractional part moves to the right, and the integral part is recorded under each operation. Stop when the fractional part is 0 or there are enough bits. The result is $(0.101)_2$.

Decimal		0.625	→	0.25	→	0.50	→	0.00
		↓		↓		↓		
Binary	•	1		0		1		

Example 2.15

The following shows how to convert 0.634 to octal using a maximum of four digits. The result is $0.634 = (0.5044)_8$. Note that we multiply by 8 (base octal).

Decimal		0.634	→	0.072	→	0.576	→	0.608	→	0.864
		↓		↓		↓		↓	
Octal	•	5		0		4		4	

Example 2.16

The following shows how to convert 178.6 in decimal to hexadecimal using only one digit to the right of the decimal point. The result is $178.6 = (B2.9)_{16}$. Note that we divide or multiply by 16 (base hexadecimal).

Decimal	0	←	11	←	178	0.6	→	0.6
			↓		↓	↓		
Hexadecimal			B		2	• 9		

Example 2.17

An alternative method for converting a small decimal integer (usually less than 256) to binary is to break the number as the sum of numbers that are equivalent to the binary place values shown:

Place value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal equivalent	128	64	32	16	8	4	2	1

Using this table, we can convert 165 to binary $(10100101)_2$ as shown below:

Decimal 165 =	128	+	0	+	32	+	0	+	0	+	4	+	0	+	1
Binary	1		0		1		0		0		1		0		1

Example 2.18

A similar method can be used to convert a decimal fraction to binary when the denominator is a power of two:

Place value	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
Decimal equivalent	$^1/_2$	$^1/_4$	$^1/_8$	$^1/_{16}$	$^1/_{32}$	$^1/_{64}$	$^1/_{128}$

Using this table, we convert $^{27}/_{64}$ to binary $(0.011011)_2$ as shown below:

Decimal $^{27}/_{64}$ =	$^{16}/_{64}$	+	$^8/_{64}$	+	$^2/_{64}$	+	$^1/_{64}$
	$^1/_4$	+	$^1/_8$	+	$^1/_{32}$	+	$^1/_{64}$

Aligning these fractions according to decimal equivalent values:

Decimal $^{27}/_{64}$ =	0	+	$^1/_4$	+	$^1/_8$	+	0	+	$^1/_{32}$	+	$^1/_{64}$
Binary	0		1		1		0		1		1

Number of digits

We often need to know the number of digits before converting a number from decimal to other bases. In a positional number system with base b, we can always find the number of digits of an integer using the relation $k = \lceil \log_b N \rceil$, in which $\lceil x \rceil$ means the smallest integer greater than or equal to x (it is also called the *ceiling* of x), and N is the decimal value of the integer. For example, we can find the required number of bits in the decimal number 234 in all four systems as follows:

a. In decimal: $k_d = \lceil \log_{10} 234 \rceil = \lceil 2.37 \rceil = 3$, which is obvious.
b. In binary: $k_b = \lceil \log_2 234 \rceil = \lceil 7.8 \rceil = 8$. This is true because $234 = (11101010)_2$
c. In octal: $k_o = \lceil \log_8 234 \rceil = \lceil 2.62 \rceil = 3$. This is true because $234 = (352)_8$
d. In hexadecimal $k_h = \lceil \log_{16} 234 \rceil = \lceil 1.96 \rceil = 2$. This is true because $234 = (EA)_{16}$

See Appendix G for information on how to calculate $\log_b N$ if your calculator does not include logs to any base.

We can easily change a number from binary to hexadecimal and vice verse. The reason for this is that there is a relationship between the two bases: four bits in binary is one digit in hexadecimal. Figure 2.10 shows how this conversion can be done.

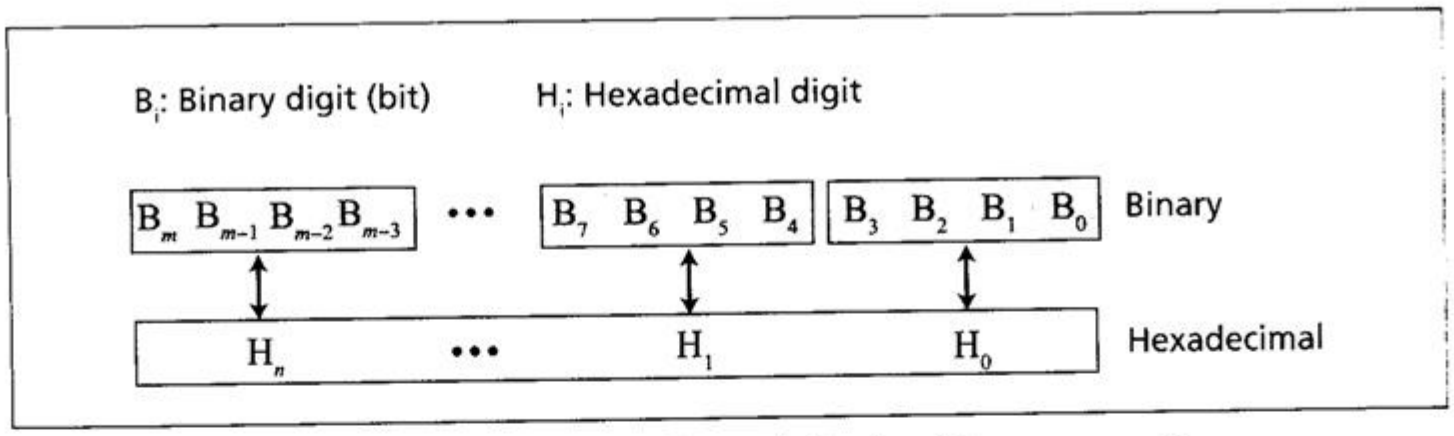


Figure 2.10 Binary to hexadecimal and hexadecimal to binary conversion

Example 2.19

Show the hexadecimal equivalent of the binary number $(10011100010)_2$.

Solution

We first arrange the binary number in 4-bit patterns: 100 1110 0010. Note that the leftmost pattern can have one to four bits. We then use the equivalent of each pattern shown in Table 2.2 on page 25 to change the number to hexadecimal: $(4E2)_{16}$.

Example 2.20

What is the binary equivalent of $(24C)_{16}$?

Solution

Each hexadecimal digit is converted to 4-bit patterns: 2 → 0010, 4 → 0100, and C → 1100. The result is $(001001001100)_2$.

Binary–octal conversion

We can easily convert a number from binary to octal and vice versa. The reason is that there is an interesting relationship between the two bases: three bits is one octal digit. Figure 2.11 shows how this conversion can be done.

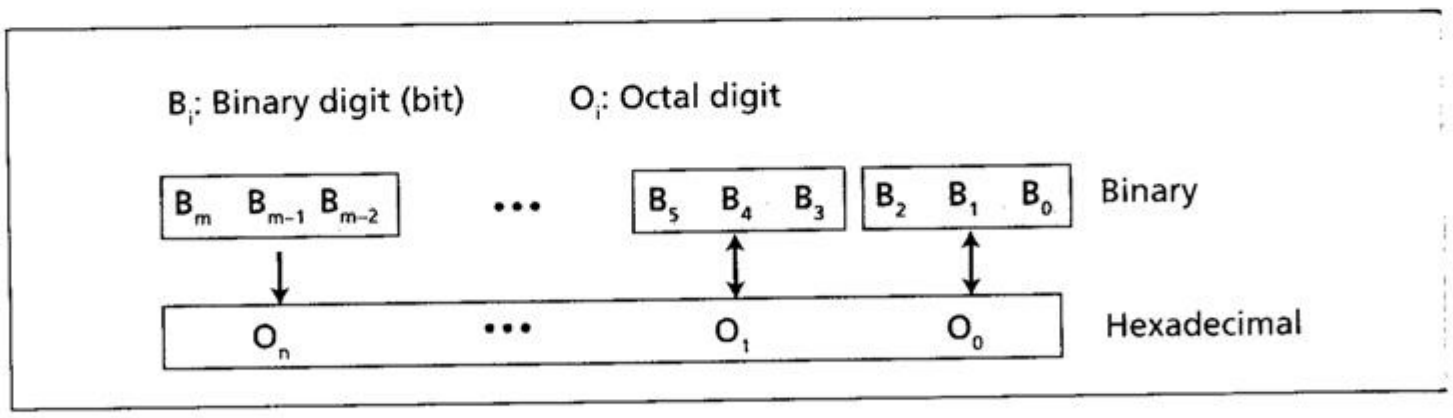


Figure 2.11 Binary to octal and octal to binary conversion

Example 2.21

Show the octal equivalent of the binary number $(101110010)_2$.

Solution

Each group of three bits is translated into one octal digit. The equivalent of each 3-bit group is shown in Table 2.2 on page 25. The result is $(562)_8$.

Example 2.22

What is the binary equivalent of $(24)_8$?

Solution

Write each octal digit as its equivalent bit pattern to get $(010100)_2$.

It is not difficult to convert a number in octal to hexadecimal or vice versa. We can use the binary system as the intermediate system. Figure 2.12 shows an example.

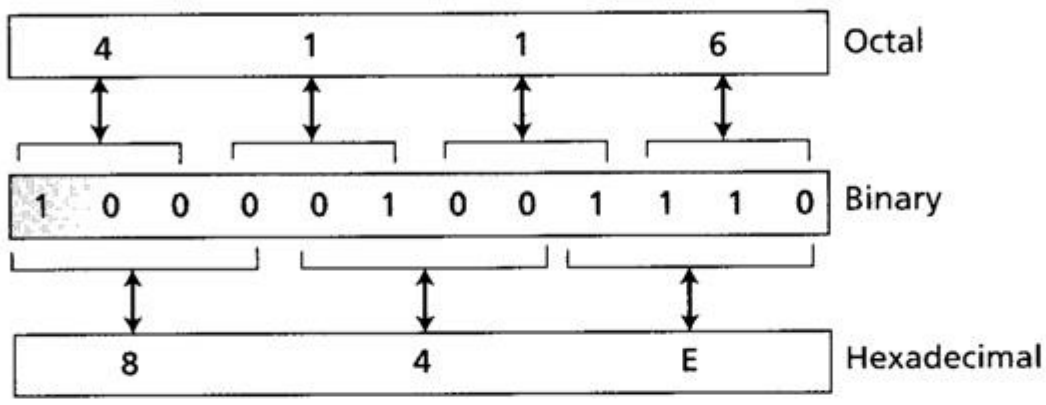


Figure 2.12 Octal to hexadecimal and hexadecimal to octal conversion

The following illustrates the process:

- To convert from octal to hexadecimal, we first convert the number in the octal system to binary. We then rearrange the bits in groups of four bits to find the hexadecimal equivalent.
- To convert from hexadecimal to octal, we first convert the number in the hexadecimal system to binary. We then rearrange the bits in groups of three to find the octal equivalent.

Number of digits

In conversion from one base to another, we often need to know the minimum number of digits we need in the destination system if we know the maximum number of digits in the source system. For example, if we know that we use at most six decimal digits in the source system, we want to know the minimum number of binary digits we need in the destination system. In general, assume that we are using k digits in base b_1 system. The maximum number we can represent in the source system is $b_1^k - 1$. The maximum number we can have in the destination system is $b_2^x - 1$. Therefore, $b_2^x - 1 \geq b_1^k - 1$. This means $b_2^x \geq b_1^k$, which means:

$$x \geq k \times (\log b_1 / \log b_2) \qquad \text{or} \qquad x = \lceil k \times (\log b_1 / \log b_2) \rceil$$

Example 2.23

Find the minimum number of binary digits required to store decimal integers with a maximum of six digits.

Solution

$k = 6$, $b_1 = 10$, and $b_2 = 2$. Then $x = \lceil k \times (\log b_1 / \log b_2) \rceil = \lceil 6 \times (1 / 0.30103) \rceil = 20$. The largest six-digit decimal number is 999,999 and the largest 20-bit binary number is 1,048,575. Note that the largest number that can be represented by a 19-bit number is 524287, which is smaller than 999,999. We definitely need twenty bits.

2.3 NONPOSITIONAL NUMBER SYSTEMS

Although nonpositional number systems are not used in computers, we give a short review here for comparison with positional number systems. A **nonpositional number system** still uses a limited number of symbols in which each symbol has a value. However, the position a symbol occupies in the number normally bears no relation to its value—the value of each symbol is fixed. To find the value of a number, we add the value of all symbols present in the representation. In this system, a number is represented as:

$$S_{k-1} \ldots S_2 S_1 S_0 \bullet S_{-1} S_{-2} \ldots S_{-l}$$

and has the value of:

$$n = \pm \underbrace{S_{k-1} + \ldots + S_1 + S_0}_{\textit{Integral part}} + \underbrace{S_{-1} + S_{-2} + \ldots + S_{-l}}_{\textit{Fractional part}}$$

There are some exceptions to the addition rule we just mentioned, as shown in Example 2.24.

Example 2.24

Roman numerals are a good example of a nonpositional number system. This system was invented by the Romans and was used until the sixteenth century in Europe. It is still used in sports events, clock dials and other applications. This number system has a set of symbols S = {I, V, X, L, C, D, M}. The values of each symbol is shown in Table 2.3

Table 2.3 Values of symbols in the Roman number system

Symbol	*I*	*V*	*X*	*L*	*C*	*D*	*M*
Value	1	5	10	50	100	500	1000

To find the value of a number, we need to add the value of symbols subject to specific rules:

1. When a symbol with a smaller value is placed after a symbol having an equal or larger value, the values are added.
2. When a symbol with a smaller value is placed before a symbol having a larger value, the smaller value is subtracted from the larger one.

3. A symbol S_1 cannot come before another symbol S_2 if $S_1 \leq 10 \times S_2$. For example, I or V cannot come before C.
4. For large numbers a bar is placed above any of the six symbols (all symbols except I) to express multiplication by 1000. For example, $\overline{V}$ = 5,000 and $\overline{M}$ = 1,000,000.
5. Although Romans used the word *nulla* (nothing) to convey the concept of zero, Roman numerals lack a zero digit in their system.

The following shows some Roman numbers and their values.

III	→	1 + 1 + 1	=	3
IV	→	5 – 1	=	4
VIII	→	5 + 1 + 1 + 1	=	8
XVIII	→	10 + 5 + 1 + 1 + 1	=	18
XIX	→	10 + (10 –1)	=	19
LXXII	→	50 + 10 + 10 + 1 + 1	=	72
CI	→	100 + 1	=	101
MMVII	→	1000 + 1000 + 5 + 1 + 1	=	2007
MDC	→	1000 + 500 + 100	=	1600

2.4 RECOMMENDED READING

For more details about the subjects discussed in this chapter, the following books are recommended:

Stalling, W: *Computer Organization and Architecture*, Upper Saddle River, NJ: Prentice Hall, 2000.

Mano M: *Computer System Architecture*, Upper Saddle River, NJ: Prentice Hall, 1993

Null L and Lobur J: *Computer Organization and Architecture*, Sudbury, MA: Jones and Bartlett, 2003

Brown S and Vranesic Z: *Fundamentals of Digital Logic with Verilog Design*, New York: McGraw-Hill, 2003

2.5 KEY TERMS

This chapter has introduced the following key terms, which are listed here with the pages on which they first occur:

base 18
binary system 20
decimal digit 18
hexadecimal digit 22
integer 19
number system 18
octal system 23
positional number system 18
real 20
binary digit 20
bit 20
decimal system 18
hexadecimal system 22
nonpositional number system 34
octal digit 23
place value 19
radix 18
Roman number system 34

2.6 SUMMARY

- A number system (or numeral system) is a system that uses distinct symbols to represent a number.
- In a positional number system, the position a symbol occupies in the number determines the value it represents. Each position has a place value associated with it.
- A nonpositional number system uses a limited number of symbols in which each symbol has a value. However, the position a symbol occupies in the number normally bears no relation to its value: the value of each symbol is normally fixed.
- In the decimal system, the base b = 10 and we use 10 symbols to represent numbers. The symbols in this system are often referred to as *decimal digits* or just *digits*.
- In the binary system, the base b = 2 and we use only two symbols to represent numbers. The symbols in this system are often referred to as *binary digits* or *bits*.
- In a hexadecimal system, the base b = 16 and we use sixteen symbols to represent numbers. The symbols in this system are often referred to as *hexadecimal digits*.
- In an octal system, the base b = 8 and we use eight symbols to represent numbers. The symbols in this system are often referred to as *octal digits*.
- We can convert a number in any system to decimal. We multiply each digit with its place value in the source system and add the result to get the number in the decimal system.
- We can convert a decimal number to its equivalent in any base using two different procedures, one for the integral part and one for the fractional part. The integral part needs repeated division and the fraction part needs repeated multiplication.
- Conversion from the binary system to the hexadecimal system and from the hexadecimal system to the binary system is very easy, because four bits in the binary system are represented as one digit in the hexadecimal system.
- Conversion from the binary system to the octal system and from the octal system to the binary system is very easy, because three bits in the binary system are represented as one digit in the octal system.

2.7 PRACTICE SET

Review questions

1. Define a number system.
2. Distinguish between positional and nonpositional number systems.
3. Define the base or radix in a positional number system. What is the relationship between a base and the number of symbols in a positional number system?
4. Explain the decimal system. Why is it called *decimal*? What is the base in this system?
5. Explain the binary system. Why is it called *binary*? What is the base in this system?
6. Explain the octal system. Why is it called *octal*? What is the base in this system?
7. Explain the hexadecimal system. Why is it called *hexadecimal*? What is the base in this system?
8. Why is it easy to convert from binary to hexadecimal and vice versa?
9. How many bits in the binary system are represented by one digit in the hexadecimal system?
10. How many bits in the binary system are represented by one digit in the octal system?

Multiple-choice questions

11. The base of the decimal number system is ____.
    a. 2
    b. 8
    c. 10
    d. 16
12. The base of the binary number system is ____.
    a. 2
    b. 8
    c. 10
    d. 16
13. The base of the octal number system is ____.
    a. 2
    b. 8
    c. 10
    d. 16
14. The base of the hexadecimal number system is ____.
    a. 2
    b. 8
    c. 10
    d. 16
15. When converting a decimal integer to base b, we repeatedly _____ by b.
    a. divide
    b. multiply
    c. neither a nor b
    d. both a and b
16. When converting a decimal fraction to base b, we repeatedly _____ by b.
    a. divide
    b. multiply
    c. neither a nor b
    d. both a and b
17. Which of the following representations is erroneous?
    a. $(10111)_2$
    b. $(349)_8$
    c. $(3AB)_{16}$
    d. 256
18. Which of the following representations is erroneous?
    a. $(10211)_2$
    b. $(342)_8$
    c. $(EEE)_{16}$
    d. 145
19. Which of the following representations is erroneous?
    a. $(111)_2$
    b. $(346)_8$
    c. $(EEG)_{16}$
    d. 221

20. Which of the following representations is erroneous?
    a. $(110)_2$
    b. $(141)_8$
    c. $(EF)_{16}$
    d. 22A
21. Which of the following is equivalent to 12 in decimal?
    a. $(1110)_2$
    b. $(C)_{16}$
    c. (15)8
    d. None of the above
22. Which of the following is equivalent to 24 in decimal?
    a. $(11000)_2$
    b. $(1A)_{16}$
    c. $(31)_8$
    d. None of the above

Exercises

23. Convert the following binary numbers to decimal without using a calculator, showing your work:
    a. $(01101)_2$
    b. $(1011000)_2$
    c. $(011110.01)_2$
    d. $(111111.111)_2$
24. Convert the following hexadecimal numbers to decimal without using a calculator, showing your work:
    a. $(AB2)_{16}$
    b. $(123)_{16}$
    c. $(ABB)_{16}$
    d. $(35E.E1)_{16}$
25. Convert the following octal numbers to decimal without using a calculator, showing your work:
    a. $(237)_8$
    b. $(2731)_8$
    c. $(617.7)_8$
    d. $(21.11)_8$
26. Convert the following decimal numbers to binary without using a calculator, showing your work:
    a. 1234
    b. 88
    c. 124.02
    d. 14.56
27. Convert the following decimal numbers to octal without using a calculator, showing your work:
    a. 1156
    b. 99
    c. 11.4
    d. 72.8
28. Convert the following decimal numbers to hexadecimal without using a calculator, showing your work:
    a. 567
    b. 1411
    c. 12.13
    d. 16.5
29. Convert the following octal numbers to hexadecimal without using a calculator, showing your work:
    a. $(514)_8$
    b. $(411)_8$
    c. $(13.7)_8$
    d. $(1256)_8$
30. Convert the following hexadecimal numbers to octal without using a calculator, showing your work:
    a. $(51A)_{16}$
    b. $(4E1)_{16}$
    c. $(BB.C)_{16}$
    d. $(ABC.D)_{16}$
31. Convert the following binary numbers to octal without using a calculator, showing your work:
    a. $(01101)_2$
    b. $(1011000)_2$
    c. $(011110.01)_2$
    d. $(111111.111)_2$
32. Convert the following binary numbers to hexadecimal without using a calculator, showing your work:
    a. $(01101)_2$
    b. $(1011000)_2$
    c. $(011110.01)_2$
    d. $(111111.111)_2$

33. Convert the following decimal numbers to binary using the alternative method discussed in Example 2.17, showing your work:
    a. 121
    b. 78
    c. 255
    d. 214
34. Change the following decimal numbers into binary using the alternative method discussed in Example 2.18, showing your work:
    a. $3\ ^{5}/_{8}$
    b. $12\ ^{3}/_{32}$
    c. $4\ ^{13}/_{64}$
    d. $12\ ^{5}/_{128}$
35. In a positional number system with base b, the largest integer number that can be represented using k digits is $b^k - 1$. Find the largest number in each of the following systems with *six* digits.
    a. Binary
    b. Decimal
    c. Hexadecimal
    d. Octal
36. Without converting, find the minimum number of digits needed in the destination system for each of the following cases:
    a. Five-digit decimal number converted to binary.
    b. Four-digit decimal converted to octal.
    c. Seven-digit decimal converted to hexadecimal.
37. Without converting, find the minimum number of digits needed in the destination system for each of the following cases:
    a. 5-bit binary number converted to decimal.
    b. Three-digit octal number converted to decimal.
    c. Three-digit hexadecimal converted to decimal.
38. The following table shows how to rewrite a fraction so the denominator is a power of two (1, 4, 8, 16, and so on).

Original	New	Original	New
0.5	$^{1}/_{2}$	0.25	$^{1}/_{4}$
0.125	$^{1}/_{8}$	0.0625	$^{1}/_{16}$
0.03125	$^{1}/_{32}$	0.015625	$^{1}/_{64}$

However, sometimes we need a combination of entries to find the appropriate fraction. For example, 0.625 is not in the table, but we know that 0.625 is 0.5 + 0.125. This means that 0.625 can be written as $^{1}/_{2} + {}^{1}/_{8}$, or $^{5}/_{8}$.

Change the following decimal fractions to a fraction with a power of 2.
    a. 0.1875
    b. 0.640625
    c. 0.40625
    d. 0.375
39. Using the results of the previous problem, change the following decimal numbers to binary numbers.
    a. 7.1875
    b. 12.640625
    c. 11.40625
    d. 0.375
40. Find the maximum value of an integer in each of the following cases:
    a. b = 10, k = 10
    b. b = 2, k = 12
    c. b = 8, k = 8
    d. b = 16, k = 7
41. Find the minimum number of required bits to store the following integers:
    a. less than 1000
    b. less than 100,000
    c. less than 64
    d. less than 256
42. A number less than b^k can be represented using k digits in base b. Show the number of digits needed in each of the following cases.
    a. Integers less than 2^{14} in binary
    b. Integers less than 10^8 in decimal
    c. Integers less than 8^{13} in octal
    d. Integers less than 16^4 in hexadecimal
43. A common base used on the Internet is b = 256. We need 256 symbols to represent a number in this system. Instead of creating this large number of symbols, the designers of this system have used decimal numbers to represent a symbol: 0 to 255. In other words, the set of symbols is S = {0, 1, 2, 3, ..., 255}. A number in this system is always in the format $S_1.S_2.S_3.S_4$ with four symbols and three dots that separate

the symbols. The system is used to define Internet addresses (see Chapter 6). An example of an address in this system is 10.200.14.72, which is equivalent to $10 \times 256^3 + 200 \times 256^2 + 14 \times 256^1 + 72 \times 256^0 = 180{,}883{,}016$ in decimal. This number system is called *dotted decimal notation*.

Find the decimal value of each of the following Internet addresses:

a. 17.234.34.14
b. 14.56.234.56
c. 110.14.56.78
d. 24.56.13.11

44. Internet addresses described in the previous problem are also represented as patterns of bits. In this case, 32 bits are used to represent an address, eight bits for each symbol in dotted decimal notation. For example, the address 10.200.14.72 can also be represented as 00001010 11001000 00001110 01001000.

Show the bit representation of the following Internet addresses:

a. 17.234.34.14
b. 14.56.234.56
c. 110.14.56.78
d. 24.56.13.11

45. Write the decimal equivalent of the following Roman numbers:

a. XV
b. XXVII
c. VLIII
d. MCLVII

46. Convert the following decimal numbers to Roman numbers:

a. 17
b. 38
c. 82
d. 999

47. Find which of the following Roman numerals are not valid:

a. MMIM
b. MIC
c. CVC
d. VX

48. Mayan civilization invented a positional vigesimal (base 20) numeral system, called the *Mayan numeral system*. They use base 20 probably because they used both their fingers and toes for counting. This system has 20 symbols that are constructed from three simpler symbols. The advanced feature of the system is that it has a symbol for zero, which is a shell. The other two symbols are a circle (or a pebble) for one and a horizontal bar (or a stick) for five. To represent a number greater than nineteen, numerals are written vertically. Search the Internet to answer the following: what are the decimal numbers 12, 123, 452, and 1256 in the Mayan numeral system?

49. Babylonian civilization is credited for developing the first positional numeral system, called the *Babylonian numeral system*. They inherited the Sumerian and Akkadian numeral system and developed it into positional sexagesimal system (base 60). This base is still used today for times and angles. For example, one hour is 60 minutes and one minute is 60 seconds: similarly, one degree is 60 minutes and one minute is 60 seconds. As a positional system with base b requires b symbols (digits), we expect a positional sexagesimal system to require 60 symbols. However, the Babylonians did not have a symbol for zero, and produced the other 59 symbols by stacking two symbols, those for one and ten. Search the Internet to answer the following questions:

a. Express the following decimal numbers in Babylonian numerals: 11291, 3646, 3582.
b. Mention problems that might arise from not having a symbol for 0. Find how the Babylonian numeral system addresses the problem.

3.1 DATA TYPES

Data today comes in different forms including numbers, text, audio, image, and video (Figure 3.1).

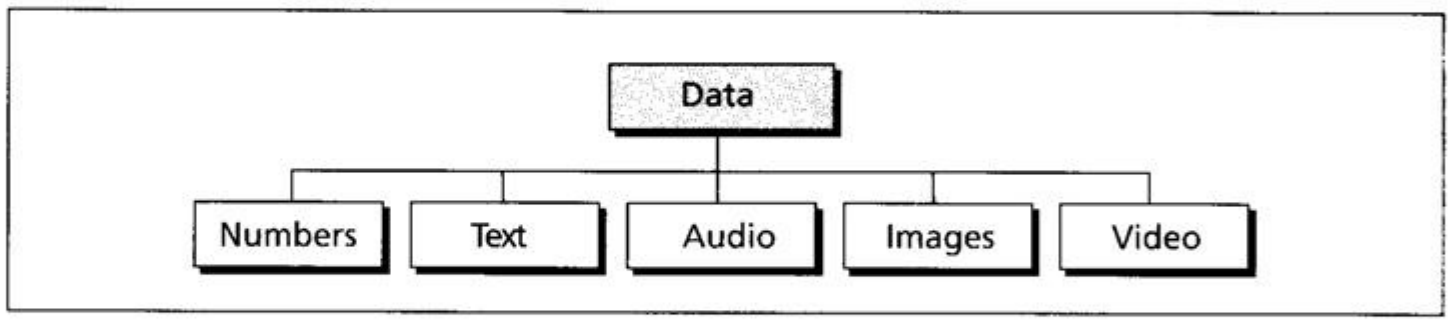


Figure 3.1 Different types of data

People need to be able to process many different types of data:

- An engineering program uses a computer mainly to process numbers: to do arithmetic, to solve algebraic or trigonometric equations, to find the roots of a differential equation, and so on.
- A word processing program, on the other hand, uses a computer mainly to process text: justify, move, delete, and so on.
- A computer also handles audio data. We can play music on a computer and can records sound as data.
- An image processing program uses a computer to manipulate images: create, shrink, expand, rotate, and so on.
- Finally, a computer can be used not only to show movies, but also to create the special effects seen in movies.

The computer industry uses the term "multimedia" to define information that contains numbers, text, images, audio, and video.

Data inside the computer

All data types are transformed into a uniform representation when they are stored in a computer and transformed back to their original form when retrieved. This universal representation is called a *bit pattern*, as discussed shortly.

Bits

A **bit (binary digit)** is the smallest unit of data that can be stored in a computer and has a value of 0 or 1. A bit represents the state of a device that can take one of two states. For example, a *switch* can be on or off. A convention can be established to represent the "on" state as 1 and the "off" state as 0, or vice versa. In this way, a switch can store one bit of information. Today, computers use various two-state devices to store data.

Bit patterns

To represent different types of data, we use a **bit pattern**, a sequence, or as it is sometimes called, a *string of bits*. Figure 3.2 shows a bit pattern made up of sixteen

bits. It is a combination of sixteen 0s and 1s. This means that if we need to store a bit pattern made of sixteen bits, we need sixteen electronic switches. If we need to store 1000 bit patterns, each sixteen bits long, we need 16,000 switches, and so on. By tradition a bit pattern with eight bits is called a **byte**. Sometimes the term **word** is used to refer to a longer bit pattern.

1 0 0 0 1 0 1 0 1 1 1 1 1 1

Figure 3.2 A bit pattern

As Figure 3.3 shows, a piece of data belonging to different data types can be stored as the same pattern in the memory.

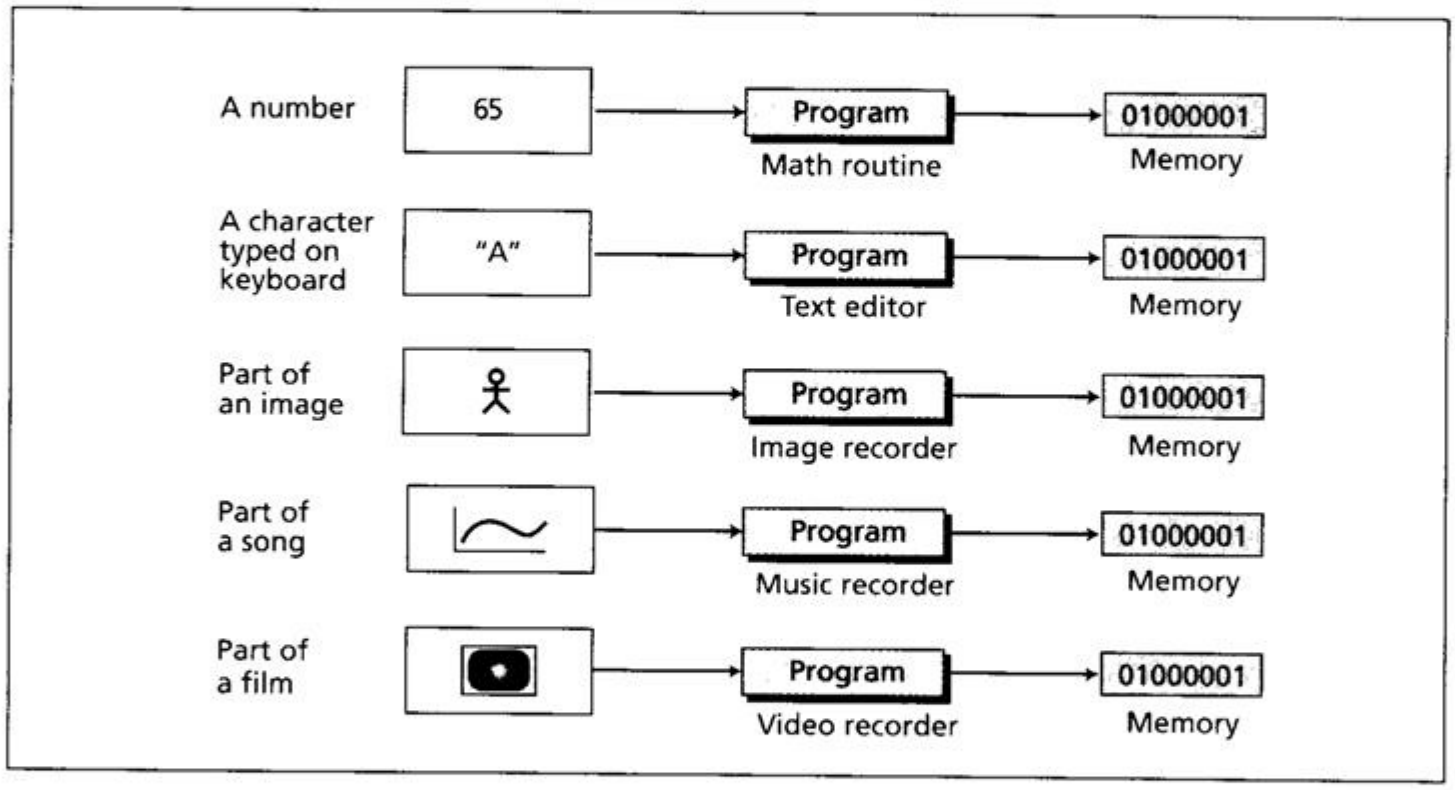


Figure 3.3 Storage of different data types

If we are using a text editor (a word processor), the character A typed on the keyboard can be stored as the 8-bit pattern 01000001. The same 8-bit pattern can represent the number 65 if we are using a mathematical routine. In the same way, the same pattern can represent part of an image, part of a song, or part of a scene in a film. The computer's memory stores all of them without recognizing what type of data they represent.

Data compression

To occupy less memory space, data is normally compressed before being stored in the computer. Data compression is a very broad and involved subject, so we have dedicated the whole of Chapter 15 to it.

Data compression is discussed in Chapter 15.

Error detection and correction

Another issue related to data is the detection and correction of errors during transmission or storage. We discuss this issue briefly in Appendix H.

Error detection and correction is discussed in Appendix H.

3.2 STORING NUMBERS

A number is changed to the binary system before being stored in the computer's memory, as described in Chapter 2. However, there are still two issues that need to be handled:

1. How to store the sign of the number.
2. How to show the decimal point.

There are several ways to handle the sign issue, discussed later in this chapter. For the decimal point, computers use two different representations: fixed-point and floating-point. The first is used to store a number as an integer—without a fractional part, the second is used to store a number as a real—with a fractional part.

Storing integers

Integers are whole numbers (numbers without a fractional part). For example, 134 and −125 are integers, whereas 134.23 and −0.235 are not. An integer can be thought of as a number in which the position of the decimal point is fixed: the decimal point is to the right of the least significant (rightmost) bit. For this reason, **fixed-point representation** is used to store an integer, as shown in Figure 3.4. In this representation the decimal point is assumed but not stored.

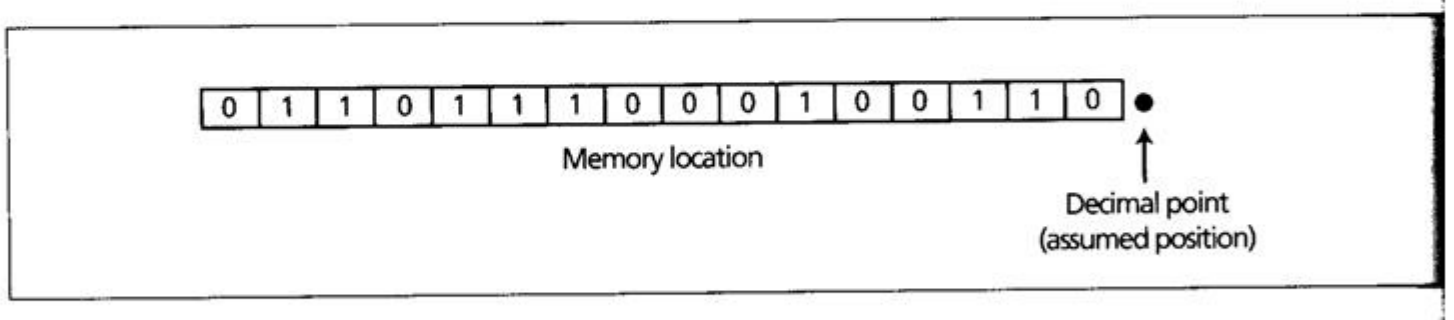


Figure 3.4 Fixed point representation of integers

However, a user (or a program) may store an integer as a real with the fractional part set to zero. This may happen, for example, if an integer is too large to be stored in the size defined for an integer. To use computer memory more efficiently, unsigned and signed integers are stored inside the computer differently.

An integer is normally stored in memory using fixed-point representation.

An **unsigned integer** is an integer that can never be negative and can take only 0 or positive values. Its range is between 0 and positive infinity. However, since no computer can possibly represent all the integers in this range, most computers define a constant called the *maximum unsigned integer*, which has the value of $(2^n - 1)$ where n is the number of bits allocated to represent an unsigned integer.

Storing unsigned integers

An input device stores an unsigned integer using the following steps:

- The integer is changed to binary.
- If the number of bits is less than n, 0s are added to the left of the binary integer so that there is a total of n bits. If the number of bits is greater than n, the integer cannot be stored. A condition referred to as *overflow* will occur, which we discuss later.

Example 3.1

Store 7 in an 8-bit memory location using unsigned representation.

Solution

First change the integer to binary, $(111)_2$. Add five 0s to make a total of eight bits, $(00000111)_2$. The integer is stored in the memory location. Note that the subscript 2 is used to emphasize that the integer is binary, but the subscript is not stored in the computer.

Change 7 to binary	→						1	1	1
Add five bits at the left	→	0	0	0	0	0	1	1	1

Example 3.2

Store 258 in a 16-bit memory location.

Solution

First change the integer to binary $(100000010)_2$. Add seven 0s to make a total of sixteen bits, $(0000000100000010)_2$. The integer is stored in the memory location.

Change 258 to binary	→								1	0	0	0	0	0	0	1	0
Add seven bits at the left	→	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0

Retrieving unsigned integers

An output device retrieves a bit string from memory as a bit pattern and converts it to an unsigned decimal integer.

Example 3.3

What is returned from an output device when it retrieves the bit string 00101011 stored in memory as an unsigned integer?

Solution

Using the procedure shown in Chapter 2, the binary integer is converted to the unsigned integer 43.

Overflow

Due to size limitations—the allocated number of bits—the range of integers that can be represented is limited. In an n-bit memory location we can only store an unsigned integer between 0 and $2^n - 1$. Figure 3.5 shows what happens if we try to store an integer that is larger than $2^4 - 1 = 15$ in a memory location that can only hold four bits. This situation, called **overflow**, happens when, for example, we have stored the integer 11 in a memory location and then try to add 9 to the integer. The minimum number of bits we need to represent the decimal 20 is five bits. In other words, $20 = (10100)_2$, so the computer drops the leftmost bit and keeps the rightmost four bits $(0100)_2$. People are surprised when they see that the new integer is printed as 4 instead of 20. Figure 3.5 shows why this happens.

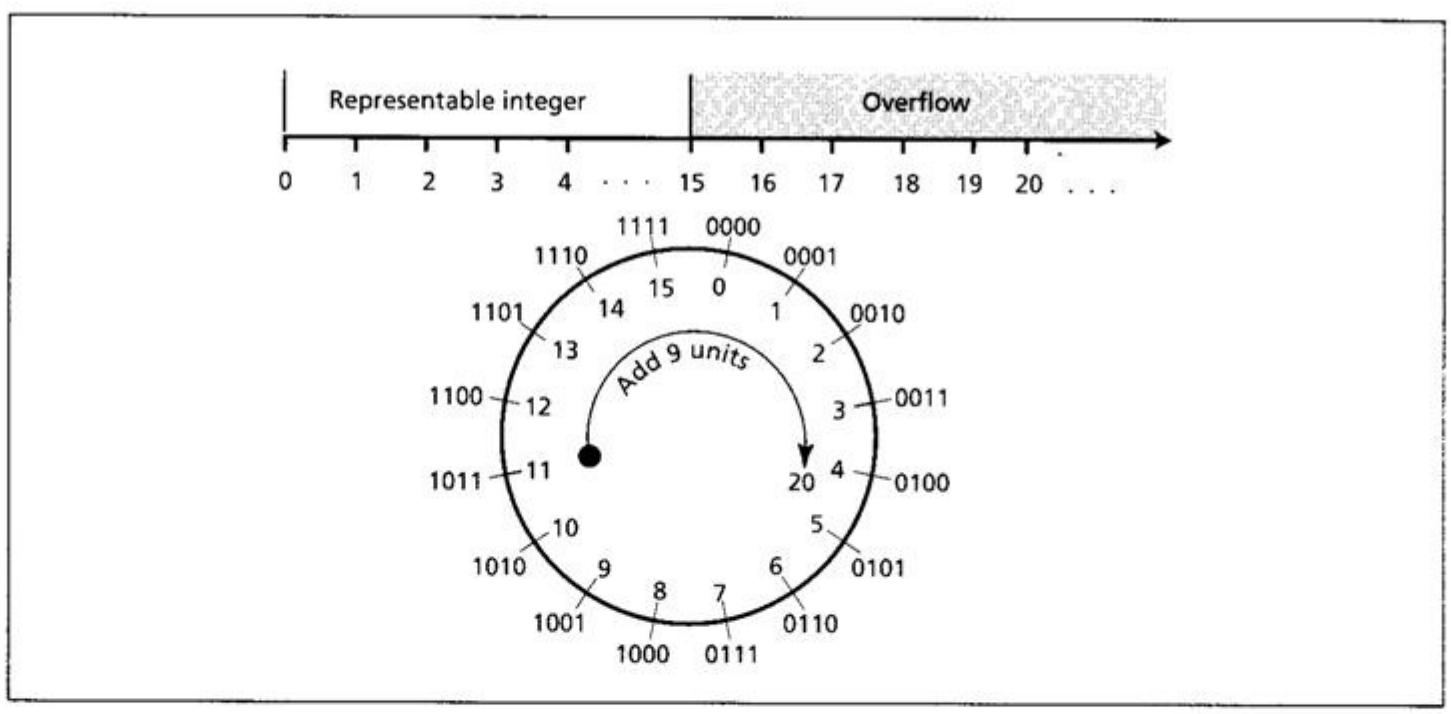


Figure 3.5 Overflow in unsigned integers

Applications of unsigned integers

Unsigned integer representation can improve the efficiency of storage because we do not need to store the sign of the integer. This means that the entire bit allocation

can be used for storing the number. Unsigned integer representation can be used whenever we do not need negative integers. The following lists some cases:

- *Counting*. When we count, we do not need negative numbers. We start counting from 1 (sometimes 0) and go up.
- *Addressing*. Some computer programs store the address of a memory location inside another memory location. Addresses are positive integers starting from 0 (the first memory location) and going up to an integer representing the total memory capacity. Here again, we do not need negative integers—unsigned integers can easily do the job.
- *Storing other data types*. Other data types (text, images, audio, and video), as we will discuss shortly, are stored as bit patterns, which can be interpreted as unsigned integers.

Although the sign-and-magnitude format is not commonly used to store integers, this format is used to store part of a real number in a computer, as described in the next section. For this reason we briefly discuss this format here. In this method, the available range for unsigned integers (0 to $2^n - 1$) is divided into two equal subranges. The first half represents positive integers, the second half, negative integers. For example, if n is 4, the range is 0000 to 1111. This range is divided into two halves: 0000 to 0111 and 1000 to 1111 (Figure 3.6). The bit patterns are then assigned to negative and positive integers. Note that the negative numbers appear to the right of the positive numbers, which is contrary to conventional thinking about positive and negative numbers. Also note that we have two 0s: positive zero (0000) and negative zero (1000).

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	–0	–1	–2	–3	–4	–5	–6	–7

Figure 3.6 Sign-and-magnitude representation

Storing an integer in sign-and-magnitude format requires one bit to represent the sign (0 for positive, 1 for negative). This means that in an 8-bit allocation, we can only use seven bits to represent the absolute value of the number (number without the sign). Therefore, the maximum positive value is one half the unsigned value. The range of numbers that can be stored in an n-bit location is $-(2^{n-1} - 1)$ to $+ (2^{n-1} - 1)$. In an n-bit allocation, the leftmost bit is dedicated to store the sign (0 for positive, 1 for negative).

In sign-and-magnitude representation, the leftmost bit defines the sign of the integer. If it is 0, the integer is positive. If it is 1, the integer is negative.

Example 3.4

Store +28 in an 8-bit memory location using sign-and-magnitude representation.

Solution

The integer is changed to 7-bit binary. The leftmost bit is set to 0. The 8-bit number is stored.

Change 28 to 7-bit binary		0	0	1	1	1	0	0
Add the sign and store	0	0	0	1	1	1	0	0

Example 3.5

Store –28 in an 8-bit memory location using sign-and-magnitude representation.

Solution

The integer is changed to 7-bit binary. The leftmost bit is set to 1. The 8-bit number is stored.

Change 28 to 7-bit binary		0	0	1	1	1	0	0
Add the sign and store	1	0	0	1	1	1	0	0

Example 3.6

Retrieve the integer that is stored as 01001101 in sign-and-magnitude representation.

Solution

Since the leftmost bit is 0, the sign is positive. The rest of the bits (1001101) are changed to decimal as 77. After adding the sign, the integer is +77.

Example 3.7

Retrieve the integer that is stored as 10100001 in sign-and-magnitude representation.

Solution

Since the leftmost bit is 1, the sign is negative. The rest of the bits (0100001) are changed to decimal as 17. After adding the sign, the integer is –17.

Overflow in sign-and-magnitude representation

Like unsigned integers, signed integers are also subjected to overflow. However, in this case, we may have both positive and negative overflow. Figure 3.7 shows both positive and negative overflow when storing an integer in sign-and-magnitude representation using a 4-bit memory location. Positive overflow occurs when we try to store a positive integer larger than 7. For example, assume that we have stored integer 5 in a memory location and we then try to add 6 to the integer. We expect the result to be 11, but the computer's response is –3. The reason is that if we start from 5 on a circular representation and go six units in the clockwise direction, we end up at –3. A positive overflow wraps the integer back to the range.

A negative overflow can happen when we try to store a integer that is less than –7, for example if we have stored the integer –5 in a memory and try to subtract 7 from it. We expect the result to be –12, but the computer's response is +6. The

reason is that if we start from –5 on a circular representation and go seven units in the counterclockwise direction, we end up at +6.

There are two 0s in sign-and-magnitude representation: +0 and –0.

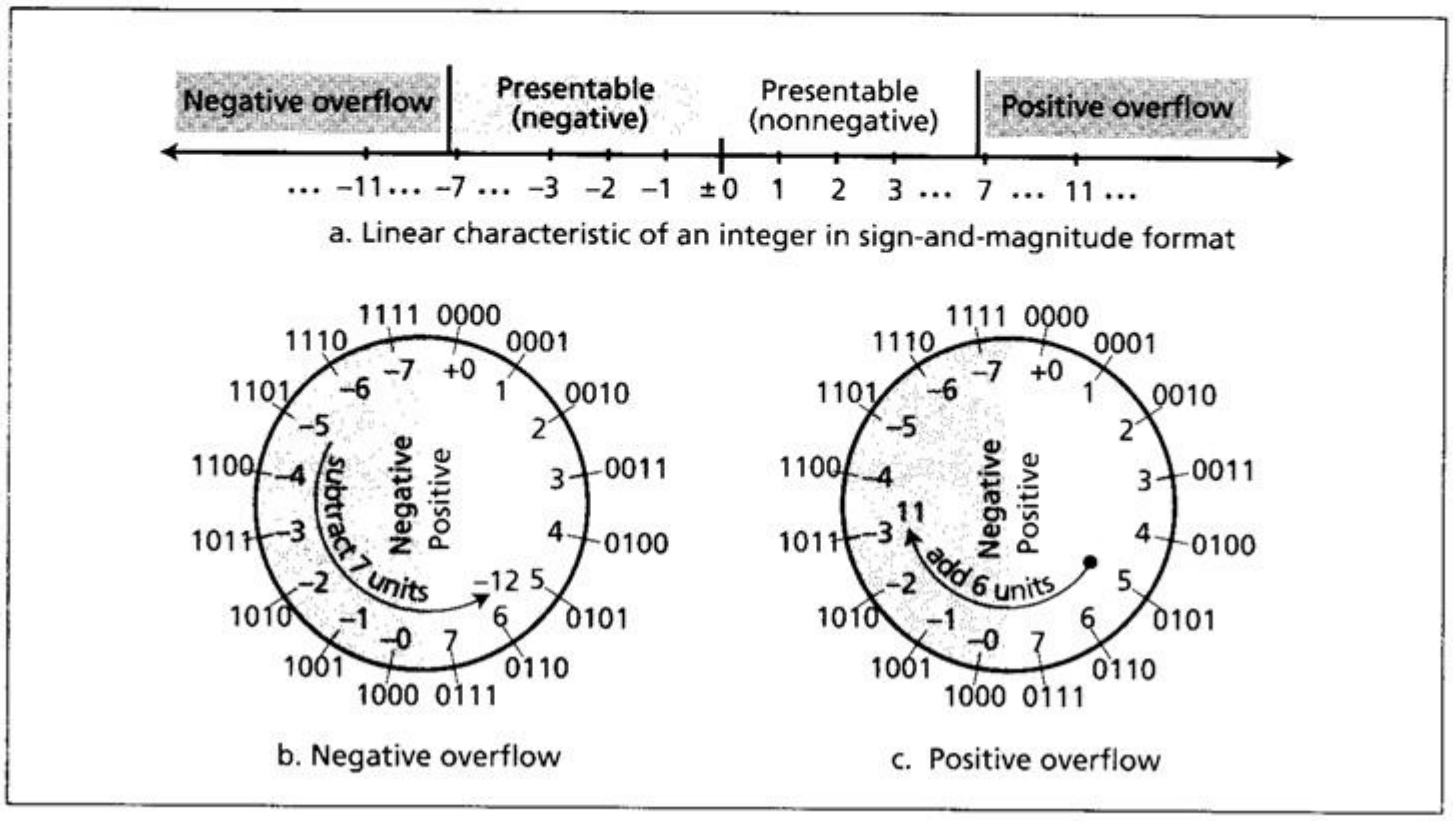


Figure 3.7 Overflow in sign-and-magnitude representation

Applications of sign-and-magnitude representation

Sign-and-magnitude representation is not used to store integers. However, it is used to store parts of real numbers, as we will see shortly. In addition, sign-and-magnitude representation is often used when we quantize an analog signal, such as audio.

Almost all computers use **two's complement** representation to store a signed integer in an n-bit memory location. In this method, the available range for an unsigned integer of (0 to $2^n - 1$) is divided into two equal subranges. The first subrange is used to represent nonnegative integers, the second half to represent negative integers. For example, if n is 4, the range is 0000 to 1111. This range is divided into two halves: 0000 to 0111 and 1000 to 1111. The two halves are swapped to be in agreement with the common convention of showing negative integers to the left of positive integers. The bit patterns are then assigned to negative and nonnegative (zero and positive) integers, as shown in Figure 3.8.

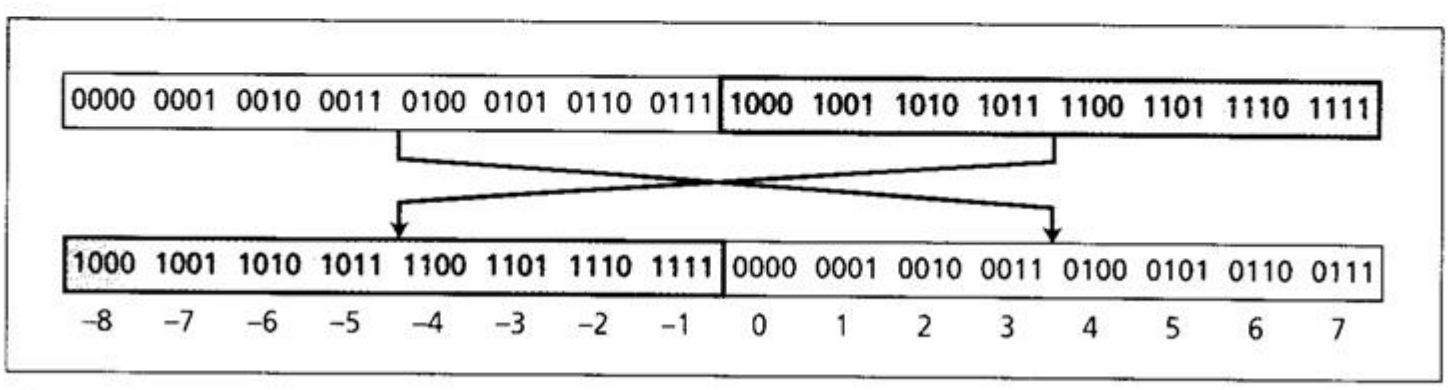


Figure 3.8 Two's complement representation

Although the sign of the integer affects every bit in the binary integer stored, the first (leftmost) bit determines the sign. If the leftmost bit is 0, the integer is nonnegative: if the leftmost bit is 1, the integer is negative.

In two's complement representation, the leftmost bit defines the sign of the integer. If it is 0, the integer is positive. If it is 1, the integer is negative.

Two operations

Before we discuss this representation further, we need to introduce two operations. The first is called *one's completing* or *taking the one's complement of an integer*. The operation can be applied to any integer, positive or negative. This operation simply reverses (flips) each bit. A 0-bit is changed to a 1-bit, a 1-bit is changed to a 0-bit.

Example 3.8

The following shows how we take the one's complement of the integer 00110110.

Original pattern	0	0	1	1	0	1	1	0
After applying one's complement operation	1	1	0	0	1	0	0	1

Example 3.9

The following shows that we get the original integer if we apply the one's complement operations twice.

Original pattern	0	0	1	1	0	1	1	0
One's complementing once	1	1	0	0	1	0	0	1
One's complementing twice	0	0	1	1	0	1	1	0

The second operation is called *two's completing* or *taking the two's complement* of an integer in binary. This operation is done in two steps. First, we copy bits from the right until a 1 is copied, Then, we flip the rest of the bits.

Example 3.10

The following shows how we take the two's complement of the integer 00110100.

Original integer	0	0	1	1	0	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing once	1	1	0	0	1	1	0	0

Example 3.11

The following shows that we always get the original integer if we apply the two's complement operation twice.

Original integer	0	0	1	1	0	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing once	1	1	0	0	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing twice	0	0	1	1	0	1	0	0

An alternative way to take the two's complement of an integer is to first take the one's complement and then add 1 to the result (see Chapter 4 for binary addition).

Storing an integer in two's complement format

To store an integer in two's complement representation, the computer follows the steps below:

- The integer is changed to an n-bit binary.
- If the integer is positive or zero, it is stored as it is: if it is negative, the computer takes the two's complement of the integer and then stores it.

Retrieving an integer in two's complement format

To retrieve an integer in two's complement representation, the computer follows the steps below:

- If the leftmost bit is 1, the computer applies the two's complement operation to the integer. If the leftmost bit is 0, no operation is applied.
- The computer changes the integer to decimal.

Example 3.12

Store the integer 28 in an 8-bit memory location using two's complement representation.

Solution

The integer is positive (no sign means positive), so after decimal to binary transformation no more action is needed. Note that five extra 0s are added to the left of the integer to make it eight bits.

Change 28 to 8-bit binary	0	0	0	1	1	1	0	0

Example 3.13

Store –28 in an 8-bit memory location using two's complement representation.

Solution

The integer is negative, so after changing to binary, the computer applies the two's complement operation on the integer.

Change 28 to 8-bit binary	0	0	0	1	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Apply two's complement operation	1	1	1	0	0	1	0	0

Example 3.14

Retrieve the integer that is stored as 00001101 in memory in two's complement format.

Solution

The leftmost bit is 0, so the sign is positive. The integer is changed to decimal and the sign is added.

Leftmost bit is 0. The sign is positive	0	0	0	0	1	1	0	1
Integer changed to decimal								13
Sign is added (optional)								+13

Example 3.15

Retrieve the integer that is stored as 11100110 in memory using two's complement format.

Solution

The leftmost bit is 1, so the integer is negative. The integer needs to be two's complemented before changing to decimal.

Leftmost bit is 1. The sign is negative	1	1	1	0	0	1	1	0
	↓	↓	↓	↓	↓	↓	↓	↓
Apply two's complement operation	0	0	0	1	1	0	1	0
Integer changed to decimal								26
Sign is added								–26

A very interesting point about two's complement is that there is only one zero in this representation. In sign-and-magnitude representation, there are two zeros (+0 and –0).

There is only one zero in two's complement notation.

Overflow in two's complement notation

Like other representations, integers stored in two's complement format are also subject to overflow. Figure 3.9 shows both positive and negative overflow when storing a signed integer in a 4-bit memory location. Positive overflow occurs when we try to store a positive integer larger than 7. For example, assume that we have stored an integer value 5 in a memory location and we then try to add 6 to the integer. We expect the result to be 11, but the computer's response is –5. The reason is if we start from 5 on the circular representation and move six units in the clockwise direction, we end up at –5. The positive overflow wraps the integer back to the range.

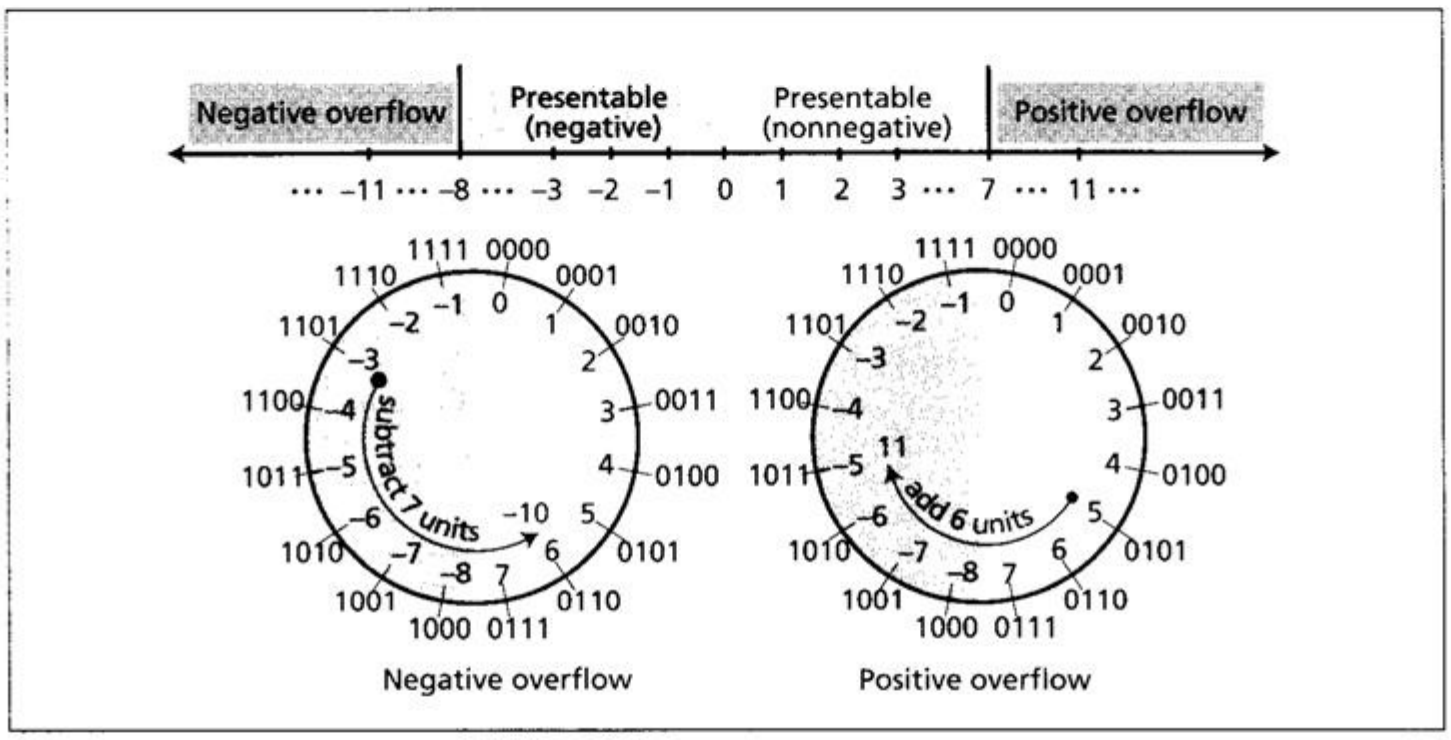


Figure 3.9 Overflow in two's complement representation

A negative overflow can happen when we try to store a integer that is less than –8, for example if we have stored –3 and try to subtract 7 from it. We expect the result to be –10, but the computer's response is +6. The reason is that if we start from –3 on a circular representation and go seven units in the counterclockwise direction, we end up at +6.

Applications of two's complement notation

Two's complement representation is the standard representation for storing integers in computers today. In the next chapter we will see why this is the case when we see the simplicity of operations using two's complement.

Comparison of the three systems

Table 3.1 shows a comparison between unsigned, two's complement, and sign-and-magnitude integers. A 4-bit memory location can store an unsigned integer between 0 and 15, and the same location can store two's complement signed integers between –8 and +7. It is very important that we store and retrieve an integer in the same format. For example, if the integer 13 is stored in signed format, it

needs to be retrieved in signed format: the same integer is retrieved as –3 in two's complement format.

Table 3.1 Summary of integer representations

Contents of memory	*Unsigned*	*Sign-and-magnitude*	*Two's complement*
0000	0	0	+0
0001	1	1	+1
0010	2	2	+2
0011	3	3	+3
0100	4	4	+4
0101	5	5	+5
0110	6	6	+6
0111	7	7	+7
1000	8	–0	–8
1001	9	–1	–7
1010	10	–2	–6
1011	11	–3	–5
1100	12	–4	–4
1101	13	–5	–3
1110	14	–6	–2
1111	15	–7	–1

Storing reals

A **real** is a number with an integral part and a fractional part. For example, 23.7 is a real number—the integral part is 23 and the fractional part is 7/10. Although a fixed-point representation can be used to represent a real number, the result may not be accurate or it may not have the required precision. The next two examples explain why.

Example 3.16

In the decimal system, assume that we use a fixed-point representation with two digits at the right of the decimal point and fourteen digits at the left of the decimal point, for a total of sixteen digits. The precision of a real number in this system is lost if we try to represent a decimal number such as 1.00234: the system stores the number as 1.00.

Example 3.17

In the decimal system, assume that we use a fixed-point representation with six digits to the right of the decimal point and ten digits to the left of the decimal point, for a total of sixteen digits. The accuracy of a real number in this system is lost if we try to represent a decimal number such as 236154302345.00. The system stores the number as 6154302345.00: the integral part is much smaller than it should be.

Real numbers with very large integral parts or very small fractional parts should not be stored in fixed-point representation.

The solution for maintaining accuracy or precision is to use **floating-point representation**. This representation allows the decimal point to *float*: we can have different numbers of digits to the left or right of the decimal point. The range of real numbers that can be stored using this method increases tremendously: numbers with large integral parts or small fractional parts can be stored in memory. In floating-point representation, either decimal or binary, a number is made up of three sections, as shown in Figure 3.10.

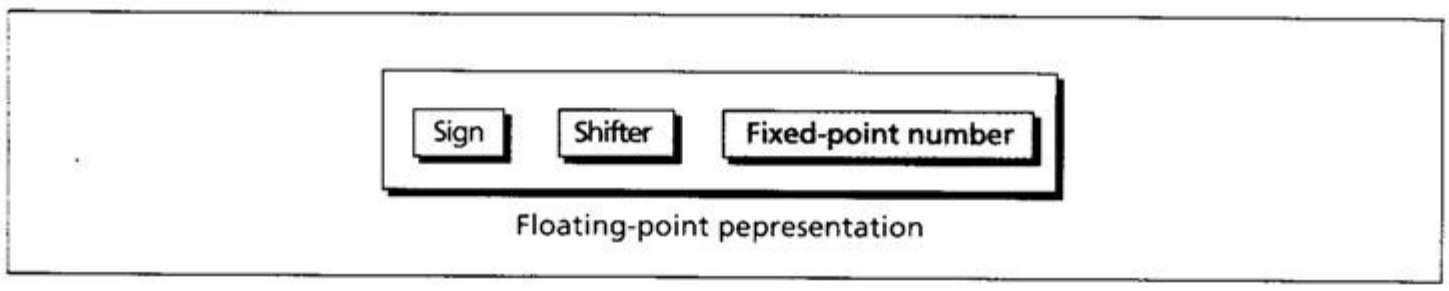


Figure 3.10 The three parts of a real number in floating-point representation

The first section is the sign, either positive or negative. The second section shows how many places the decimal point should be shifted to the right or left to form the actual number. The third section is a fixed-point representation in which the position of the decimal is fixed.

A floating point representation of a number is made up of three parts: a sign, a shifter, and a fixed-point number.

Floating-point representation is used in science to represent very small or very large decimal numbers. In this representation, which is called *scientific notation*, the fixed-point section has only one digit to the left of the decimal point and the shifter is the power of 10.

Example 3.18

The following shows the decimal number 7425,000,000,000,000,000,000.00 in scientific notation (floating-point representation).

Solution

Actual number	→	+	7,425,000,000,000,000,000,000.00
Scientific notation	→	+	7.425×10^{21}

The three sections are the sign (+), the shifter (21), and the fixed-point part (7.425). Note that the shifter is the exponent. We can easily see the advantage of this. Even if we just want to write the number on a piece of paper, the scientific notation is shorter and takes less space. The notation uses the concept of floating-point because the position of the decimal point, which is near the right-hand end in the example, has moved 21 digits to the left to make the fixed-point part of the number. Some programming languages and calculators shows the number as +7.425E21 because the base 10 is understood and does not need to be mentioned.

Example 3.19

Show the number –0.0000000000000232 in scientific notation.

Solution

We use the same approach as in the previous example—we move the decimal point after the digit 2, as shown below:

Actual number	→	–	0.0000000000000232
Scientific notation	→	–	2.32×10^{-14}

Note that the exponent is negative here because the decimal point in 2.32 needs to move to the left (fourteen positions) to form the original number. Again, we can say that the number in this notation is made of three parts: sign (–), the real number (2.32), and the negative integer (–14). Some programming languages and calculators show this as –2.32E–14.

Similar approaches have been used to represent very large or very small numbers (both integers and reals) in binary, to be stored in computers.

Example 3.20

Show the number $(10100100000000000000000000000000.00)_2$ in floating-point format.

Solution

We use the same idea, keeping only one digit to the left of the decimal point.

Actual number	→	+	$(10100100000000000000000000000000.00)_2$
Scientific notation	→	+	1.01001×2^{32}

Note that we don't have to worry about all those 0s at the right of the rightmost 1, because they are not significant when we use the real $(1.01001)_2$. The exponent is shown as 32, but it is actually stored in the computer in binary, as we will see shortly. We have also shown the sign as positive, but it would be stored as one bit.

Example 3.21

Show the number $-(0.00000000000000000000000101)_2$ in floating-point format.

Solution

We use the same idea, keeping only one non-zero digit on the left-hand side of the decimal point.

Actual number	→	–	$(0.00000000000000000000000101)_2$
Scientific notation	→	–	1.01×2^{-24}

Note that exponent is stored as a negative binary in the computer.

Normalization To make the fixed part of the representation uniform, both the scientific method (for the decimal system) and the floating-point method (for the binary system) use only one non-zero digit to the left of the decimal point. This is called **normalization**. In the decimal system this digit can be 1 to 9, while in the binary system it can only be 1. In the following, d is a non-zero digit, x is a digit, and y is either 0 or 1.

Decimal	→	±	d.xxxxxxxxxxxxxx	Note: d is 1 to 9 and each x is 0 to 9
Binary	→	±	1.yyyyyyyyyyyyyy	Note: each y is 0 or 1

Sign, exponent, mantissa After a binary number is normalized, only three pieces of information about the number are stored: sign, exponent, and mantissa (the bits to the right of the decimal point). For example, +1000111.0101 becomes:

+	2^6	×	1.0001110101
+	6		0001110101
↑	↑		↑
Sign	Exponent		Mantissa

Note that the point and the bit 1 to the left of the fixed-point section are not stored—they are implicit.

Sign

The sign of the number can be stored using 1 bit (0 or 1).

Exponent

The exponent (power of 2) defines the shifting of the decimal point. Note that the power can be negative or positive. The **Excess** representation (discussed later) is the method used to store the exponent.

Mantissa

The **mantissa** is the binary integer to the right of the decimal point. It defines the precision of the number. The mantissa is stored in fixed-point notation. If we think of the mantissa and the sign together, we can say this combination is stored as an integer in sign-and-magnitude format. However, we need to remember that it is not an integer—it is a fractional part that is stored like an integer. We emphasize this point because in a mantissa, if we insert extra 0s to the *right* of the number, the value will not change, whereas in a real integer if we insert extra 0s to the *left* of the number, the value will not change.

The mantissa is a fractional part that, together with the sign, is treated like an integer stored in sign-and-magnitude representation.

The Excess system

The mantissa can be stored as an unsigned integer. The exponent, the power that shows how many bits the decimal point should be moved to the left or right, is a signed number. Although this could have been stored using two's complement representation, a new representation, called the Excess system, is used instead. In the Excess system, both positive and negative integers are stored as unsigned integers. To represent a positive or negative integer, a positive integer (called a *bias*) is added to each number to shift them uniformly to the non-negative side. The value of this bias is $2^{m-1} - 1$, where m is the size of the memory location to store the exponent.

Example 3.22

We can express sixteen integers in a number system with 4-bit allocation. Using one location for 0 and splitting the other fifteen (not quite equally) we can express integers in the range of –7 to 8, as shown in Figure 3.11. By adding seven units to each integer in this range, we can uniformly translate all integers to the right and make all of them positive without changing the relative position of the integers with respect to each other, as shown in the figure. The new system is referred to as *Excess-7*, or biased representation with biasing value of 7.

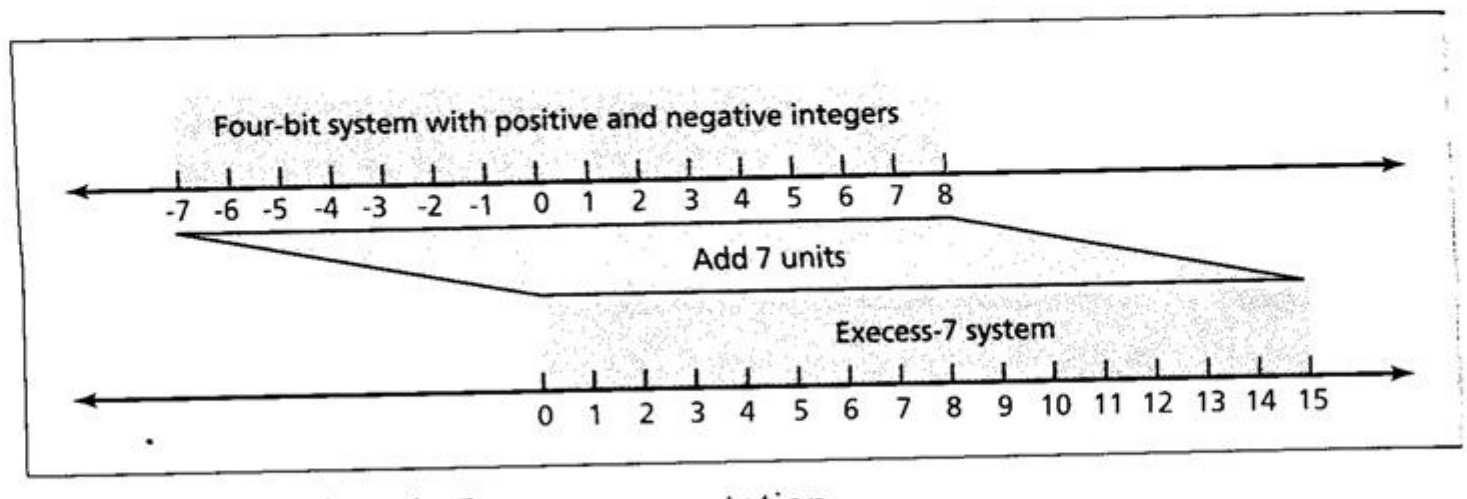


Figure 3.11 Shifting in Excess representation

The advantage of this new representation compared to that before the translation is that all integers in the Excess system are positive, so we don't need to be concerned about the sign when we are comparing or doing operations on the integers. For 4-bit allocation, the bias is $2^{4-1} - 1 = 7$, as we expected.

The Institute of Electrical and Electronics Engineers (IEEE) has defined several standards for storing floating-point numbers. We discuss the two most common ones here, single precision and double precision. These formats are shown in Figure 3.12. The numbers above the boxes are the number of bits for each field.

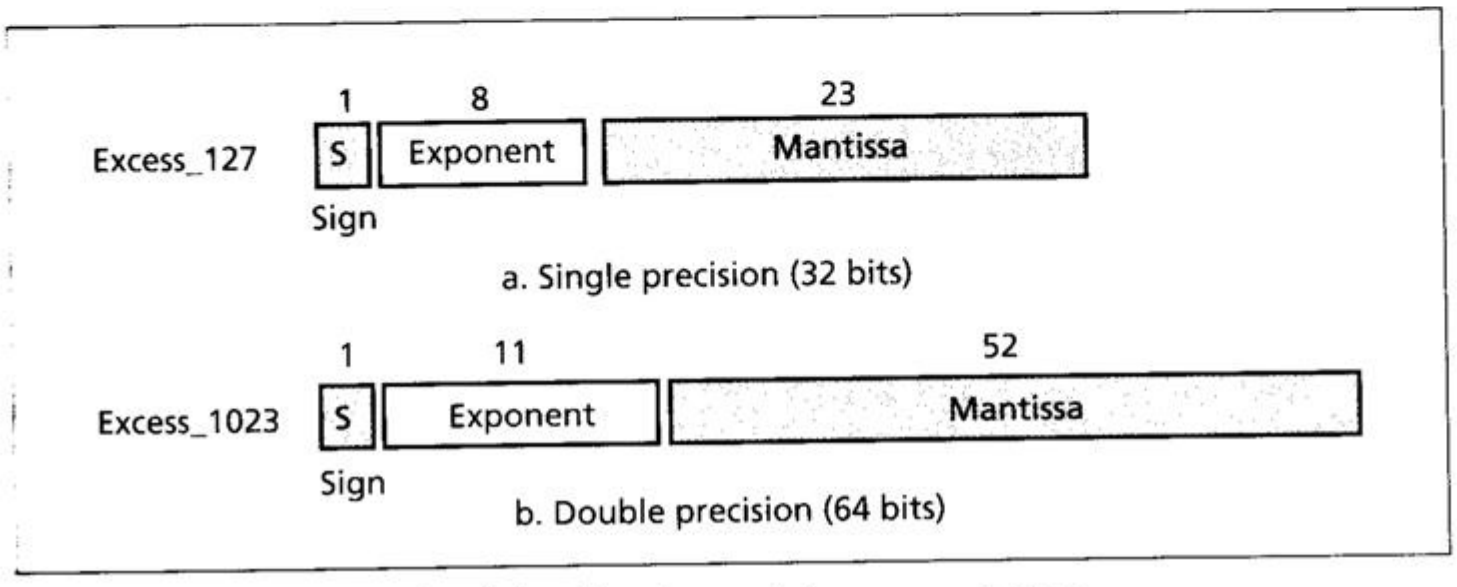


Figure 3.12 IEEE standards for floating-point representation

Single precision format uses a total of 32 bits to store a real number in floating-point representation. The sign occupies one bit (0 for positive and 1 for negative), the exponent occupies eight bits (using a bias of 127), the mantissa uses 23 bits (unsigned number). This standard is sometimes referred to as **Excess_127** because the bias is 127.

Double precision format uses a total of 64 bits to store a real number in floating-point representation. The sign occupies one bit, the exponent occupies eleven bits (using a bias of 1023), and the mantissa uses 52 bits. The standard is sometimes referred to as **Excess_1023** because the bias is 1023. Table 3.2 summarizes the specification of the two standards.

Table 3.2 Specifications of the two IEEE floating-point standards

Parameter	*Single precision*	*Double precision*
Memory location size (number of bits)	32	64
Sign size (number of bits)	1	1
Exponent size (number of bits)	8	11
Mantissa size (number of bits)	23	52
Bias (integer)	127	1023

Storage of IEEE standard floating point numbers

A real number can be stored in one of the IEEE standard floating-point formats using the following procedure, with reference to Figure 3.12:

1. Store the sign in S (0 or 1).
2. Change the number to binary.
3. Normalize.
4. Find the values of E and M.
5. Concatenate S, E, and M.

Example 3.23

Show the Excess_127 (single precision) representation of the decimal number 5.75.

Solution

a. The sign is positive, so S = 0.

b. Decimal to binary transformation: $5.75 = (101.11)_2$.

c. Normalization: $(101.11)_2 = (1.0111)_2 \times 2^2$.

d. $E = 2 + 127 = 129 = (10000001)_2$, M = 0111. We need to add nineteen zeros at the right of M to make it 23 bits.

e. The presentation is shown below:

0	**10000001**	**01110000000000000000000**
S	**E**	**M**

The number is stored in the computer as 01000000101110000000000000000000.

Example 3.24

Show the Excess_127 (single precision) representation of the decimal number –161.875.

Solution

a. The sign is negative, so S = 1.

b. Decimal to binary transformation: $161.875 = (10100001.111)_2$.

c. Normalization: $(10100001.111)_2 = (1.0100001111)_2 \times 2^7$.

d. $E = 7 + 127 = 134 = (10000110)_2$ and $M = (0100001111)_2$.

e. Representation:

1	**10000110**	**01000011110000000000000**
S	**E**	**M**

The number is stored in the computer as 11000011010000111100000000000000.

Example 3.25

Show the Excess_127 (single precision) representation of the decimal number −0.0234375.

Solution

a. S = 1 (the number is negative).

b. Decimal to binary transformation: $0.0234375 = (0.0000011)_2$.

c. Normalization: $(0.0000011)_2 = (1.1)_2 \times 2^{-6}$.

d. $E = -6 + 127 = 121 = (01111001)_2$ and $M = (1)_2$.

e. Representation:

S	E	M
1	01111001	10000000000000000000000

The number is stored in the computer as 10111100110000000000000000000000.

A number stored in one of the IEEE floating-point formats can be retrieved using the following method:

1. Find the value of S, E, and M.
2. If S = 0, set the sign to positive, otherwise set the sign to negative.
3. Find the shifter (E −127).
4. Denormalize the mantissa.
5. Change the denormalized number to binary to find the absolute value.
6. Add the sign.

Example 3.26

The bit pattern $(11001010000000000111000100001111)_2$ is stored in memory in Excess_127 format. Show what the value of the number is in decimal notation.

Solution

a. The first bit represents S, the next eight bits, E, and the remaining 23 bits, M.

S	E	M
1	10010100	00000000111000100001111

b. The sign is negative.

c. The shifter = E − 127 = 148 − 127 = 21.

d. Denormalization gives us $(1.00000000111000100001111)_2 \times 2^{21}$.

e. The binary number is $(100000000111000100001 1.11)_2$.

f. The absolute value is 2,104,378.75.

g. The number is −2,104,378.75.

Overflow and underflow

In the case of floating point numbers, we can have both an overflow and **underflow**. Figure 3.13 shows the ranges of floating-point representations using 32-bit memory locations (Excess_127). This representation cannot store numbers with very small or very large absolute values. An attempt to store numbers with very small absolute values results in an underflow condition, while an attempt to store numbers with very large absolute values results in an overflow condition. We leave the calculation of boundary values (+largest, –largest, +smallest, and –smallest) as exercises.

Storing zero

You may have noticed that a real number with an integral part and the fractional part set to zero, that is, 0.0, cannot be stored using the steps discussed above. To handle this special case, it is agreed that in this case the sign, exponent, and the mantissa are set to 0s.

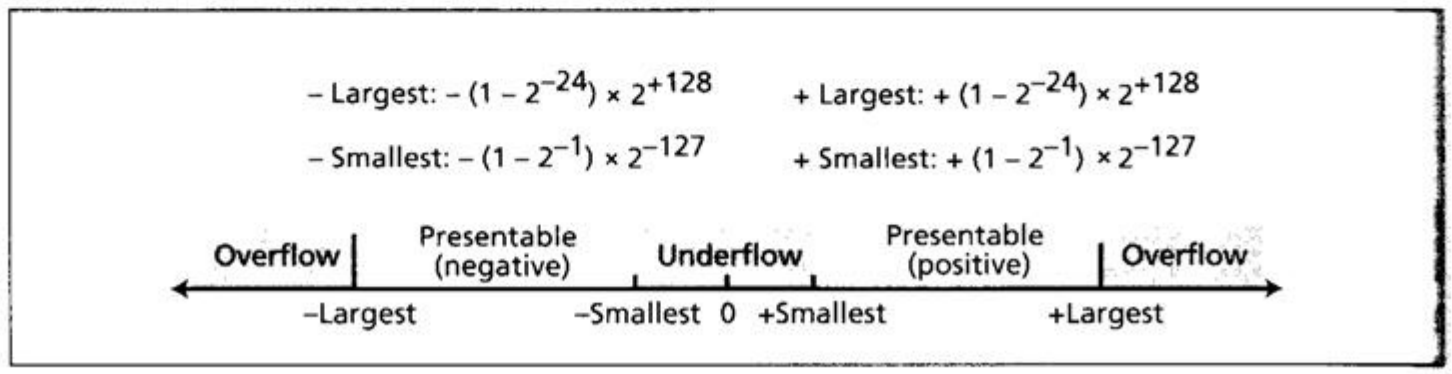


Figure 3.13 Overflow and underflow in floating-point representation of reals

Truncation errors

When a real number is stored using floating-point representation, the value of the number stored may not be exactly as we expect it to be. For example, assume we need to store the number:

$$(1111111111111111.11111111111)_2$$

in memory using Excess_127 representation. After normalization, we have:

$$(1.1111111111111111111111111)_2$$

This means that the mantissa has 26 1s. This mantissa needs to be truncated to 23 1s. In other words, what is stored in the computer is:

$$(11111111111111111.11111111)_2$$

with the three 1s at the right of the fractional part truncated. The difference between the original number and what is retrieved is called the **truncation error**. This type of error is very important in areas in which very small or very large number are used, such as calculations in the space industry. In such cases we need to use larger memory locations and other presentations. The IEEE defines other standards with larger mantissas for these purposes.

3.3 STORING TEXT

A section of **text** in any language is a sequence of symbols used to represent an idea in that language. For example, the English language uses 26 symbols (A, B, C,..., Z) to represent uppercase letters, 26 symbols (a, b, c, ..., z) to represent lowercase letters,

ten symbols (0, 1, 2, ..., 9) to represent numeric characters (not actual numbers—numbers are treated separately, as we explained in the previous section), and symbols (., ?, :, ; , ..., !) to represent punctuation. Other symbols such as blank, newline, and tab are used for text alignment and readability.

We can represent each symbol with a bit pattern. In other words, text such as "CATS", which is made up from four symbols, can be represented as four n-bit patterns, each pattern defining a single symbol (Figure 3.14).

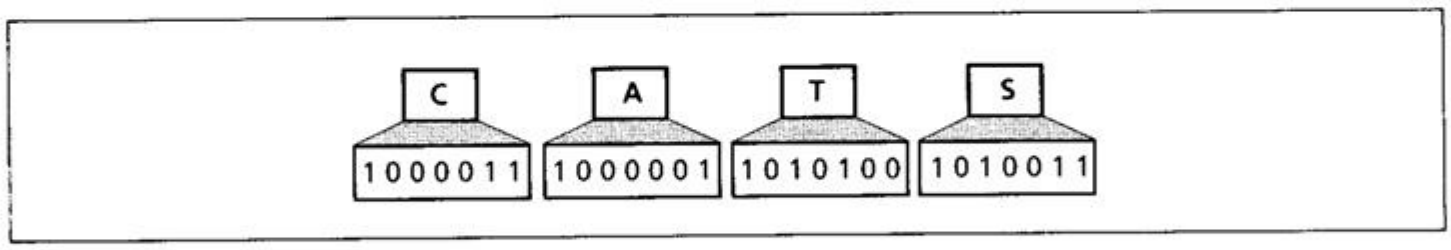


Figure 3.14 Representing symbols using bit patterns

Now the question is: how many bits are needed in a bit pattern to represent a symbol in a language? It depends on how many symbols are in the set used for the language. For example, if we create an imaginary language that uses only English uppercase letters, we need only 26 symbols. A bit pattern in this language needs to represent at least 26 symbols.

For another language, such as Chinese, we may need many more symbols. The length of the bit pattern that represents a symbol in a language depends on the number of symbols used in that language. More symbols mean a longer bit pattern.

Although the length of the bit pattern depends on the number of symbols, the relationship is not linear: it is logarithmic. If we need two symbols, the length is one bit ($\log_2 2$ is 1). If we need four symbols, the length is two bits ($\log_2 4$ is 2). Table 3.3 shows the relationship. A bit pattern of two bits can take four different forms: 00, 01, 10, and 11. Each of these forms can represent a symbol. In the same way, a bit pattern of three bits can take eight different forms: 000, 001, 010, 011, 100, 101, 110, and 111.

Table 3.3 Number of symbols and bit pattern length

Number of symbols	*Bit pattern length*	*Number of symbols*	*Bit pattern length*
2	1	128	7
4	2	256	8
8	3	65,536	16
16	4	4,294,967,296	32

Different sets of bit patterns have been designed to represent text symbols. Each set is called a **code**, and the process of representing symbols is called *coding*. In this section, we explain the common codes.

ASCII

The **American National Standards Institute (ANSI)** developed a code called **American Standard Code for Information Interchange (ASCII)**. This code uses seven bits for each symbol. This means that $2^7 = 128$ different symbols can be defined in this code. The full bit patterns for ASCII code are included in Appendix A. Today ASCII is part of Unicode, which is discussed next.

Unicode

A coalition of hardware and software manufacturers have designed a code called **Unicode** that uses 32 bits and can therefore represent up to $2^{32} = 4{,}294{,}967{,}296$ symbols. Different sections of the code are allocated to symbols from different languages in the world. Some parts of the code are used for graphical and special symbols. A brief set of Unicode symbols is listed in Appendix A. ASCII is part of Unicode today.

Other codes

Other codes have been developed during the last few decades. Most of these codes have been made less common with the advent of Unicode. We leave the exploration of these codes as an exercise.

3.4 STORING AUDIO

Audio is a representation of sound or music. Audio, by nature, is different than the numbers or text we have discussed so far. Text is composed of countable entities (characters): we can count the number of characters in text. Text is an example of **digital** data. In contrast, audio is not countable. Audio is an entity that changes with time—we can only measure the intensity of the sound at each moment. When we discuss storing audio in computer memory, we mean storing the intensity of an audio signal, such as the signal from a microphone, over a period of time: one second, one hour.

Audio is an example of **analog** data. Even if we are able to measure all its values in a period of time, we cannot store these in the computer's memory, as we would need infinite number of memory locations. Figure 3.15 shows the nature of an analog signal, such as audio, that varies with time.

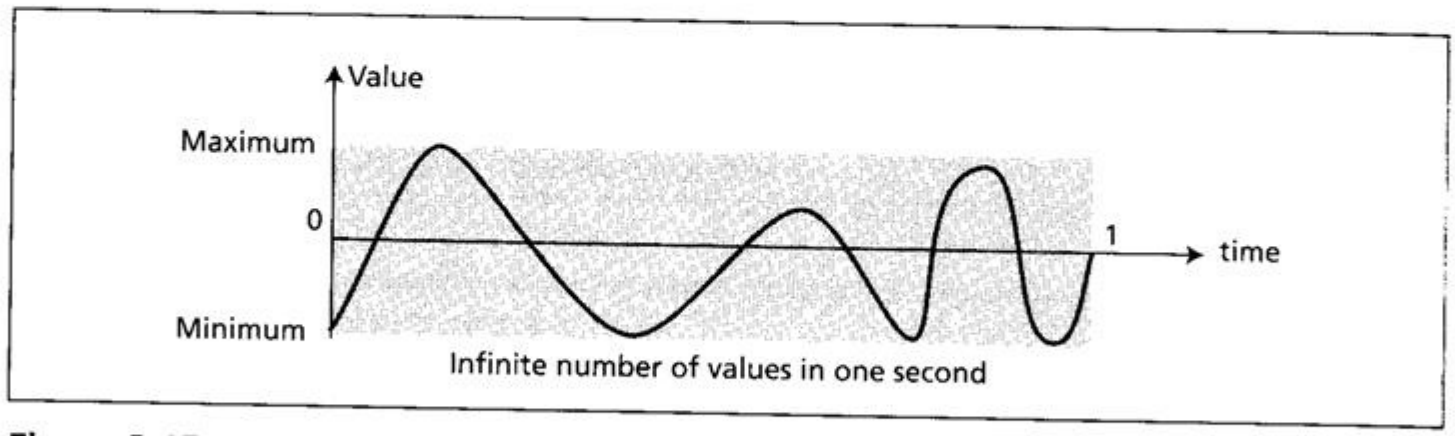


Figure 3.15 An audio signal

Sampling

If we cannot record all the values of an audio signal over an interval, we *can* record some of them. **Sampling** means that we select only a finite number of points on the analog signal, measure their values, and record them. Figure 3.16 shows that selection of ten samples from the signal: we can then record these values to represent the analog signal.

The next logical question is, how many samples do we need in each second to be able to retrieve a replica of the original signal? The number of samples depends on the maximum number of changes in the analog signal. If the signal is smooth, we need less samples: if the signal is changing rapidly, we need more samples. It has been shown that a **sampling rate** of 40,000 samples per second is good enough to reproduce an audio signal.

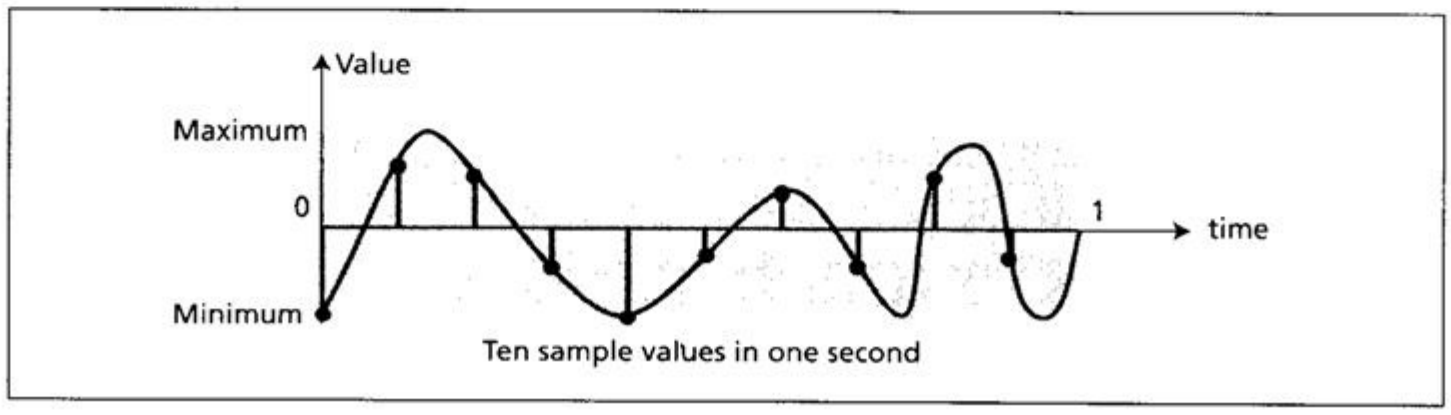


Figure 3.16 Sampling an audio signal

Quantization

The value measured for each sample is a real number. This means that we can store 40,000 real values for each one-second sample. However, it is simpler to use an unsigned integer (a bit pattern) for each sample. **Quantization** refers to a process that rounds the value of a sample to the closest integer value. For example, if the real value is 17.2, it can be rounded down to 17: if the value is 17.7, it can be rounded up to 18.

Encoding

The next task is **encoding**. The quantized sample values need to be encoded as bit patterns. Some systems assign positive and negative values to samples, some just shift the curve to the positive part and assign only positive values. In other words, some systems use an unsigned integer to represent a sample, while others use signed integers to do so. However, the signed integers don't have to be in two's complement, they can be sign-and-magnitude values. The leftmost bit is used to represent the sign (0 for positive values and 1 for negative values), and the rest of the bits are used to represent the absolute values.

The system needs to decide how many bits should be allocated for each sample. Although in the past only eight bits were assigned to sound samples, today 16, 24

or even 32 bits per sample is normal. The number of bits per sample is sometimes referred to as the **bit depth**.

Bit rate

If we call the bit depth or number of bits per sample B, the number of samples per second, S, we need to store S × B bits for each second of audio. This product is sometimes referred to as **bit rate**, R. For example, if we use 40,000 samples per second and 16 bits per each sample, the bit rate is R = 40,000 × 16 = 640,000 bits per second = 640 kilobits per second.

Standards for sound encoding

Today the dominant standard for storing audio is **MP3** (short for *MPEG Layer 3*). This standard is a modification of the **MPEG** (Motion Picture Experts Group) compression method used for video. It uses 44,100 samples per second and 16 bits per sample. The result is a signal with a bit rate of 705,600 bits per second, which is compressed using a compression method that discards information that cannot be detected by the human ear. This is called *lossy* compression, as opposed to lossless compression: see Chapter 15.

3.5 STORING IMAGES

Images are stored in computers using two different techniques: *raster graphics* and *vector graphics*.

Raster graphics

Raster graphics (or **bitmap graphics**) is used when we need to store an analog image such as a photograph. A photograph consists of analog data, similarly to audio information: the difference is that the intensity (color) of data varies in space instead of in time. This means that data must be sampled. However, sampling in this case is normally called **scanning**. The samples are called **pixels** (which stands for *picture elements*). In other words, the whole image is divided into small pixels where each pixel is assumed to have a single intensity value.

Resolution

Just like audio sampling, in image scanning we need to decide how many pixels we need to record for each square or linear inch. The scanning rate in image processing is called **resolution**. If the resolution is sufficiently high, the human eye cannot recognize the discontinuity in reproduced images.

Color depth

The number of bits used to represent a pixel, its **color depth**, depends on how a pixel's color is handled by different encoding techniques. The perception of color is how our eyes respond to a beam of light. Our eyes have different types of *photoreceptor* cells: some respond to the three primary colors red, green, and blue (often called **RGB**), while others merely respond to the intensity of light.

True-Color

One of the techniques used to encode a pixel is called **True-Color**, which uses 24 bits to encode a pixel. In this technique, each of the three primary colors (RGB) are represented by eight bits. Since an 8-bit pattern can represent a number between 0 to 255 in this technique, each color is represented by three decimal numbers between 0 to 255. Table 3.4 shows the three values for some of the colors in this technique.

Table 3.4 Some colors defined in True-Color

Color	*Red*	*Green*	*Blue*	*Color*	*Red*	*Green*	*Blue*
Black	0	0	0	Yellow	255	255	0
Red	255	0	0	Cyan	0	255	255
Green	0	255	0	Magenta	255	0	255
Blue	0	0	255	White	255	255	255

Note that the True-Color scheme can encode 2^{24} or 16,776,216 colors. In other words, the color intensity of each pixel is one of these values.

Indexed color

The True-Color scheme uses more than 16 million colors. Many applications do not need such a large range of colors. The **indexed color**—or **palette color**—scheme uses only a portion of these colors. In this scheme each application selects a few (normally 256) colors from the large set of colors and indexes them, assigning a number between 0 and 255 to each selected color. This is similar to the way in which an artist might have a great many colors in their studio, but at each moment use only a few on their palette. Figure 3.17 illustrates the idea of indexed color.

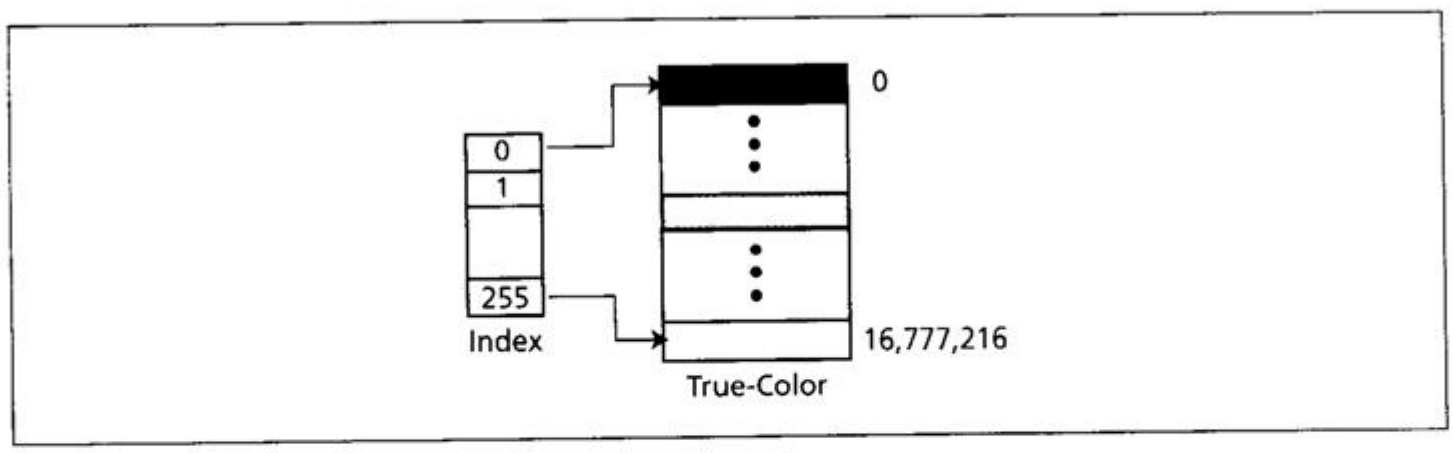


Figure 3.17 Relationship of the indexed color scheme to the True-Color scheme

The use of indexing reduces the number of bits required to store a pixel. For example, in the True-Color scheme 24 bits are needed to store a single pixel. The index color scheme normally uses 256 indexes, which needs only eight bits to store the same pixel. For example, a high-quality digital camera uses almost three million

pixels for a 3 × 5 inch photo. The following shows the number of bits that need to be stored using each scheme:

True-Color:	3,000,000	×	24	=	72,000,000
Indexed-Color:	3,000,000	×	8	=	24,000,000

Standards for image encoding

Several de facto standards for image encoding are in use. **JPEG** (Joint Photographic Experts Group) uses the True-Color scheme, but compresses the image to reduce the number of bits (see Chapter 15). **GIF** (Graphic Interchange Format), on the other hand, uses the indexed color scheme.

Vector graphics

Raster graphics has two disadvantages: the file size is big and rescaling is troublesome. To enlarge a raster graphics image means enlarging the pixels., so the image looks ragged when it is enlarged. The **vector graphic** image encoding method, however, does not store bit patterns for each pixel. An image is decomposed into a combination of geometrical shapes such as lines, squares, or circles. Each geometrical shape is represented by a mathematical formula. For example, a line may be described by the coordinates of its endpoints, and a circle may be described by the coordinates of its center and the length of its radius. A vector graphic image is made up from a series of commands that defines how these shape should be drawn.

When the image is to be displayed or printed, the size of the image is given to the system as an input. The system rescales the image to the new size and uses the same formulae to draw the image. In this case, each time an image is drawn, the formulae are reevaluated. For this reason, vector graphics are also called *geometric modeling* or *object-oriented graphics*.

For example, consider a circle of radius r. The main pieces of information a program needs to draw this circle are:

- The radius r and equation of a circle.
- The location of the center point of the circle.
- The stroke line style and color.
- The fill style and color.

When the size of the circle is changed, the program changes the value of the radius and recalculates the information to draw the circle again. Rescaling does not change the quality of the drawing.

Vector graphics is not suitable for storing the subtleties of photographic images. JPEG or GIF raster graphics provide much better and more vivid pictures. Vector graphics *is* suitable for applications that use mainly geometric primitives to create images. It is used in applications such as FLASH, and to create TrueType (Microsoft, Apple) and PostScript (Adobe) fonts. Computer-aided design (CAD) also uses vector graphics for engineering drawings.

3.6 STORING VIDEO

Video is a representation of images (called *frames*) over time. A movie consists of a series of frames shown one after another to create the illusion of motion. In other words, video is the representation of information that changes in space (single image) and in time (a series of images). So, if we know how to store an image inside a computer, we also know how to store video: each image or frame is transformed into a set of bit patterns and stored. The combination of the images then represents the video. Today video is normally compressed. In Chapter 15 we discuss MPEG, a common video compression technique.

RECOMMENDED READING

[illegible] details about the subjects discussed in this chapter, the following books are recommended:

[illegible]all F: *Multimedia Communication*, Boston, MA: Addison Wesley, 2001

[illegible]ren I: *Computer Arithmetic Algorithms*, Natick, MA: A K Peters, 2001

[illegible]ng B: *Complete Digital Photography*, Hignham, MA: Charles River Media, 200

[illegible]no M: *Computer System Architecture*, Upper Saddle River, NJ: Prentice Hall, 1993

[illegible]no, J: *Compressed Image File Formats*, Boston, MA: Addison Wesley, 1999

KEY TERMS

[illegible]apter has introduced the following key terms, which are listed here with the pages on which **[illegible]st** occur:

[illegible]n National Standards Institute (ANSI) 64

[illegible]g 64

[illegible]it 42

[illegible]th 66

[illegible]66

[illegible]43

[illegible]th 66

[illegible]ng 65

[illegible]23 59

[illegible]point representation 44

[illegible]nterchange Format (GIF) 68

[illegible]hotographer Expert Group (JPEG) 68

[illegible]

American Standard Code for Information Interchange (ASCII) 64

audio 64

bit 42

bit pattern 42

bitmap graphic 66

code 63

digital 64

Excess representation 58

Excess_127 59

floating-point representation 55

indexed color 67

mantissa 58

MPEG 66

normalization 57
overflow 46
pixel 66
raster graphic 66
resolution 66
sampling 65
scanning 66
text 62
truncation error 62
underflow 62
unsigned integer 45
video 69
one's complement 49
palette color 67
quantization 65
real 54
RGB 66
sampling rate 65
sign-and-magnitude representation 47
True-Color 67
two's complement 49
unicode 64
vector graphic 68
word 43

3.9 SUMMARY

- Data comes in different forms, including numbers, text, audio, image, and video. All data types are transformed into a uniform representation called a *bit pattern*.
- A number is changed to the binary system before being stored in computer memory. There are several ways to handle the sign. There are two ways to handle the decimal point: fixed-point and floating-point.
- An integer can be thought of as a number in which the position of the decimal point is fixed: the decimal point is at the right of the least significant bit. An unsigned integer is an integer that can never be negative.
- One of the methods used to store a signed integer is the sign-and-magnitude format. In this format, the leftmost bit is used to show the sign and the rest of the bits define the magnitude. Sign and magnitude are separated from each other.
- Almost all computers use the *two's complement representation* to store a signed integer in an *n*-bit memory location. In this method, the available range for unsigned integers is divided into two equal subranges. The first half is used to represent non-negative integers, the second half is used to represent negative integers. In two's complement representation, the leftmost bit defines the sign of the integer, but sign and magnitude are not separated from each other.
- A *real* is a number with an integral part and a fractional part. Real numbers are stored in the computer using *floating-point representation*. In floating-point representation a number is made up of three sections: a sign, a shifter, and a fixed-point number.
- A piece of text in any language is a sequence of symbols. We can represent each symbol with a bit pattern. Different sets of bit patterns (codes) have been designed to represent text symbols. A coalition of hardware and software manufacturers have designed a code called *Unicode* that uses 32 bits to represent a symbol.
- Audio is a representation of sound or music. Audio is analog data. We cannot record an infinite number of values in an interval, we can only record some samples. The number of samples depends on the maximum number of changes in the analog signal. The values measured for each sample is a real number. *Quantization* refers to a process that rounds up the sample values to integers.
- Storage of images is done using two different techniques: *raster graphics* and *vector graphics*. Raster graphics are used when we need to store an analog image such as a photograph. The image

scanned (sampled) and pixels are stored. In the vector graphic method, an image is decomposed into a combination of geometrical shapes such as lines, squares, or circles. Each geometrical shape is represented by a mathematical formula.

- Video is a representation of images (called *frames*) in time. A movie is a series of frames shown one after another to create the illusion of continuous motion. In other words, video is the representation of information that changes in space (single image) and in time (a series of images).

3.10 PRACTICE SET

Review questions

1. Name five types of data that a computer can process.
2. How is bit pattern length related to the number of symbols the bit pattern can represent?
3. How does the bitmap graphic method represent an image as a bit pattern?
4. What is the advantage of the vector graphic method over the bitmap graphic method? What is the disadvantage?
5. What steps are needed to convert audio data to bit patterns?
6. Compare and contrast the representation of positive integers in unsigned, sign-and-magnitude format, and two's complement format.
7. Compare and contrast the representation of negative integers in sign-and-magnitude and two's complement format.
8. Compare and contrast the representation of zero in sign-and-magnitude, two's complement, and Excess formats.
9. Discuss the role of the leftmost bit in sign-and-magnitude, and two's complement, formats.
10. Answer the following questions about floating-point representations of real numbers:
    a. Why is normalization necessary?
    b. What is the mantissa?
    c. After a number is normalized, what kind of information does a computer store in memory?

Multiple-choice questions

11. A byte consists of _____ bits.
    a. 2
    b. 4
    c. 8
    d. 16
12. In a set of 64 symbols, each symbol requires a bit pattern length of _____ bits.
    a. 4
    b. 5
    c. 6
    d. 7
13. How many symbols can be represented by a bit pattern with ten bits?
    a. 128
    b. 256
    c. 512
    d. 1024
14. If the ASCII code for E is 1000101, then the ASCII code for e is _____. Answer the question without consulting the ASCII table.
    a. 1000110
    b. 1000111
    c. 0000110
    d. 1100101
15. A 32-bit code called _____ represents symbols in all languages.
    a. ANSI
    b. Unicode
    c. EBCDIC
    d. Extended ASCII
16. An image can be represented in a computer using the _____ method.
    a. bitmap graphic
    b. vector graphic
    c. Excess system
    d. a or b

17. In the _____ graphic method of representing an image in a computer, each pixel is assigned a bit pattern.
    a. bitmap
    b. vector
    c. quantized
    d. binary
18. In the _____ graphic method of representing an image in a computer, the image is decomposed into a combination of geometrical figures.
    a. bitmap
    b. vector
    c. quantized
    d. binary
19. In the _____ graphic method of representing an image in a computer, rescaling of the image creates a ragged or grainy image.
    a. bitmap
    b. vector
    c. quantized
    d. binary
20. When we want to store music in a computer, the audio signal must be _____.
    a. sampled
    b. quantized
    c. coded
    d. all of the above
21. If the leftmost bit in _____ number representation is zero, then the decimal number is nonnegative.
    a. two's complement
    b. floating point
    c. Excess system
    d. both a and b
22. If the leftmost bit in _____ number representation is 1, then the decimal number is negative.
    a. two's complement
    b. floating point
    c. Excess system
    d. both a and b
23. Which number representation method is often used to store the exponential value of a fractional part?
    a. unsigned integers
    b. two's complement
    c. Excess system
    d. none of the above
24. In an Excess conversion, we _____ the bias number to the number to be converted.
    a. add
    b. subtract
    c. multiply
    d. divide
25. When a fractional part is normalized, the computer stores the _____.
    a. sign
    b. exponent
    c. mantissa
    d. all of the above
26. The precision of the fractional part of a number stored in a computer is defined by the _____.
    a. sign
    b. exponent
    c. mantissa
    d. any of the above
27. The combination of sign and mantissa of a real number in IEEE standard floating point format is stored as an integer in the _____ representation
    a. unsigned
    b. sign-and-magnitude
    c. two's complement
    d. none of the above

Exercises

28. How many distinct 5-bit patterns can we have?
29. In some countries vehicle license plates have two decimal digits (0 to 9). How many distinct plates can we have? If the digit 0 is not allowed on the license plate, how many distinct plates can we have?
30. Redo Exercise 29 for a license plate that has two digits followed by three uppercase letters (A to Z).
31. A machine has eight different cycles. How many bits are needed to represent each cycle?
32. A student's grade in a course can be A, B, C, D, F, W (withdraw), or I (incomplete). How many bits are needed to represent the grade?

A company has decided to assign a unique bit pattern to each employee. If the company has 500 employees, what is the minimum number of bits needed to create this system of representation? How many patterns are unassigned? If the company hires another 300 employees, should it increase the number of bits? Explain your answer.

If we use a 4-bit pattern to represent the digits 0 to 9, how many bit patterns are wasted?

An audio signal is sampled 8,000 times per second. Each sample is represented by 256 different levels. How many bits per second are needed to represent this signal?

Change the following decimal numbers to 8-bit unsigned integers.

a. 23
b. 121
c. 34
d. 342

Change the following decimal numbers to 16-bit unsigned integers.

a. 41
b. 411
c. 1234
d. 342

Change the following decimal numbers to 8-bit two's complement integers.

a. −12
b. −145
c. 56
d. 142

Change the following decimal numbers to 16-bit two's complement integers.

a. 102
b. −179
c. 534
d. 62,056

Change the following 8-bit unsigned numbers to decimal.

a. 01101011
b. 10010100
c. 00000110
d. 01010000

41. Change the following 8-bit two's complement numbers to decimal.
    a. 01110111
    b. 11111100
    c. 01110100
    d. 11001110
42. The following are two's complement binary numbers. Show how to change the sign of the number.
    a. 01110111
    b. 11111100
    c. 01110111
    d. 11001110
43. If we apply the two's complement operation to a number twice, we should get the original number. Apply the two's complement operation to each of the following numbers and see if we can get the original number.
    a. 01110111
    b. 11111100
    c. 01110100
    d. 11001110
44. Normalize the following binary floating point numbers. Explicitly show the value of the exponent after normalization.
    a. 1.10001
    b. $2^3 \times 111.1111$
    c. $2^{-2} \times 101.110011$
    d. $2^{-5} \times 101101.00000110011000$
45. Convert the following numbers in 32-bit IEEE format.
    a. $-2^0 \times 1.10001$
    b. $+2^3 \times 1.111111$
    c. $+2^{-4} \times 1.01110011$
    d. $-2^{-5} \times 1.01101000$
46. Convert the following numbers in 64-bit IEEE format:
    a. $-2^0 \times 1.10001$
    b. $+2^3 \times 1.111111$
    c. $+2^{-4} \times 1.01110011$
    d. $-2^{-5} \times 1.01101000$

47. Convert the following numbers in 32-bit IEEE format.
    a. 7.1875
    b. −12.640625
    c. 11.40625
    d. −0.375
48. The following are sign-and-magnitude binary numbers in an 8-bit allocation. Convert them to decimal.
    a. 01110111
    b. 11111100
    c. 01110100
    d. 11001110
49. Convert the following decimal integers to sign-and-magnitude with 8-bit allocation.
    a. 53
    b. −107
    c. −5
    d. 154
50. One method of representing signed numbers in a computer is one's complement representation. In this representation, to represent a positive number, we store the binary number. To represent a negative number, we apply the one's complement operation to the number. Store the following decimal integers to one's complement with 8-bit allocation.
    a. 53
    b. −107
    c. −5
    d. 154
51. The following are one's complement binary numbers in an 8-bit allocation. Convert them to decimal (see Exercise 50).
    a. 01110111
    b. 11111100
    c. 01110100
    d. 11001110
52. If we apply the one's complement operation to a number twice, we should get the original number. Apply the one's complement operation twice to each of the following numbers and see if you can get the original number (see Exercise 50).
    a. 01110111
    b. 11111100
    c. 01110100
    d. 11001110
53. An alternative method to find the two's complement of a number is to first take the one's complement of the number (see Exercise 50) and then add 1 to the result. (Adding binary integers is explained in Chapter 4). Try both methods using the following numbers. Compare and contrast the results.
    a. 01110111
    b. 11111100
    c. 01110100
    d. 11001110
54. The equivalent of one's complement (see Exercise 50) in the binary system is nine's complement in the decimal system (1 = 2 − 1 and 9 = 10 − 1). With n-digit allocation, we can represent nine's complement numbers in the range of:

$$-[(10^n/2) - 1] \quad \text{to} \quad +[(10^n/2 - 1)]$$

The nine's complement of a number with n digit allocation is obtained as follows. If the number is positive, the nine's complement of the number is itself. If the number is negative, we subtract each digit from 9. Answer the following questions for three-digit allocation:
    a. What is the range of the numbers we can represent using nine's complement?
    b. In this system, how can we determine the sign of a number?
    c. Do we have two zeros in this system?
    d. If the answer to c. is yes, what is the representation for +0 and −0?
55. Assuming three-digit allocation, find the nine's complement of the following decimal numbers (see Exercise 54):
    a. +234
    b. +560
    c. −125
    d. −111
56. The equivalent of two's complement in the binary system is ten's complement in the decimal system (in the binary system, 2 is the base, in the decimal system, 10 is the base). Using

n-digit allocation, we can represent numbers in the range of:

$$-(10^n/2) \quad \text{to} \quad +(10^n/2-1)$$

in ten's complement format. The ten's complement of a number with n-digit allocation is obtained by first finding the nine's complement of the number (as described in Exercise 54) and then adding 1 to the result. Answer the following questions for three-digit allocation.

a. What is the range of the numbers we can represent using ten's complement?

b. In this system, how can we determine the sign of a number?

c. Do we have two zeros in this system?

d. If the answer to c. is yes, what is the representation for +0 and −0?

Assuming three-digit allocation, find the ten's complement of the following decimal numbers. (The information for doing this problem is presented in Exercise 56).

a. +234

b. +560

c. −125

d. −111

The equivalent of one's complement (Exercise 50) in the binary system is fifteen's complement in the hexadecimal system (1 = 2 − 1 and 15 = 16 − 1). Read the explanation provided for Exercise 54 to answer the following questions:

a. What range of numbers can we represent with three-digit allocation in fifteen's complement?

b. Explain how the fifteen's complement of a number is obtained in the hexadecimal system.

c. Do we have two zeros in this system?

d. If the answer to c. is yes, what is the representation for +0 and −0?

59. Assuming three-digit allocation, find the fifteen's complement of the following hexadecimal numbers (see Exercise 58):

a. +B14

b. +FE1

c. −1A

d. −1E2

60. The equivalent of two's complement in the binary system is sixteen's complement in the hexadecimal system. Read the explanation provided for Exercise 56 to answer the following questions.

a. What range of numbers can we represent with three-digit allocation in sixteen's complement?

b. Explain how a sixteen's complement of a number is obtained in the hexadecimal system.

c. Do we have two zeros in this system?

d. If the answer to c. is yes, what is the representation for +0 and −0?

61. Assuming three-digit allocation, find the sixteen's complement of the following hexadecimal numbers (see Exercise 60):

a. +B14

b. +FE1

c. −1A

d. −1E2

4.1 LOGIC OPERATIONS

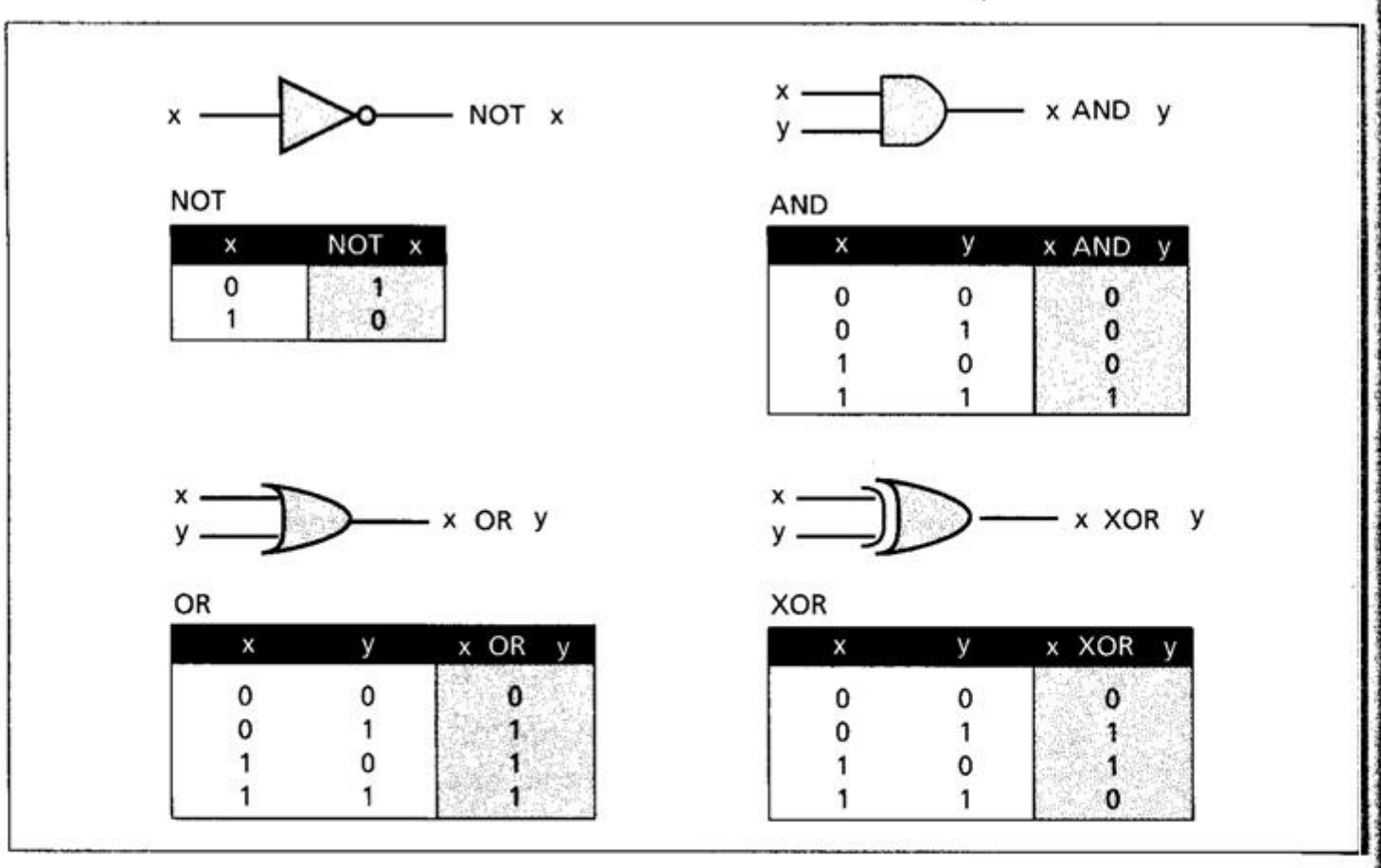
In Chapter 3 we discussed the fact that data inside a computer is stored as patterns of bits. **Logic operations** refer to those operations that apply the same basic operation on individual bits of a pattern, or on two corresponding bits in two patterns. This means that we can define logic operations at the bit level and at the pattern level. A logic operation at the pattern level is *n* logic operations, of the same type, at the bit level where *n* is the number of bits in the pattern.

Logic operations at bit level

A bit can take one of the two values: 0 or 1. If we interpret 0 as the value *false* and 1 as the value *true*, we can apply the operations defined in **Boolean algebra** to manipulate bits. Boolean algebra, named in honor of George Boole, belongs to a special field of mathematics called *logic*. Boolean algebra and its application to building logic circuits in computers are briefly discussed in Appendix E. In this section, we show briefly four bit-level operations that are used to manipulate bits: NOT, AND, OR, and XOR.

Boolean algebra and logic circuits are discussed in Appendix E.

Figure 4.1 shows the symbols for these four bit-level operators and their truth tables. A **truth table** defines the values of the output for each possible input or inputs. Note that the output of each operator is always one bit, but the input can be one or two bits.



NOT

x	NOT x
0	1
1	0

AND

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.1 Logic operations at the bit level

The **NOT operator** is a unary operator: it takes only one input. The output bit is the complement of the input. If the input is 0, the output is 1, if the input is 1, the output is 0. In other words, the NOT operator flips its input. The truth table of the NOT operator has only two rows because the single input can be either 0 or 1: two possibilities.

The **AND operator** is a binary operator: it takes two inputs. The output bit is 1 if both inputs are 1s and the output is 0 in the other three cases. The truth table of the AND operator has four rows because, with two inputs, there are four possible input combinations.

A property

One interesting point about the AND operator is that if a bit in one input is 0, we do not have to check the corresponding bit in the other input: we can quickly conclude that the result is 0. We use this property when we discuss the application of this operator in relation to a bit pattern.

For $x = 0$ or 1 $\quad x$ AND 0 $\rightarrow$ 0 $\quad$ and $\quad$ 0 AND $x \rightarrow 0$

The **OR operator** is a also a binary operator: it takes two inputs. The output bit is 0 if both inputs are 0s and the output is 1 in the other three cases. The truth table of the OR operator has also four rows. The OR operator is sometimes called the *inclusive-or operator* because the output is 1 not only when one of the inputs is 1, but also when both inputs are 1s. This is in contrast to the operator we introduce next.

A property

One interesting point about the OR operator is that if a bit in one input is 1, we do not have to check the corresponding bit in the other input: we can quickly conclude that the result is 1. We use this property when we discuss the application of this operator in relation to a bit pattern.

For $x = 0$ or 1 $\quad x$ OR 1 $\rightarrow$ 1 $\quad$ and $\quad$ 1 OR $x \rightarrow 1$

The **XOR operator** (pronounced "exclusive-or") is also a binary operator like the OR operator, with only one difference: the output is 0 if both inputs are 1s. We can look at this operator in another way: the output is 0 when both inputs are the same, and the output is 1 when the inputs are different.

Example 4.1

In English we use the conjunction "or" sometimes to means an inclusive-or, and sometimes to means an exclusive-or.

a. The sentence "I wish to have a car *or* a house" uses "or" in the inclusive sense—I wish to have a car, a house, or both.

b. The sentence "Today is either Monday or Tuesday" uses "or" in the exclusive sense—today is either Monday or Tuesday, but it cannot be both.

Example 4.2

The XOR operator is not actually a new operator. We can always simulate it using the other three operators. The following two expressions are equivalent

x XOR y ↔ [x AND (NOT y)] OR [(NOT x) AND y]

The equivalence can be proved if we make the truth table for both.

A property

A property of XOR is that if a bit in one input is 1, the result is the complement of the corresponding bit in the other input. We use this property when we discuss the application of this operator in relation to a bit pattern.

For x = 0 or 1 1 XOR x → NOT x *and* x XOR 1 → NOT x

Logic operations at pattern level

The same four operators (NOT, AND, OR, and XOR) can be applied to an n-bit pattern. The effect is the same as applying each operator to each individual bit for NOT and to each corresponding pair of bits for the other three operators. Figure 4.2 shows these four operators with input and output patterns.

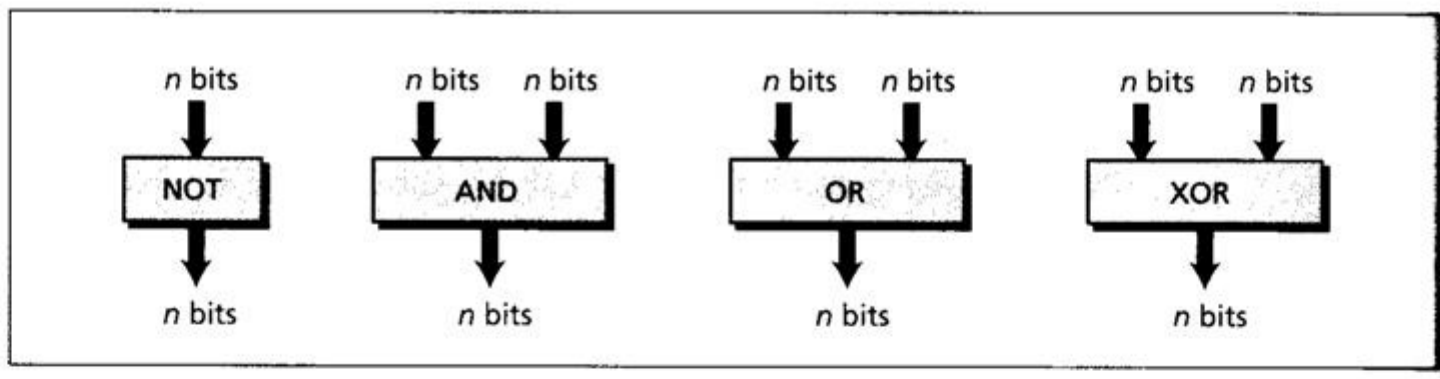


Figure 4.2 Logic operators applied to bit patterns

Example 4.3

Use the NOT operator on the bit pattern 10011000.

Solution

The solution is shown below. Note that the NOT operator changes every 0 to 1 and every 1 to 0.

NOT	1	0	0	1	1	0	0	0	Input
	0	1	1	0	0	1	1	1	Output

Example 4.4

Use the AND operator on the bit patterns 10011000 and 00101010.

Solution

The solution is shown below. Note that only one bit in the output is 1, where both corresponding inputs are 1s.

	1	0	0	1	1	0	0	0	Input 1
AND	0	0	1	0	1	0	1	0	Input 2
	0	0	0	0	1	0	0	0	Output

Example 4.5

Use the OR operator on the bit patterns 10011001 and 00101110.

Solution

The solution is shown below. Note that only one bit in the output is 0, where both corresponding inputs are 0s.

	1	0	0	1	1	0	0	1	Input 1
OR	0	0	1	0	1	1	1	0	Input 2
	1	0	1	1	1	1	1	1	Output

Example 4.6

Use the XOR operator on the bit patterns 10011001 and 00101110.

Solution

The solution is shown below. Compare the output in this example with the one in Example 4.5. The only difference is that when the two inputs are 1s, the result is 0 (the effect of exclusion).

	1	0	0	1	1	0	0	1	Input 1
XOR	0	0	1	0	1	1	1	0	Input 2
	1	0	1	1	0	1	1	1	Output

ons Four logic operations can be used to modify a bit pattern.

Complementing

The only application of the NOT operator is to complement the whole pattern. Applying this operator to a pattern changes every 0 to 1 and every 1 to 0. This is sometimes referred to as a one's complement operation. Example 4.3 shows the effect of complementing.

Unsetting specific bits

One of the applications of the AND operator is to **unset** (force to 0) specific bits in a bit pattern. The second input in this case is called a **mask**. The 0-bits in the mask unset the corresponding bits in the first input: the 1-bits in the mask leave the corresponding bits in the first input unchanged. This is due to the property we mentioned for the AND operator: if one of the inputs is 0, the output is 0 no matter

what the other input is. Unsetting the bits in a pattern has many applications. For example, if an image uses only one bit per pixel (a black and white image), then we can make a specific pixel black using a mask and the AND operator.

Example 4.7

Use a mask to unset (clear) the five leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

The mask is 00000111. The result of applying the mask is:

	1	0	1	0	0	1	1	0	Input
AND	0	0	0	0	0	1	1	1	Mask
	0	0	0	0	0	1	1	0	Output

Note that the three rightmost bits remain unchanged, while the five leftmost bits are unset (changed to 0) no matter what their previous values.

Setting specific bits

One of the applications of the OR operator is to **set** (force to 1) specific bits in a bit pattern. Again we can use a mask, but a different one. The 1-bits in the mask **set** the corresponding bits in the first input, and the 0-bits in the mask leave the corresponding bits in the first input unchanged. This is due to the property we mentioned for the OR operator: if one of the inputs is 1, the output is 1 no matter what the other input is. Setting the bits in a pattern has many applications. For example, if an image uses only one bit per pixel (a black and white image), then we can make a specific pixel white using a mask and the OR operator.

Example 4.8

Use a mask to set the five leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

The mask is 11111000. The result of applying the mask is:

	1	0	1	0	0	1	1	0	Input
OR	1	1	1	1	1	0	0	0	Mask
	1	1	1	1	1	1	1	0	Output

Flipping specific bits

One of the applications of the XOR operator is to flip (complement) specific bits in a bit pattern. Again we can use a mask, but a different one. The 1-bits in the mask flip the corresponding bits in the first input, and the 0-bits in the mask leave the corresponding bits in the first input unchanged. This is due to the property we mentioned for the XOR operator: if one of the inputs is 1, the output is the complement of the corresponding bit. Note the difference between the NOT operator

and the XOR operator. The NOT operator complements all the bits in the input, while the XOR operator complements only the specific bits in the first input as defined by the mask.

Example 4.9

Use a mask to flip the five leftmost bits of a pattern. Test the mask with the pattern 10100110.

Solution

The mask is 11111000. The result of applying the mask is:

	1	0	1	0	0	1	1	0	Input
XOR	1	1	1	1	1	0	0	0	Mask
	0	1	0	1	1	1	1	0	Output

4.2 SHIFT OPERATIONS

Shift operations move the bits in a pattern, changing the positions of the bits. They can move bits to the left or to the right. We can divide shift operations into two categories: logical shift operations and arithmetic shift operations.

ical shift rations

A **logical shift operation** is applied to a pattern that does not represent a signed number. The reason is that these shift operations may change the sign of the number that is defined by the leftmost bit in the pattern. We distinguishes two types of logical shift operations, as described below.

Logical shift

A logical right shift operation shifts each bit one position to the right. In an n-bit pattern, the rightmost bit is lost and a 0 fills the leftmost bit. A logical left shift operation shifts each bit one position to the left. In an n-bit pattern, the leftmost bit is lost and a 0 fills the rightmost bit. Figure 4.3 shows the logical right shift and logical left shift operations for an 8-bit pattern.

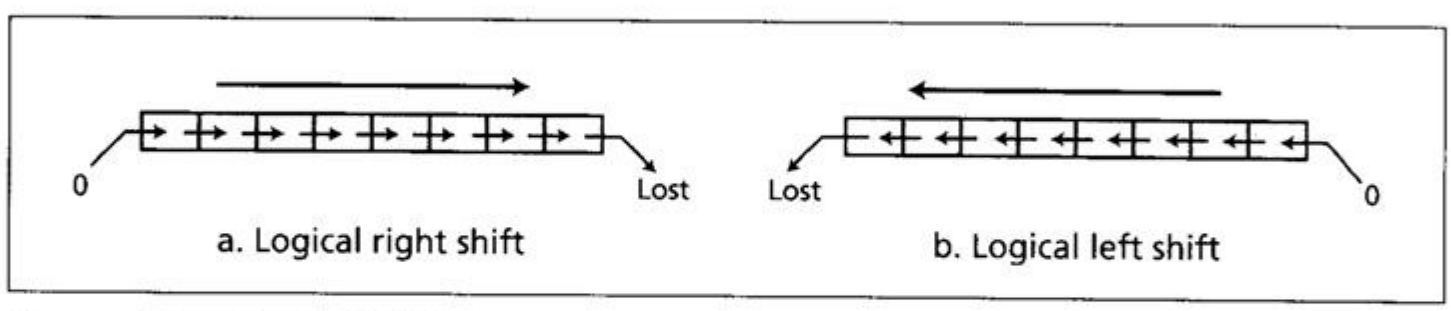


Figure 4.3 Logical shift operations

Example 4.10

Use a logical left shift operation on the bit pattern 10011000.

Solution

The solution is shown below. The leftmost bit is lost and a 0 is inserted as the rightmost bit.

←	1	0	0	1	1	0	0	0	Original
	0	0	1	1	0	0	0	0	After shift

Circular shift

A **circular shift operation** (or **rotate operation**) shifts bits, but no bit is lost or added. A circular right shift (or *right rotate*) shifts each bit one position to the right. The rightmost bit is circulated and becomes the leftmost bit. A circular left shift (or *left rotate*) shifts each bit one position to the left. The leftmost bit circulates and become the rightmost bit. Figure 4.4 shows the circular shift left and circular shift right operations.

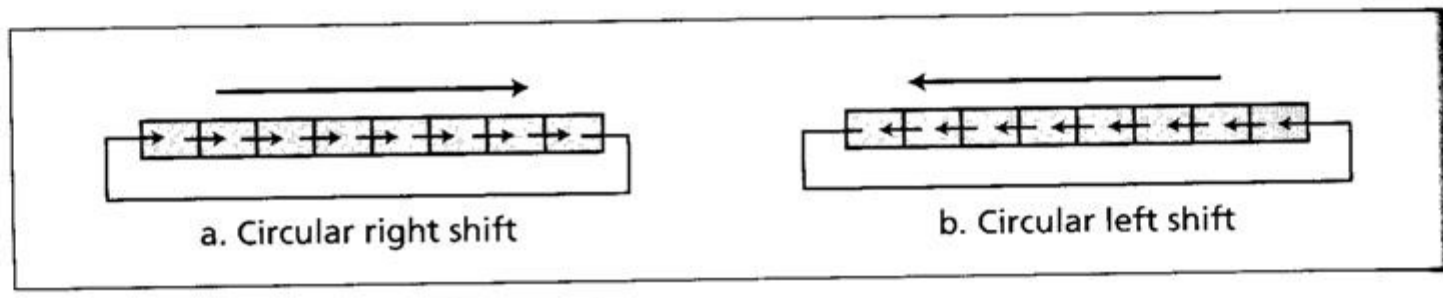


Figure 4.4 Circular shift operations

Example 4.11

Use a circular left shift operation on the bit pattern 10011000.

Solution

The solution is shown below. The leftmost bit is circulated and becomes the rightmost bit.

1	0	0	1	1	0	0	0	Original
0	0	1	1	0	0	0	1	After shift

Arithmetic shift operations

Arithmetic shift operations assume that the bit pattern is a signed integer in two's complement format. Arithmetic right shift is used to divide an integer by two, while arithmetic left shift is used to multiply an integer by two (discussed later). These operations should not change the sign (leftmost) bit. An arithmetic right shift retains the sign bit, but also copies it into the next right bit, so that the sign is preserved. An arithmetic left shift discards the sign bit and accepts the bit to the left of the sign bit as the sign. If the new sign bit is the same as the previous one, the operation is successful, otherwise an overflow or underflow has occurred and the result is not valid. Figure 4.5 shows these two operations.

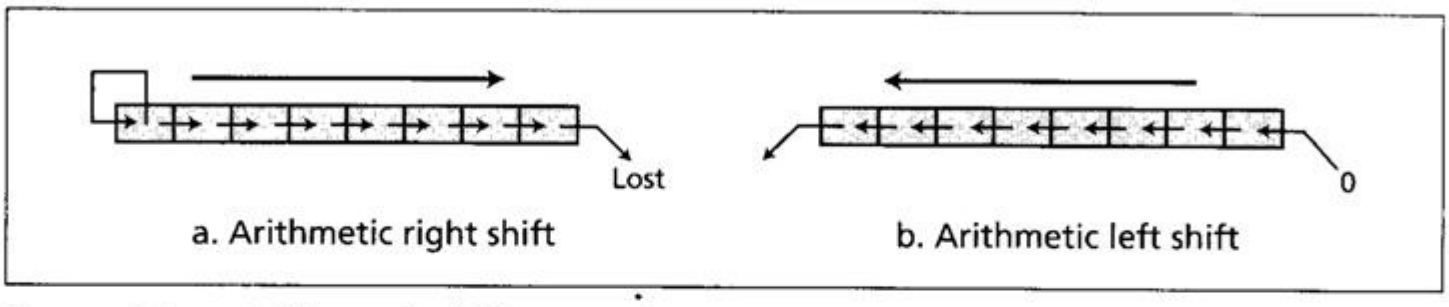


Figure 4.5 Arithmetic shift operations

Example 4.12

Use an arithmetic right shift operation on the bit pattern 10011001. The pattern is an integer in two's complement format.

Solution

The solution is shown below. The leftmost bit is retained and also copied to its right neighbor bit.

1	0	0	1	1	0	0	1	Original
1	1	0	0	1	1	0	0	After shift

The original number was −103 and the new number is −52, which is the result of dividing −103 by 2 truncated to the smaller integer.

Example 4.13

Use an arithmetic left shift operation on the bit pattern 11011001. The pattern is an integer in two's complement format.

Solution

The solution is shown below. The leftmost bit is lost and a 0 is inserted as the rightmost bit.

1	1	0	1	1	0	0	1	Original
1	0	1	1	0	0	1	**0**	After shift

The original number was −39 and the new number is −78. The original number is multiplied by two. The operation is valid because no underflow occurred.

Example 4.14

Use an arithmetic left shift operation on the bit pattern 01111111. The pattern is an integer in two's complement format.

Solution

The solution is shown below. The leftmost bit is lost and a 0 is inserted as the rightmost bit.

0	1	1	1	1	1	1	1	Original
1	1	1	1	1	1	1	**0**	After shift

The original number was 127 and the new number is –2. Here the result is not valid because an overflow has occurred. The expected answer 127 × 2 = 254 cannot be represented by an 8-bit pattern.

Example 4.15

Combining logic operations and logical shift operations gives us some tools for manipulating bit patterns. Assume that we have a pattern and we need to use the third bit (from the right) of this pattern in a decision-making process. We want to know if this particular bit is 0 or 1. The following shows how we can find out.

	h	g	f	e	d	c	b	a	Original
	0	h	g	f	e	d	c	b	One right shift
	0	0	h	g	f	e	d	c	Two right shifts
AND	0	0	0	0	0	0	0	1	Mask
	0	0	0	0	0	0	0	c	Result

We shift the pattern two bits to the right so that the target bit moves to the rightmost position. The result is then ANDed with a mask which has one 1 at the leftmost position. The result is a pattern with seven 0s and the target bit at the rightmost position. We can then test the result: if it is an unsigned integer 1, the target bit was 1, whereas if the result is an unsigned integer 0, the target bit was 0.

4.3 ARITHMETIC OPERATIONS

Arithmetic operations involve adding, subtracting, multiplying, and dividing. We can apply these operations to integers and floating-point numbers.

Arithmetic operations on integers

All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to integers. Although multiplication (division) of integers can be implemented using repeated addition (subtraction), the procedure is not efficient. There are more efficient procedures for multiplication and division, such as Booth procedures, but these are beyond the scope of this book. For this reason, we only discuss addition and subtraction of integers here.

Addition and subtraction for two's complement integers

We first discuss addition and subtraction for integers in two's complement representation, because it is easier. As we discussed in Chapter 3, integers are normally stored in two's complement format. One of the advantages of two's complement representation is that there is no difference between addition and subtraction. When the subtraction operation is encountered, the computer simply changes it to an addition operation, but makes two's complement of the second number. In other words:

$$A - B \leftrightarrow A + (\overline{B} + 1) \quad \text{where } ((\overline{B} + 1)) \text{ means the two's complement of B}$$

This means that we only need to discuss addition. Adding numbers in two's complement is like adding the numbers in decimal: we add column by column, and if there is a carry, it is added to the next column. However, the carry produced from the last column is discarded.

We should remember that we add integers column by column. In each column, we have either two bits to add if there is no carry from the previous column, or three bits to add if there is a carry from the previous column. The number of 1s in each column can be zero, one, two, or three. The following table shows the sum and carry (C).

Table 4.1 Carry and sum resulting from adding two bits

Column	*Carry*	*Sum*	*Column*	*Carry*	*Sum*
Zero 1s	0	0	Two 1s	1	0
One 1	0	1	Three 1s	1	1

Now we can show the procedure for addition or subtraction of two integers in two's complement format (Figure 4.6). Note that we use the notation $(\overline{X} + 1)$ to mean two's complement of X. This notation is very common in literature because $\overline{X}$ denotes the one's complement of X. If we add 1 to the one's complement of an integer, we get its two's complement.

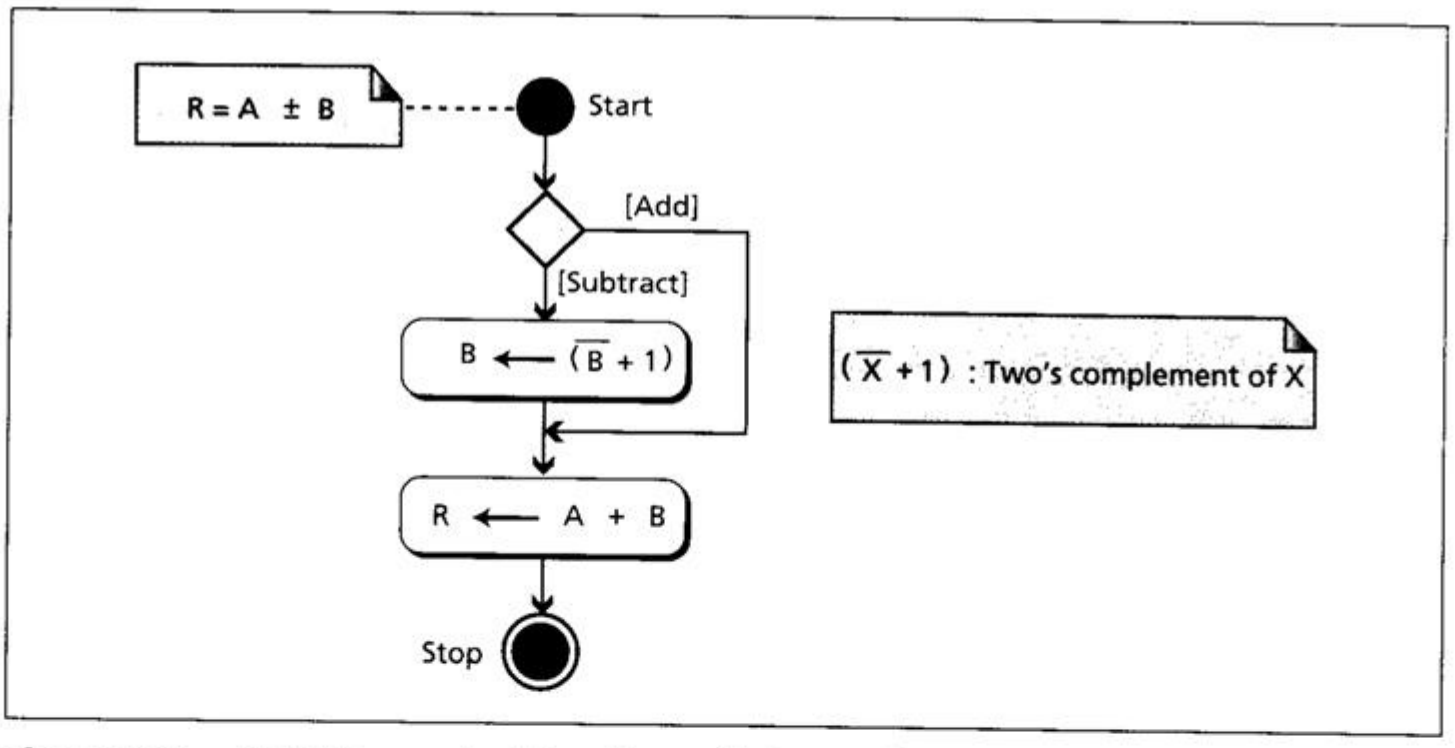


Figure 4.6 Addition and subtraction of integers in two's complement format

The procedure is as follows:

1. If the operation is subtraction, we take the two's complement of the second integer. Otherwise, we move to the next step.
2. We add the two integers.

Example 4.16

Two integers A and B are stored in two's complement format. Show how B is added to A.

$$A = (00010001)_2 \quad B = (00010110)_2$$

Solution

The operation is adding. A is added to B and the result is stored in R.

			1						Carry
	0	0	0	1	0	0	0	1	A
+	0	0	0	1	0	1	1	0	B
	0	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**R**

We check the result in decimal: (+17) + (+22) = (+39).

Example 4.17

Two integers A and B are stored in two's complement format. Show how B is added to A.

$$A = (00011000)_2 \quad B = (11101111)_2$$

Solution

The operation is adding. A is added to B and the result is stored in R.

	1	1	1	1	1				Carry
	0	0	0	1	1	0	0	0	A
+	1	1	1	0	1	1	1	1	B
	0	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**R**

Checking the result in decimal, (+24) + (−17) = (+7).

Example 4.18

Two integers A and B are stored in two's complement format. Show how B is subtracted from A.

$$A = (00011000)_2 \quad B = (11101111)_2$$

Solution

The operation is subtracting. A is added to $(\overline{B} + 1)$ and the result is stored in R.

			1						Carry
	0	0	0	1	1	0	0	0	A
+	0	0	0	1	0	0	0	1	$(\overline{B} + 1)$
	0	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**R**

Checking the result in decimal, (+24) − (−17) = (+41).

Example 4.19

Two integers A and B are stored in two's complement format. Show how B is subtracted from A.

$$A = (11011101)_2 \quad B = (00010100)_2$$

Solution

The operation is subtracting. A is added to $(\overline{B} + 1)$ and the result is stored in R.

1	1	1	1	1	1				Carry
	1	1	0	1	1	1	0	1	A
+	1	1	1	0	1	1	0	0	$(\overline{B} + 1)$
	1	1	0	0	1	0	0	1	R

Note that the last carry is discarded. Checking the result in decimal, (–35) – (+20) = (–55).

Example 4.20

Two integers A and B are stored in two's complement format. Show how B is added to A.

$$A = (01111111)_2 \quad B = (00000011)_2$$

Solution

The operation is adding. A is added to B and the result is stored in R.

	1	1	1	1	1	1	1		Carry
	0	1	1	1	1	1	1	1	A
+	0	0	0	0	0	0	1	1	B
	1	0	0	0	0	0	1	0	R

We expect the result to be 127 + 3 = 130, but the answer is –126. The error is due to overflow, because the expected answer (+130) is not in the range –128 to +127.

When we do arithmetic operations on numbers in a computer, we should remember that each number and the result should be in the range defined by the bit allocation.

Addition and subtraction for integers in sign-and-magnitude representation looks very complex. We have four different combination of signs (two signs, each of two values) for addition, and four different conditions for subtraction. This means that we need to consider eight different situations. However, if we first check the signs, we can reduce these cases, as shown in Figure 4.7.

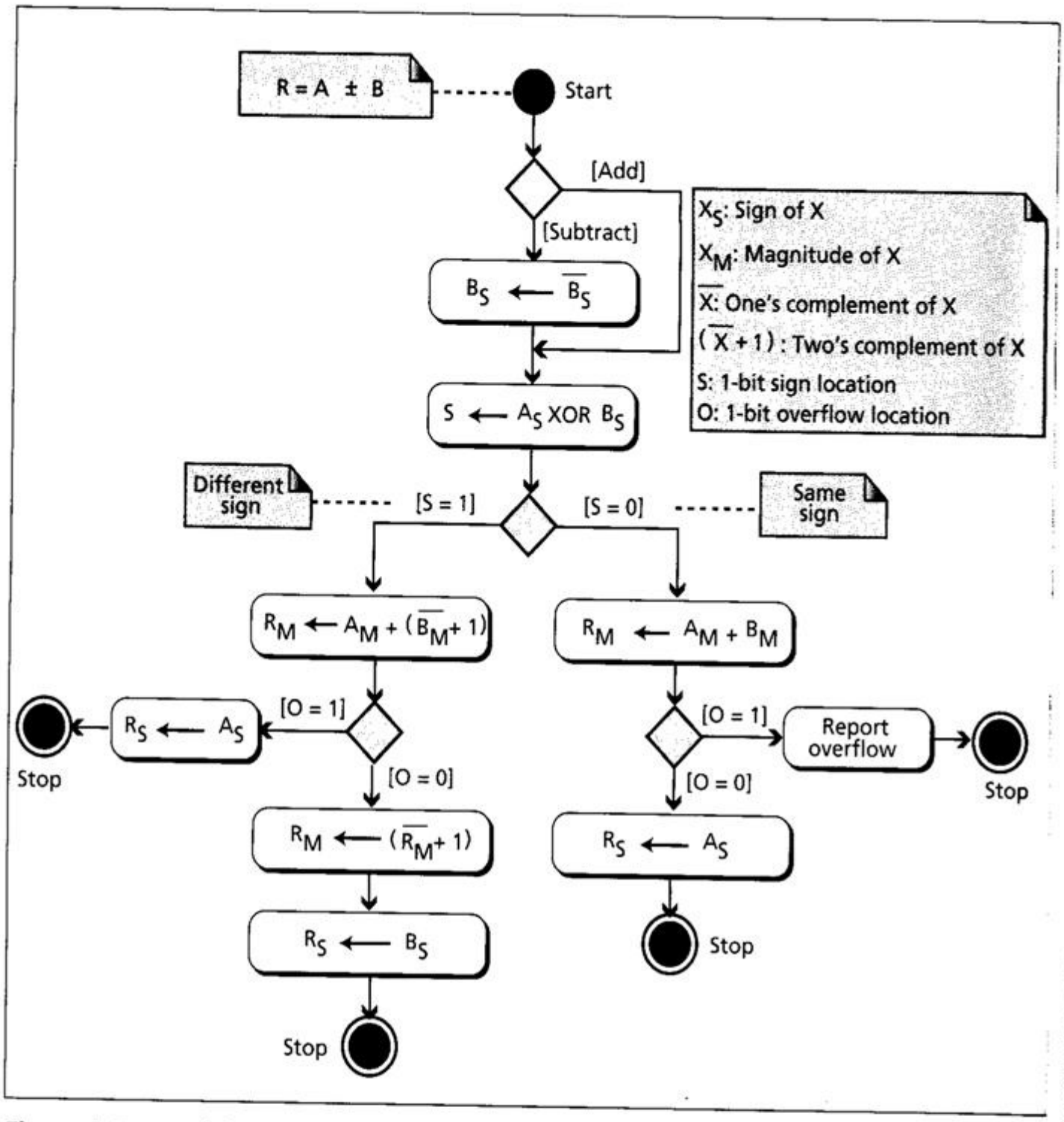


Figure 4.7 Addition and subtraction of integers in sign-and-magnitude format

Let us first explain the diagram:

1. We check the operation. If the operation is subtraction, we change the sign of the second integer (B). This means we now only have to worry about addition of two signed integers.
2. We apply the XOR operation to the two signs. If the result (stored in temporary location S) is 0, it means that the signs are the same (either both signs are positive or both are negative).
3. If the signs are the same, $R = \pm (A_M + B_M)$. We need to add the magnitude and the sign of the result is the common sign. So, we have:

$$R_M = (A_M) + (B_M) \quad \text{and} \quad R_S = A_S$$

where the subscript M means magnitude and subscript S means sign. In this case, however, we should be careful about the overflow. When we add the two magnitudes, an overflow may occur that must be reported and the process aborted.

4. If the signs are different, $R = \pm (A_M - B_M)$. So we need to subtract B_M from A_M and then make a decision about the sign. Instead of subtracting bit by bit, we take the two's complement of the second magnitude (B_M) and add them. The sign of the result is the sign.of the integer with larger magnitude.
   a. It can be shown that if $A_M \geq B_M$, there is an overflow and the result is a positive number. Therefore, if there is an overflow, we discard the overflow and let the sign of the result be the sign of A.
   b. It can be shown that if $A_M < B_M$, there is no overflow, but the result is a negative number. So if there is no overflow, we make the two's complement of the result and let the sign of the result be the sign of B.

Example 4.21

Two integers A and B are stored in sign-and-magnitude format (we have separated the sign from the magnitude for clarity). Show how B is added to A.

$$A = (0\ 0010001)_2 \quad B = (0\ 0010110)_2$$

Solution

The operation is adding: the sign of B is not changed. Since $S = A_S \text{ XOR } B_S = 0$, $R_M = A_M + B_M$ and $R_S = A_S$. There is no overflow.

		No overflow		1						Carry
A_S	0			0	0	1	0	0	0	1 A_M
B_S	0		+	0	0	1	0	1	1	0 B_M
R_S	0			0	1	0	0	1	1	1 R_M

Checking the result in decimal, (+17) + (+22) = (+39).

Example 4.22

Two integers A and B are stored in sign-and-magnitude format. Show how B is added to A.

$$A = (0\ 0010001)_2 \quad B = (1\ 0010110)_2$$

Solution

The operation is adding: the sign of B is not changed. $S = A_S \text{ XOR } B_S = 1$; $R_M = A_M + (\overline{B}_M + 1)$. Since there is no overflow, we need to take the two's complement of R_M. The sign of R is the sign of B.

		No overflow								Carry
A_S	0			0	0	1	0	0	0	1 A_M
B_S	1		+	1	1	0	1	0	1	0 $(\overline{B}_M + 1)$
				1	1	1	1	0	1	1 R_M
R_S	1			0	0	0	0	1	0	1 $R_M = (\overline{R}_M + 1)$

Checking the result in decimal, (+17) + (−22) = (−5).

Example 4.23

Two integers A and B are stored in sign-and-magnitude format. Show how B is subtracted from A.

$$A = (1\ 1010001)_2 \quad B = (1\ 0010110)_2$$

Solution

The operation is subtracting: $B_S = \overline{B}_S$. $S = A_S$ XOR $B_S = 1$, $R_M = A_M + (\overline{B}_M + 1)$. Since there is an overflow, the value of R_M is final. The sign of R is the sign of A.

		Overflow →	1								Carry
A_S	1			1	0	1	0	0	0	1	A_M
B_S	1		+	1	1	0	1	0	1	0	$(\overline{B}_M + 1)$
R_S	1			0	1	1	1	0	1	1	R_M

Checking the result in decimal, (–81) – (–22) = (–59).

Arithmetic operations on reals

All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to reals stored in floating-point format. Multiplication of two reals involves multiplication of two integers in sign-and-magnitude representation. Division of two reals involves division of two integers in sign-and-magnitude representations. Since we did not discuss the multiplication or division of integers in sign-and-magnitude representation, we will not discuss the multiplication and division of reals, and only show addition and subtraction for reals.

Addition and subtraction of reals

Addition and subtraction of real numbers stored in floating-point numbers is reduced to addition and subtraction of two integers stored in sign-and-magnitude (combination of sign and mantissa) after the alignment of decimal points. Figure 4.8 shows a simplified version of the procedure (there are some special cases that we have ignored).

The simplified procedure works as follows:

1. If any of the two numbers (A or B) is zero, we let the result be 0 and stop.
2. If the operation is subtraction, we change the sign of the second number (B) to simulate addition.
3. We denormalize both numbers by including the hidden 1 in the mantissa and incrementing the exponents. The mantissa is now treated as an integer.
4. We then align the exponents, which means that we increment the lower exponent and shift the corresponding mantissa until both have the same exponent. For example, if we have:

$$1.11101 \times 2^4 + 1.01 \times 2^2$$

We need to make both exponents 4:

$$1.11101 \times 2^4 + 0.0101 \times 2^4$$

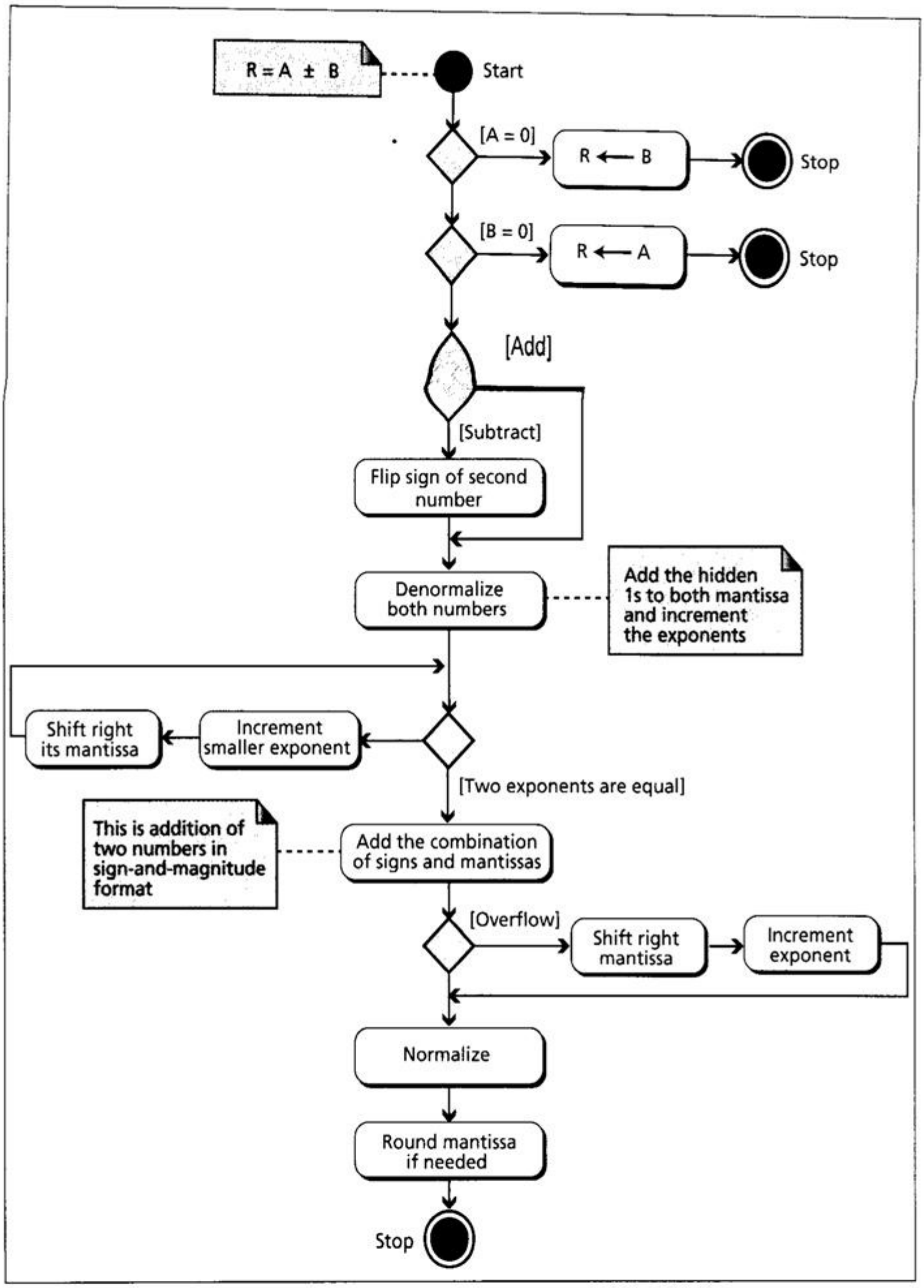


Figure 4.8 Addition and subtraction of reals in floating-point format

5. Now, we treat the combination of the sign and mantissa of each number as an integer in sign-and-magnitude format. We add these two integers, as explained earlier in this chapter.
6. Finally, we normalized the number again to 1.000111×2^5.

Example 4.24

Show how the computer finds the result of (+5.75) + (+161.875) = (+167.625).

Solution

As we saw in Chapter 3, these two numbers are stored in floating-point format, as shown below, but we need to remember that each number has a hidden 1 (which is not stored, but assumed).

	S	E	M
A	0	10000001	01110000000000000000000
B	0	10000110	01000011110000000000000

The first few steps in the UML diagram (Figure 4.8) are not needed. We move to denormalization and denormalize the numbers by adding the hidden 1s to the mantissa and incrementing the exponent. Now both denormalized mantissas are 24 bits and include the hidden 1s. They should be stored in a location that can hold all 24 bits. Each exponent is incremented.

	S	E	Denormalized M
A	0	10000010	101110000000000000000000
B	0	10000111	101000011110000000000000

Now we need to align the mantissas. We need to increment the first exponent and shift its mantissa to the right. We change the first exponent to $(10000111)_2$, so we need to shift the first mantissa right by five positions.

	S	E	Denormalized M
A	0	10000111	000001011100000000000000
B	0	10000111	101000011110000000000000

Now we do sign-and-magnitude addition, treating the sign and the mantissa of each number as one integer stored in sign-and-magnitude representation.

	S	E	Denormalized M
R	0	10000111	101001111010000000000000

There is no overflow in the mantissa, so we normalize.

	S	E	M
R	0	10000110	01001111010000000000000

The mantissa is only 23 bits, no rounding is needed. $E = (10000110)_2 = 134$ $M = 0100111101$. In other words, the result is $(1.0100111101)_2 \times 2^{134-127} = (10100111.101)_2 = 167.625$.

Example 4.25

Show how the computer finds the result of (+5.75) + (−7.0234375) = −1.2734375.

Solution

These two numbers can be stored in floating-point format, as shown below:

	S	E	M
A	0	10000001	01110000000000000000000
B	1	10000001	11000001100000000000000

Denormalization results in:

	S	E	Denormalized M
A	0	10000010	10111000000000000000000
B	1	10000010	11100000110000000000000

Alignment is not needed (both exponents are the same), so we apply addition operation on the combinations of sign and mantissa. The result is shown below, in which the sign of the result is negative:

	S	E	Denormalized M
R	1	10000010	00101000110000000000000

Now we need to normalize. We decrement the exponent three times and shift the denormalized mantissa to the left three positions:

	S	E	M
R	1	01111111	01000110000000000000000

The mantissa is now 24 bits, so we round it to 23 bits.

	S	E	M
R	1	01111111	01000110000000000000000

The result is $R = -2^{127-127} \times 1.0100011 = -1.2734375$, as expected.

4.4 RECOMMENDED READING

For more details about the subjects discussed in this chapter, the following books are recommended:

- Mano M: *Computer System Architecture*, Upper Saddle River, NJ: Prentice Hall, 1993
- Null L and Lobur J: *Computer Organization and Architecture*, Sudbury, MA: Jones and Bartlett, 2003
- Stalling, W: *Computer Organization and Architecture*, Upper Saddle River, NJ: Prentice Hall, 2000.

4.5 KEY TERMS

This chapter has introduced the following key terms, which are listed here with the pages on which they first occur:

AND operation 79
arithmetic shift operation 84
circular shift operation 84
logical shift operation 83
NOT operation 79
rotate operation 84
truth table 78
XOR operation 79
arithmetic operation 86
Boolean algebra 78
logic operation 78
mask 81
OR operation 79
set 81
unset 81

4.6 SUMMARY

- Operations on data can be divided into three broad categories: logic operations, shift operations, and arithmetic operations. Logic operations refer to those operations that apply the same basic operation to individual bits of a pattern or to two corresponding bits in two patterns. Shift operations move the bits in the pattern. Arithmetic operations involve adding, subtracting, multiplying, and dividing.
- The four logic operators discussed in this chapter (NOT, AND, OR, and XOR) can be used at the bit level or the pattern level. The NOT operator is a unary operator, while the AND, OR, and XOR operators are binary operators.
- The only application of the NOT operator is to complement the whole pattern. One of the applications of the AND operator is to unset (force to 0) specific bits in a bit pattern. One of the applications of the OR operator is to set (force to 1) specific bits in a bit pattern. One of the applications of the XOR operator is to flip (complement) specific bits in a bit pattern.
- Shift operations move the bits in the pattern; they change the positions of the bits. We can divide shift operations into two categories: logical shift operations and arithmetic shift operations. A logical shift operation is applied to a pattern that does not represent a signed number. Arithmetic shift operations assume that the bit pattern is a signed integer in two's complement format.
- All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to integers. Integers are normally stored in two's complement format. One of the advantages

of two's complement representation is that there is no difference between addition and subtraction. When the subtraction operation is encountered, the computer simply changes it to an addition operation, but forms the two's complement of the second number. Addition and subtraction for integers in sign-and-magnitude representation looks very complex. We have eight situations to consider.

- All arithmetic operations such as addition, subtraction, multiplication, and division can be applied to reals stored in floating-point format. Addition and subtraction of real numbers stored in floating-point numbers is reduced to addition and subtraction of two integers stored in sign and magnitude after the alignment of decimal points.

4.7 PRACTICE SET

Review questions

1. What is the difference between an arithmetic operation and a logical operation?
2. What happens to the carry from the leftmost column in the addition of integers in two's complement format?
3. Can n, the bit allocation, equal 1? Why, or why not?
4. Define the term *overflow*.
5. In the addition of floating-point numbers, how do we adjust the representation of numbers with different exponents?
6. What is the difference between a unary operation and a binary operation?
7. Name the logical binary operations.
8. What is a truth table?
9. What does the NOT operator do?
10. When is the result of an AND operator true?
11. When is the result of an OR operator true?
12. When is the result of an XOR operator true?
13. Mention an important property of the AND operator discussed in this chapter.
14. Mention an important property of the OR operator discussed in this chapter.
15. Mention an important property of the XOR operator discussed in this chapter.
16. What binary operation can be used to set bits? What bit pattern should the mask have?
17. What binary operation can be used to unset bits? What bit pattern should the mask have?
18. What binary operation can be used to flip bits? What bit pattern should the mask have?
19. What is the difference between logical and arithmetic shifts?

Multiple-choice questions

20. _____ is an arithmetic bit operation.
    a. The exclusive OR
    b. The unary NOT
    c. Subtraction
    d. All of the above
21. _____ is a logical bit operator.
    a. The exclusive OR
    b. The unary NOT
    c. The binary AND
    d. All of the above
22. The _____ method of integer representation is the most common method for storing integers in computer memory.
    a. sign-and-magnitude
    b. one's complement
    c. two's complement
    d. unsigned integers
23. In two's complement addition, if there is a final carry after the leftmost column addition, _____.
    a. add it to the rightmost column
    b. add it to the leftmost column
    c. discard it
    d. increase the bit length

24. For an 8-bit allocation, the smallest decimal number that can be represented in two's complement form is _____.
    a. –8
    b. –127
    c. –128
    d. –256
25. For an 8-bit allocation, the largest decimal number that can be represented in two's complement form is _____.
    a. 8
    b. 127
    c. 128
    d. 256
26. In two's complement representation with a 4-bit allocation, we get _____ when we add 1 to 7.
    a. 8
    b. 1
    c. –7
    d. –8
27. In two's complement representation with a 4-bit allocation, we get _____ when we add 5 to 5.
    a. –5
    b. –6
    c. –7
    d. 10
28. If the exponent in Excess_127 is binary 10000101, the exponent in decimal is _____.
    a. 6
    b. 7
    c. 8
    d. 9
29. If we are adding two numbers, one of which has an exponent value of 7 and the other an exponent value of 9, we need to shift the decimal point of the smaller number _____.
    a. one place to the left
    b. one place to the right
    c. two places to the left
    d. two places to the right
30. The binary _____ operator takes two inputs to produce one output.
    a. AND
    b. OR
    c. XOR
    d. all of the above
31. The unary _____ operator inverts its single input.
    a. AND
    b. OR
    c. NOT
    d. XOR
32. For the binary _____ operator, if the input is two 0s, the output is 0.
    a. AND
    b. OR
    c. XOR
    d. all of the above
33. For the binary _____ operator, if the input is two 1s, the output is 0.
    a. AND
    b. OR
    c. XOR
    d. all of the above
34. For the binary AND operation, only an input of _____ gives an output of 1.
    a. two 0s
    b. two 1s
    c. one 0 and one 1
    d. any of the above
35. For the binary OR operation, only an input of _____ gives an output of 0.
    a. two 0s
    b. two 1s
    c. one 0 and one 1
    d. any of the above
36. We use a bit pattern called a _____ to modify another bit pattern.
    a. mask
    b. carry
    c. float
    d. byte
37. To flip all the bits of a bit pattern, make a mask of all 1s and then _____ the bit pattern and the mask.
    a. AND
    b. OR

c. XOR
d. NOT

38. To unset (force to 0) all the bits of a bit pattern, make a mask of all 0s and then _____ the bit pattern and the mask.
a. AND
b. OR
c. XOR
d. NOT

39. To set (force to 1) all the bits of a bit pattern, make a mask of all 1s and then _____ the bit pattern and the mask.
a. AND
b. OR
c. XOR
d. NOT

Exercises

40. Show the result of the following operations:
a. NOT $(99)_{16}$
b. NOT $(FF)_{16}$
c. NOT $(00)_{16}$
d. NOT $(01)_{16}$

41. Show the result of the following operations:
a. $(99)_{16}$ AND $(99)_{16}$
b. $(99)_{16}$ AND $(00)_{16}$
c. $(99)_{16}$ AND $(FF)_{16}$
d. $(FF)_{16}$ AND $(FF)_{16}$

42. Show the result of the following operations:
a. $(99)_{16}$ OR $(99)_{16}$
b. $(99)_{16}$ OR $(00)_{16}$
c. $(99)_{16}$ OR $(FF)_{16}$
d. $(FF)_{16}$ OR $(FF)_{16}$

43. Show the result of the following operations:
a. NOT [$(99)_{16}$ OR $(99)_{16}$)]
b. $(99)_{16}$ OR [NOT $(00)_{16}$]
c. [$(99)_{16}$ AND $(33)_{16}$)] OR [$(00)_{16}$ AND $(FF)_{16}$]
d. $(99)_{16}$ OR $(33)_{16}$ AND [$(00)_{16}$ OR $(FF)_{16}$]

44. We need to unset (force to 0) the four leftmost bits of a pattern. Show the mask and the operation.

45. We need to set (force to 1) the four rightmost bits of a pattern. Show the mask and the operation.

46. We need to flip the three rightmost and the two leftmost bits of a pattern. Show the mask and the operation.

47. We need to unset the three leftmost bits and set the two rightmost bits of a pattern. Show the masks and operations.

48. Use the shift operation to divide an integer by 4.

49. Use the shift operation to multiply an integer by 8.

50. Use a combination of logical and shift operations to extract the fourth and fifth bits of an unsigned integer.

51. Using an 8-bit allocation, first convert each of the following integers to two's complement, do the operation, and then convert the result to decimal.
a. 19 + 23
b. 19 – 23
c. –19 + 23
d. –19 – 23

52. Using a 16-bit allocation, first convert each of the following numbers to two's complement, do the operation, and then convert the result to decimal.
a. 161 + 1023
b. 161 – 1023
c. –161 + 1023
d. –161 –1023

53. Which of the following operations creates an overflow if the numbers and the result are represented in 8-bit two's complement representation?
a. 11000010 + 00111111
b. 00000010 + 00111111
c. 11000010 + 11111111
d. 00000010 + 11111111

54. Without actually doing the calculation, can we tell which of the following creates an overflow if the numbers and the result are in 8-bit two's complement representation?
    a. 32 + 105
    b. 32 – 105
    c. –32 + 105
    d. –32 – 105
55. Show the result of the following operations assuming that the numbers are stored in 16-bit two's complement representation. Show the result in hexadecimal notation.
    a. $(012A)_{16} + (0E27)_{16}$
    b. $(712A)_{16} + (9E00)_{16}$
    c. $(8011)_{16} + (0001)_{16}$
    d. $(E12A)_{16} + (9E27)_{16}$
56. Using an 8-bit allocation, first convert each of the following numbers to sign-and-magnitude representation, do the operation, and then convert the result to decimal.
    a. 19 + 23
    b. 19 – 23
    c. –19 + 23
    d. –19 – 23
57. Show the result of the following floating-point operations using IEEE_127—see Chapter 3.
    a. 34.75 + 23.125
    b. –12.625 + 451.00
    c. 33.1875 – 0.4375
    d. –344.3125 – 123.5625
58. In which of the following situations does an overflow never occur? Justify the answer.
    a. Adding two positive integers.
    b. Adding one positive integer to a negative integer.
    c. Subtracting one positive integer from a negative integer.
    d. Subtracting two negative integers.
59. What is the result of adding an integer to its one's complement?
60. What is the result of adding an integer to its two's complement?

We can divide the parts that make up a computer into three broad categories or subsystem: the central processing unit (CPU), the main memory, and the input/output subsystem. The next three sections discusses these subsystems and how they are connected to make a standalone computer. Figure 5.1 shows the three subsystems of a standalone computer.

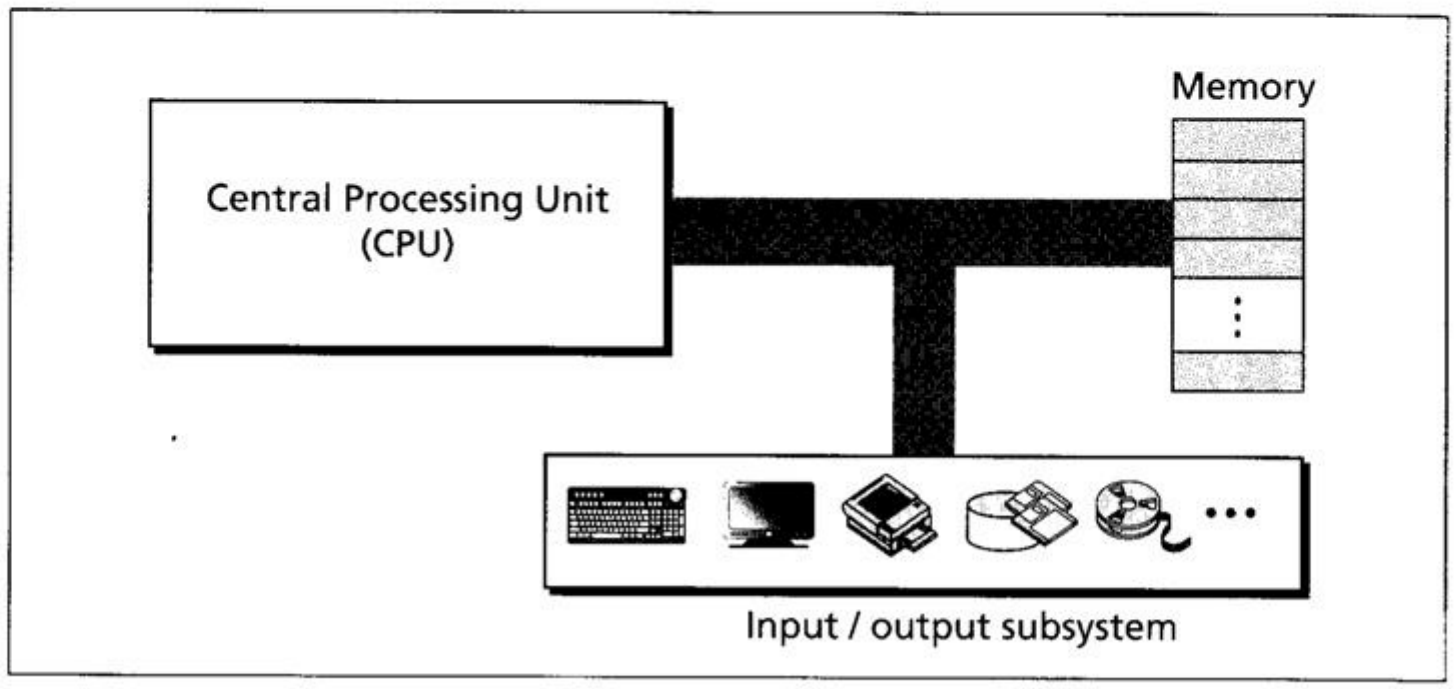


Figure 5.1 Computer hardware (subsystems)

5.1 CENTRAL PROCESSING UNIT

The **central processing unit** (CPU) performs operations on data. In most architectures it has three parts: an arithmetic logic unit (ALU), a control unit, and a set of registers, fast storage locations (Figure 5.2).

The arithmetic logic unit (ALU)

The **arithmetic logic unit** (ALU) performs logic, shift, and arithmetic operations on data.

Logic operations

We discussed several logic operations, such as NOT, AND, OR, and XOR, in Chapter 4. These operations treat the input data as bit patterns and the result of the operation is also a bit pattern.

Shift operations

We discussed two groups of shift operations on data in Chapter 4: logical shift operations and arithmetic shift operations. Logical shift operations are used to shift bit patterns to the left or right, while logical arithmetic operations are applied to integers. Their main purpose is to divide or multiply integers by two.

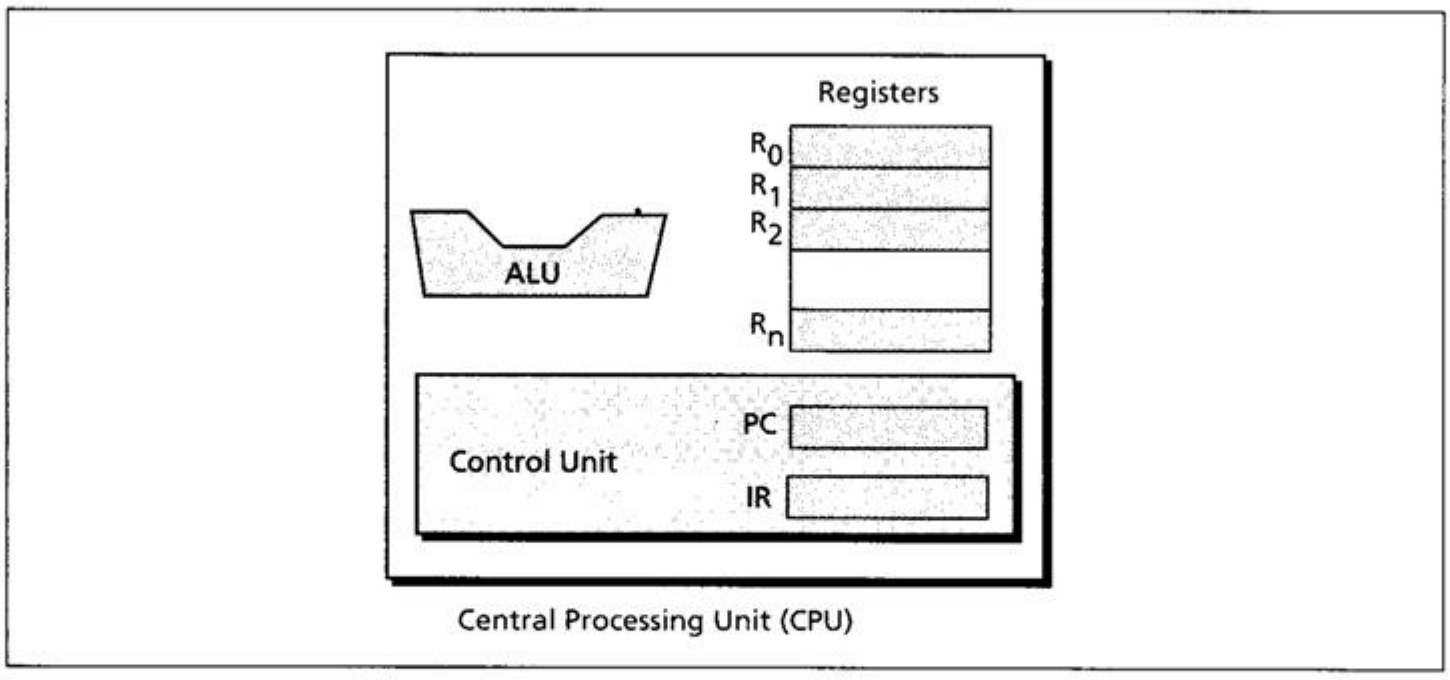


Figure 5.2 Central processing unit (CPU)

We discussed some arithmetic operations on integers and reals in Chapter 4. We mention that some operations can be implemented more efficiently in hardware.

Registers

Registers are fast stand-alone storage locations that hold data temporarily. Multiple registers are needed to facilitate the operation of the CPU. Some of these registers are shown in Figure 5.2.

In the past computers had only a few data registers to hold the input data and the result of the operations. Today, computers use dozens of registers inside the CPU to speed up their operations, because complex operations are done using hardware instead of software. These require several registers to hold the intermediate results. Data registers are named R_1 to R_n in Figure 5.2.

Today computers store not only data, but also programs, in their memory. The CPU is responsible for fetching instructions one by one from memory, storing them in the **instruction register** (register IR in Figure 5.2), decoding them, and executing them. We will discuss this issue later in the chapter.

Another common register in the CPU is the **program counter** (PC in Figure 5.2). The program counter keeps track of the instruction currently being executed. After execution of the instruction, the counter is incremented to point to the address of the next instruction in memory.

The control unit

The third part of any CPU is the control unit. The **control unit** controls the operation of each subsystem. Controlling is achieved through signals sent from the control unit to other subsystems.

5.2 MAIN MEMORY

Main memory is the second major subsystem in a computer (Figure 5.3). It consists of a collection of storage locations, each with a unique identifier, called an *address*. Data is transferred to and from memory in groups of bits called *words*. A word can be a group of 8 bits, 16 bits, 32 bits, or 64 bits (and growing). If the word is 8 bits, it is referred to as a *byte*. The term "byte" is so common in computer science that sometimes a 16-bit word is referred to as a 2-byte word, or a 32-bit word is referred to as a 4-byte word.

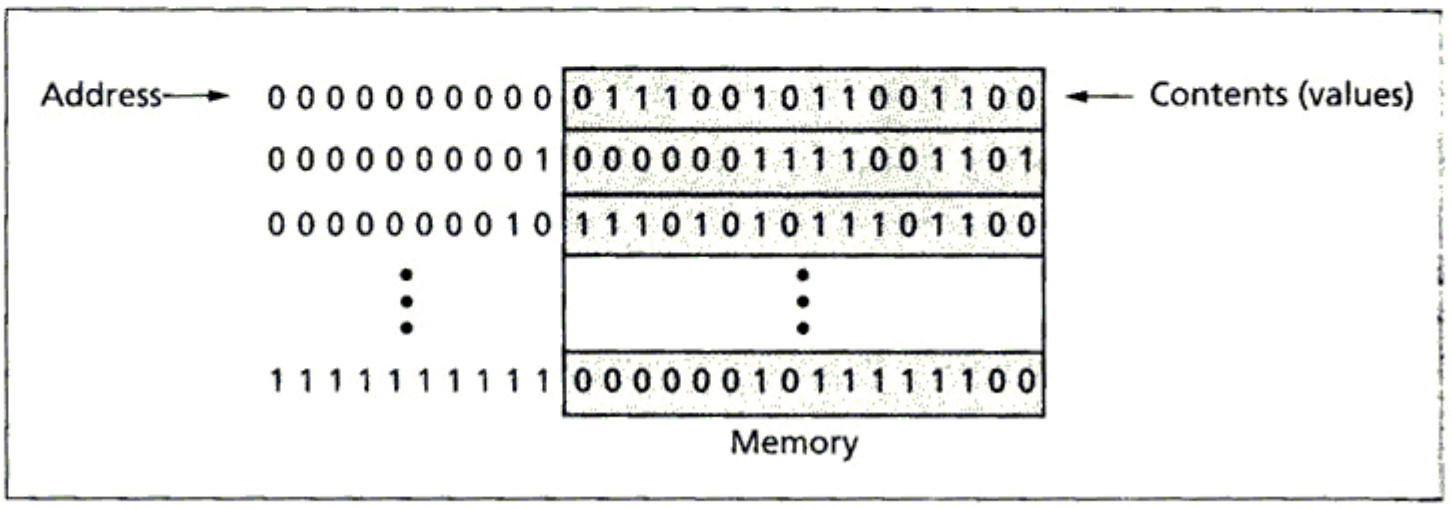


Figure 5.3 Main memory

Address space

To access a word in memory requires an identifier. Although programmers use a name to identify a word (or a collection of words), at the hardware level each word is identified by an address. The total number of uniquely identifiable locations in memory is called the **address space**. For example, a memory with 64 kilobytes and a word size of 1 byte has an address space that ranges from 0 to 65,535.

Table 5.1 shows the units used to refer to memory. Note that the terminology is misleading: it approximates the number of bytes in powers of 10, but the actual number of bytes is in powers of 2. Units in powers of 2 facilitates addressing.

Table 5.1 Memory units

Unit	*Exact number of bytes*	*Approximation*
kilobyte	2^{10} (1024) bytes	10^3 bytes
megabyte	2^{20} (1,048,576) bytes	10^6 bytes
gigabyte	2^{30} (1,073,741,824) bytes	10^9 bytes
terabyte	2^{40} bytes	10^{12} bytes

Addresses as bit patterns

Because computers operate by storing numbers as bit patterns, a memory address is also represented as a bit pattern. So if a computer has 64 kilobytes (2^{16}) of memory

with a word size of 1 byte, we need a bit pattern of 16 bits to define an address. Recall from Chapter 3 that addresses can be represented as unsigned integers (we do not have negative addresses). In other words, the first location is referred to as address 0000000000000000 (address 0), and the last location is referred to as address 1111111111111111 (address 65535). In general, if a computer has N words of memory, we need an unsigned integer of size $\log_2 N$ bits to refer to each memory location.

Memory addresses are defined using unsigned binary integers.

Example 5.1

A computer has 32 MB (megabytes) of memory. How many bits are needed to address any single byte in memory?

Solution

The memory address space is 32 MB, or 2^{25} ($2^5 \times 2^{20}$). This means that we need $\log_2 2^{25}$, or 25 bits, to address each byte.

Example 5.2

A computer has 128 MB of memory. Each word in this computer is eight bytes. How many bits are needed to address any single word in memory?

Solution

The memory address space is 128 MB, which means 2^{27}. However, each word is eight (2^3) bytes, which means that we have 2^{24} words. This means that we need $\log_2 2^{24}$, or 24 bits, to address each word.

Memory types

Two main types of memory exist: RAM and ROM.

Random access memory (RAM) makes up most of the main memory in a computer. In a random access device, a data item can be accessed randomly—using the address of the memory location—without the need to access all data items located before it. However, the term is confusing, because ROM can also be accessed randomly. What distinguishes RAM from ROM is that RAM can be read from and written to. The CPU can write something to RAM and later overwrite it. Another characteristic of RAM is that it is *volatile*: the information (program or data) is lost if the computer is powered down. In other words, all information in RAM is erased if you turn off the computer or if there is a power outage. RAM technology is divided into two broad categories: SRAM and DRAM.

SRAM

Static RAM (SRAM) technology uses traditional *flip-flop gates* (see Appendix E) to hold data. The gates hold their state (0 or 1), which means that data is stored as long as the power is on and there is no need to refresh memory locations. SRAM is fast but expensive.

DRAM

Dynamic RAM (DRAM) technology uses capacitors, electrical devices that can store energy, for data storage. If a capacitor is charged, the state is 1, if it is discharged, the state is 0. Because a capacitor loses some of its charge with time, DRAM memory cells need to be refreshed periodically. DRAMs are slow but inexpensive.

ROM

The contents of **read-only memory (ROM)** are written by the manufacturer, and the CPU can read from, but not write to, ROM. Its advantage is that it is *nonvolatile*—its contents are not lost if you turn off the computer. Normally, it is used for programs or data that must not be erased or changed even if you turn off the computer. For example, some computers come with ROM that holds the *boot program* that runs when we switch on the computer.

PROM

One variation of ROM is **programmable read-only memory (PROM)**. This type of memory is blank when the computer is shipped. The user of the computer, with some special equipment, can store programs on it. When programs are stored, it behaves like ROM and cannot be overwritten. This allows a computer user to store specific programs in PROM.

EPROM

A variation of PROM is **erasable programmable read-only memory (EPROM)**. It can be programmed by the user, but can also be erased with a special device that applies ultraviolet light. To erase EPROM memory requires physical removal and reinstallation of the EPROM.

EEPROM

A variation of EPROM is **electrically erasable programmable read-only memory (EEPROM)**. EEPROM can be programmed and erased using electronic impulses without being removed from the computer.

Memory hierarchy

Computer users need a lot of memory, especially memory that is very fast and inexpensive. This demand is not always possible to satisfy—very fast memory is usually not cheap. A compromise needs to be made. The solution is hierarchical levels of memory (Figure 5.4). The hierarchy is based on the following:

- Using a very small amount of costly high-speed memory where speed is crucial. The registers inside the CPU are of this type.
- Using a moderate amount of medium-speed memory to store data that is accessed often. *Cache memory*, discussed next, is of this type.
- Using a large amount of low-speed memory for data that is accessed very often. Main memory is of this type.

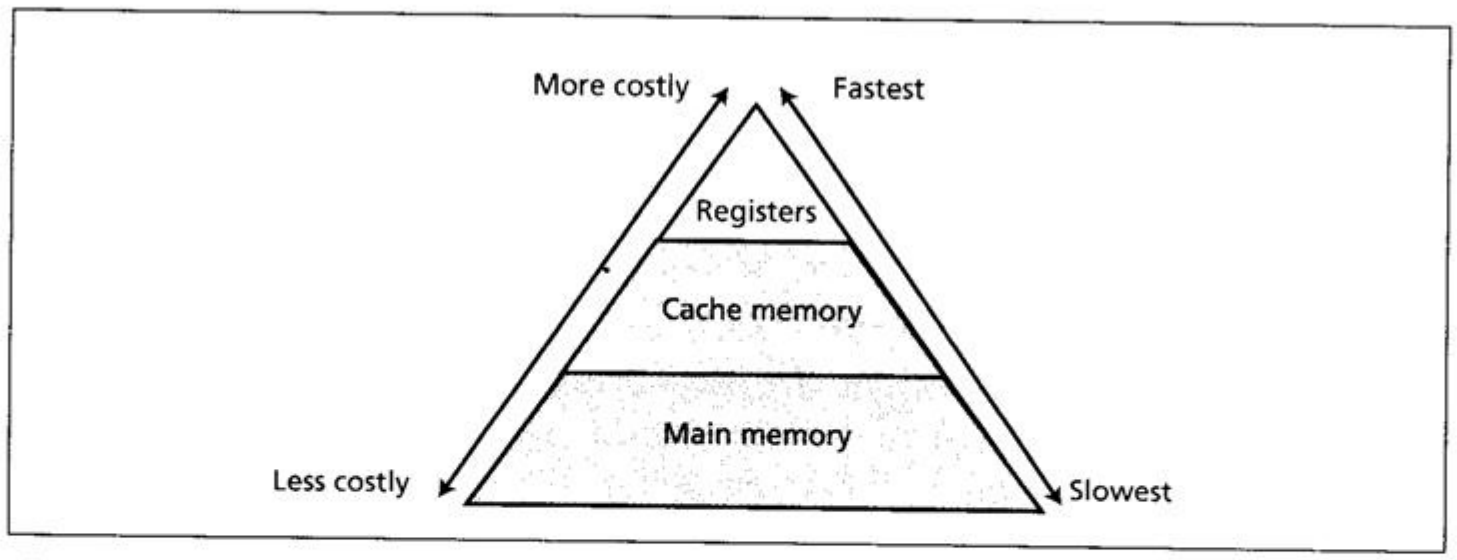


Figure 5.4 Memory hierarchy

Cache memory

Cache memory is faster than main memory but slower than the CPU and its registers. Cache memory, which is normally small in size, is placed between the CPU and main memory (Figure 5.5).

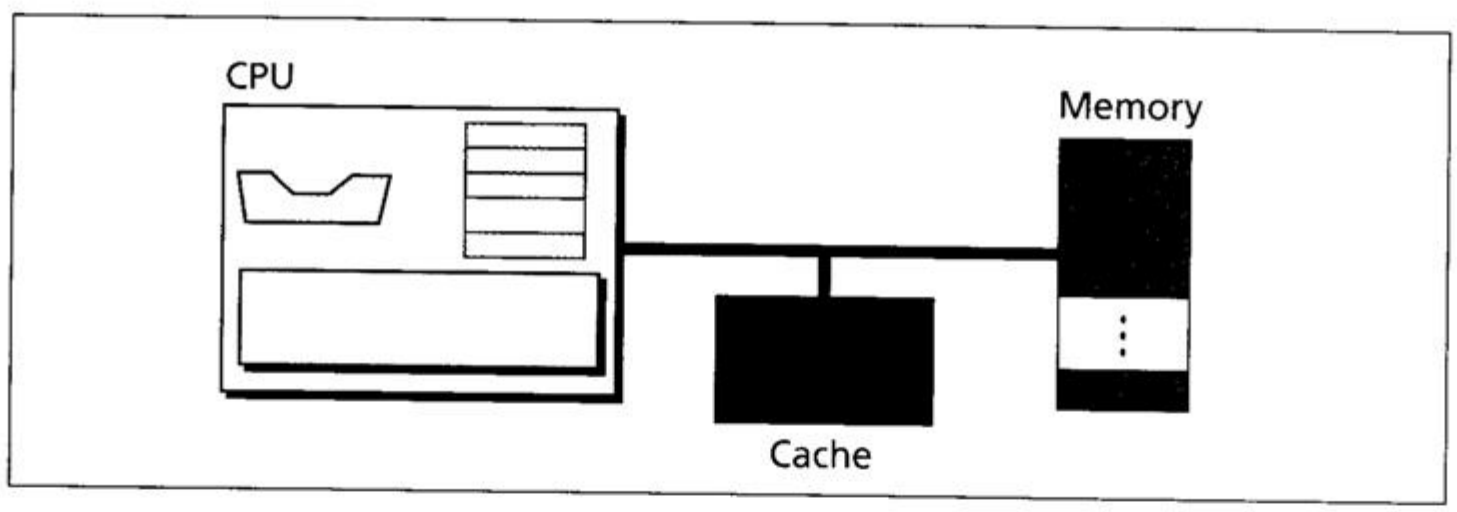


Figure 5.5 Cache memory

Cache memory at any time contains a copy of a portion of main memory. When the CPU needs to access a word in main memory, it follows this procedure:

1. The CPU checks the cache.
2. If the word is there, it copies the word: if not, the CPU accesses main memory and copies a block of memory starting with the desired word. This block replaces the previous contents of cache memory.
3. The CPU accesses the cache and copies the word.

This procedure can expedite operations: if the word is in the cache, it is accessed immediately. If the word is not in the cache, the word and a whole block are copied to the cache. Since it is probable that the CPU, in its next cycle, will need to access the words following the first word, the existence of the cache speeds processing.

We might wonder why cache memory is so efficient despite its small size. The answer lies in the "80–20 rule". It has been observed that most computers typically spend 80 percent of their time accessing only 20 percent of the data. In other words, the same data is accessed over and over again. Cache memory, with its high

speed, can hold this 20 percent to make access faster at least 80 percent of the time.

5.3 INPUT/OUTPUT SUBSYSTEM

The third major subsystem in a computer is the collection of devices referred to as the **input/output (I/O) subsystem**. This subsystem allows a computer to communicate with the outside world, and to store programs and data even when the power is off. Input/output devices can be divided into two broad categories: nonstorage and storage devices.

Nonstorage devices

Nonstorage devices allow the CPU/memory to communicate with the outside world, but they cannot store information.

Keyboard and monitor

Two of the more common nonstorage input/output devices are the keyboard and the monitor. The **keyboard** provides input, the **monitor** displays output and at the same time echoes input typed on the keyboard. Programs, commands, and data are input or output using strings of characters. The characters are encoded using a code such as ASCII (see Appendix A). Other devices that fall in this category are *mice*, *joysticks*, and so on.

Printer

A **printer** is an output device that creates a permanent record. A printer is a nonstorage device because the printed material cannot be entered directly into a computer again unless someone retypes or scans it.

Storage devices

Storage devices, although classified as I/O devices, can store large amounts of information to be retrieved at a later time. They are cheaper than main memory, and their contents are nonvolatile—that is, not erased when the power is turned off. They are sometimes referred to as *auxiliary storage devices*. We can categorize them as either magnetic or optical.

Magnetic storage devices

Magnetic storage devices use magnetization to store bits of data. If a location is magnetized, it represents 1, if not magnetized, it represents 0.

Magnetic disks

A **magnetic disk** consists of one or more disks stacked on top of each other. The disks are coated with a thin magnetic film. Information is stored on and retrieved from the surface of the disk using a **read/write head** for each magnetized surface of the disk. Figure 5.6 shows the physical layout of a magnetic disk drive and the organization of a disk.

- **Surface organization**. To organize data stored on the disk, each surface is divided into **tracks**, and each track is divided into **sectors** (Figure 5.6). The

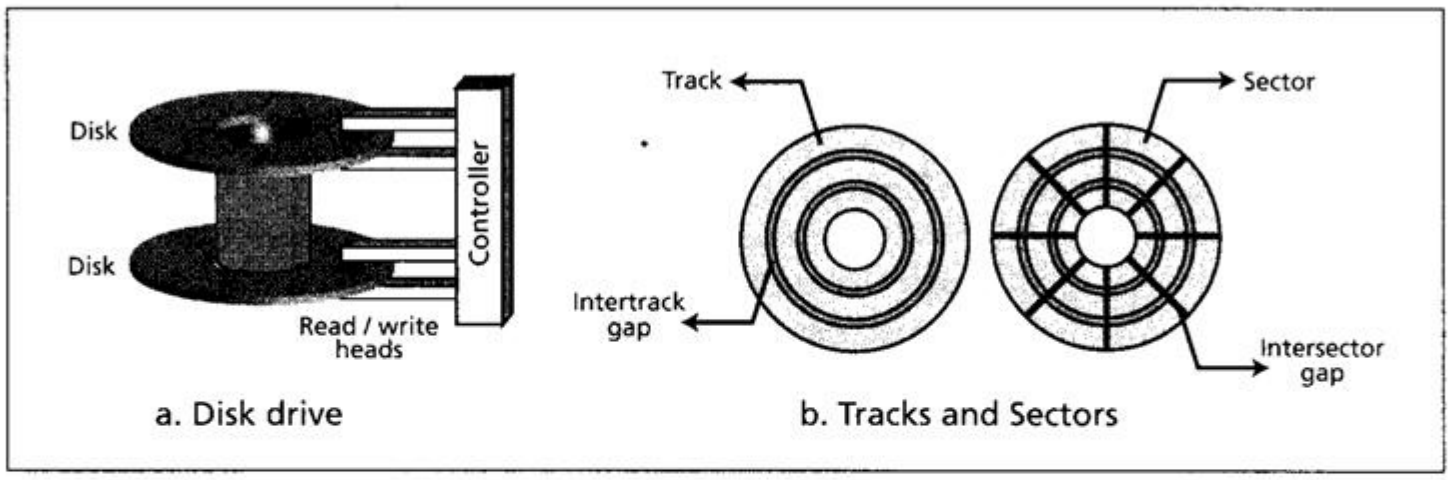


Figure 5.6 A magnetic disk

tracks are separated by an **intertrack gap**, and the sectors are separated by an **intersector gap**.

- **Data access**. A magnetic disk is considered a random access device. In a random access device, a data item can be access randomly without the need to access all other data items located before it. However, the smallest storage area that can be accessed at one time is a sector. A block of data can be stored in one or more sectors and retrieved without the need to retrieve the rest of the information on the disk.
- **Performance**. The performance of a disk depends on several factors, the most important being the rotational speed, the seek time, and the transfer time. The **rotational speed** defines how fast the disk is spinning. The **seek time** defines the time to move the read/write head to the desired track where the data is stored. The **transfer time** defines the time to move data from the disk to the CPU/memory.

Magnetic tape

Magnetic tape comes in various sizes. One common type is half- inch plastic tape coated with a thick magnetic film. The tape is mounted on two reels and uses a read/write head that reads or writes information when the tape is passed through it. Figure 5.7 shows the mechanical configuration of a magnetic tape drive.

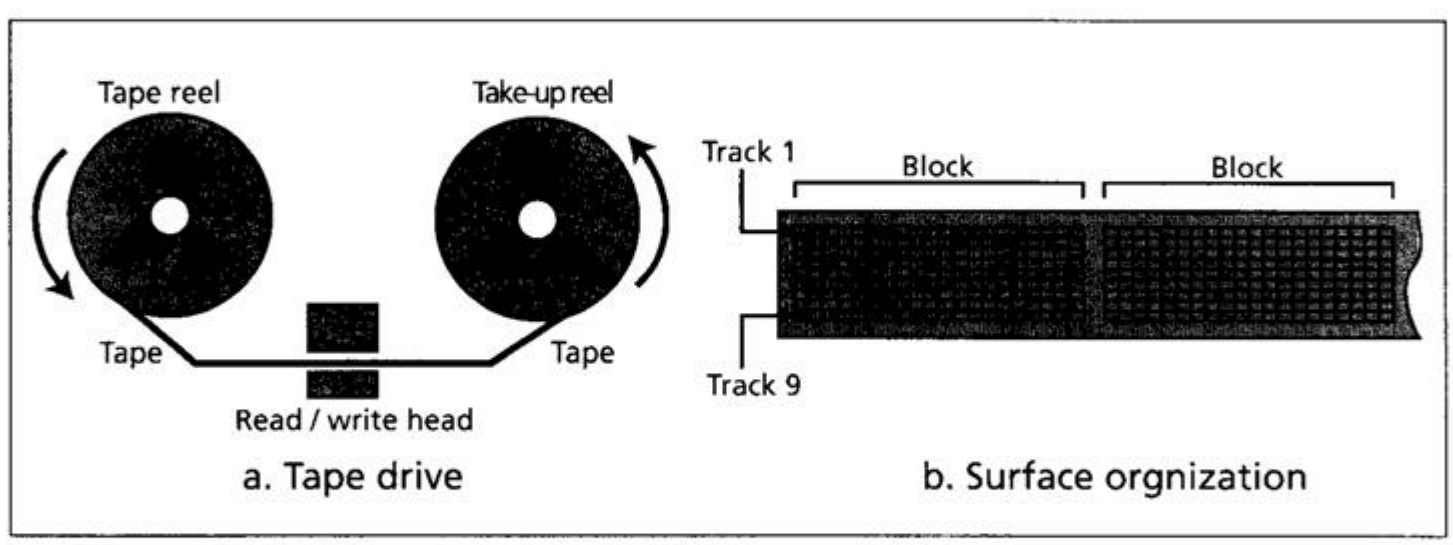


Figure 5.7 Magnetic tape

- **Surface organization**. The width of the tape is divided into nine tracks, each location on a track storing 1 bit of information. Nine vertical locations can store 8 bits of information related to a byte plus a bit for error detection (Figure 5.7).
- **Data access**. A magnetic tape is considered a sequential access device. Although the surface may be divided into blocks, there is no addressing mechanism to access each block. To retrieve a specific block on the tape, we need to pass through all the previous blocks.
- **Performance**. Although magnetic tape is slower than a magnetic disk, it is cheaper. Today, people use magnetic tape to back up large amounts of data.

Optical storage devices

Optical storage devices, a relatively recent technology, use laser light to store and retrieve data. The use of optical storage technology followed the invention of the compact disk (CD) used to store audio information. Today, the same technology—slightly improved—is used to store information in a computer. Devices that use this technology include CD-ROMs, CD-Rs, CD-RWs, and DVDs.

CD-ROMs

Compact disk read-only memory (CD-ROM) disks use the same technology as the audio CD, originally developed by Phillips and Sony for recording music. The only difference between these two technologies is enhancement: a CD-ROM drive is more robust and checks for errors. Figure 5.8 shows the steps involved in creating and using a CD-ROM.

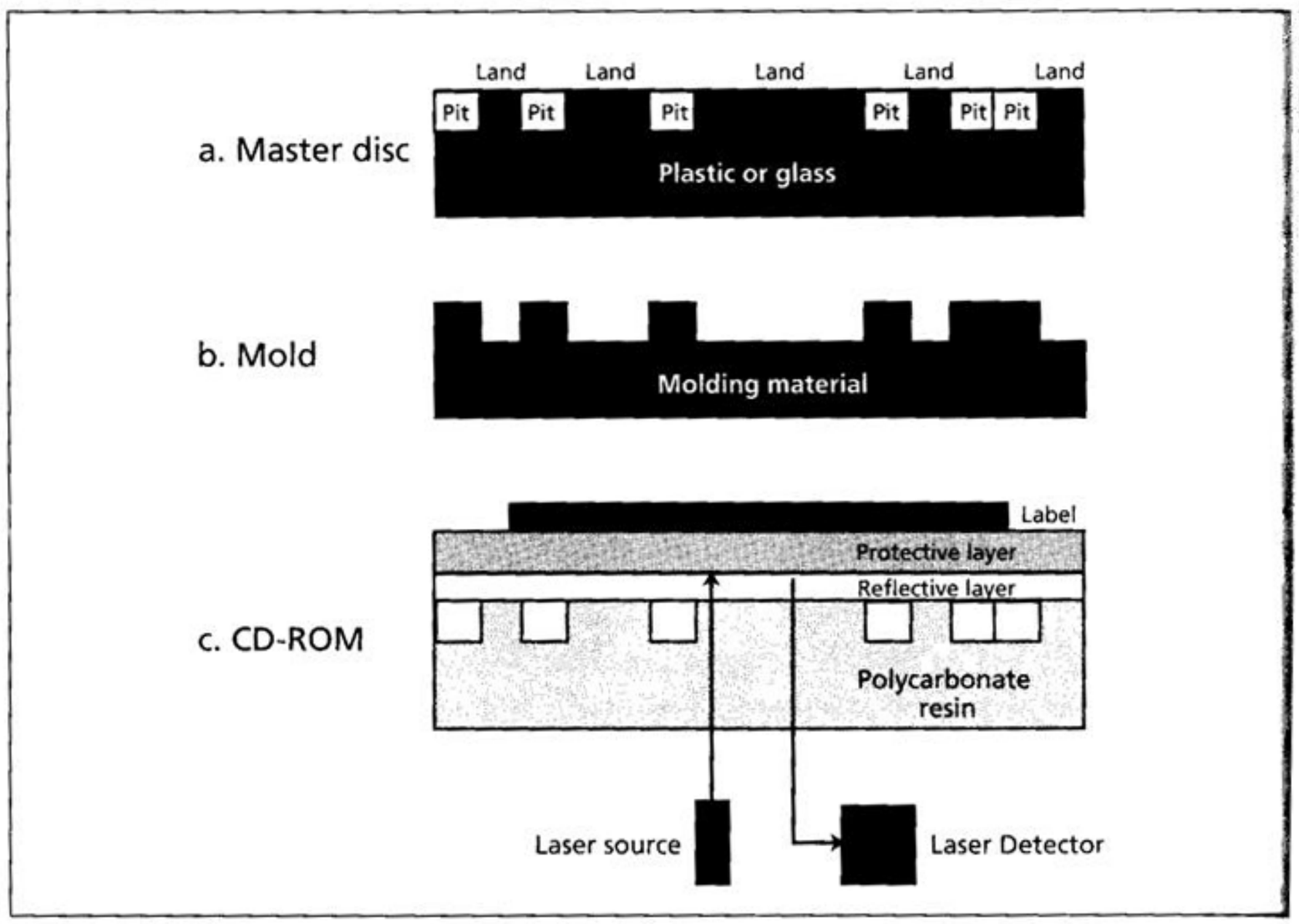


Figure 5.8 Creation and use of CD-ROMs

- **Creation**. CD-ROM technology uses three steps to create a large number of discs:
  a. A **master disk** is created using a high-power infrared laser that creates bit patterns on coated plastic. The laser translates the bit patterns into a sequence of **pits** (holes) and **lands** (no holes). The pits usually represent 0s and the lands usually represent 1s. However, this is only a convention, and it can be reversed. Other schemes use a transition (pit to hole or hole to pit) to represent 1, and a lack of transition to represent 0.
  b. From the master disk, a mold is made. In the mold, the pits (holes) are replaced by bumps.
  c. Molten **polycarbonate resin** is injected into the mold to produce the same pits as the master disk. A very thin layer of aluminum is added to the polycarbonate to provide a reflective surface. On top of this, a protective layer of lacquer is applied and a label is added. Only this last step needs to be repeated for each disk.
- **Reading**. The CD-ROM is read using a low-power laser beam. The beam is reflected by the aluminum surface when passing through a land. It is reflected twice when it encounters a pit, once by the pit boundary and once by the aluminum boundary. The two reflections have a destructive effect, because the depth of the pit is chosen to be exactly one-fourth of the beam wavelength. In other words, the sensor installed in the drive detects more light when the location is a land and less light when the location is a pit, so can read what was recorded on the original master disk and copied to the CD-ROM.
- **Format**. CD-ROM technology uses a different format than magnetic disk (Figure 5.9). The format of data on a CD-ROM is based on:
  a. A block of 8-bit data transformed into a 14-bit symbol using an error-correction method called Hamming code.
  b. A frame made up from 42 symbols (14 bits/symbol).
  c. A sector made up from 98 frames (2352 bytes).
- **Speed**. CD-ROM drives come in different speeds. Single speed is referred to as 1x, double speed 2x, and so on. If the drive is single speed, it can read up to 153,600 bytes per second. Table 5.2 shows the speeds and their corresponding data rates.

Table 5.2 CD-ROM speeds

Speed	*Data rate*	*Approximation*
1*x*	153,600 bytes per second	150 KB/s
2*x*	307,200 bytes per second	300 KB/s
4*x*	614,400 bytes per second	600 KB/s
6*x*	921,600 bytes per second	900 KB/s

Table 5.2 CD-ROM speeds (continued)

Speed	*Data rate*	*Approximation*
8*x*	1,228,800 bytes per second	1.2 MB/s
12*x*	1,843,200 bytes per second	1.8 MB/s
16*x*	2,457,600 bytes per second	2.4 MB/s
24*x*	3,688,400 bytes per second	3.6 MB/s
32*x*	4,915,200 bytes per second	4.8 MB/s
40*x*	6,144,000 bytes per second	6 MB/s

- **Application**. The expense involved in creating a master disk, mold, and the actual disk can be justified if there are a large number of potential customers. In other words, this technology is economical if the discs are mass produced.

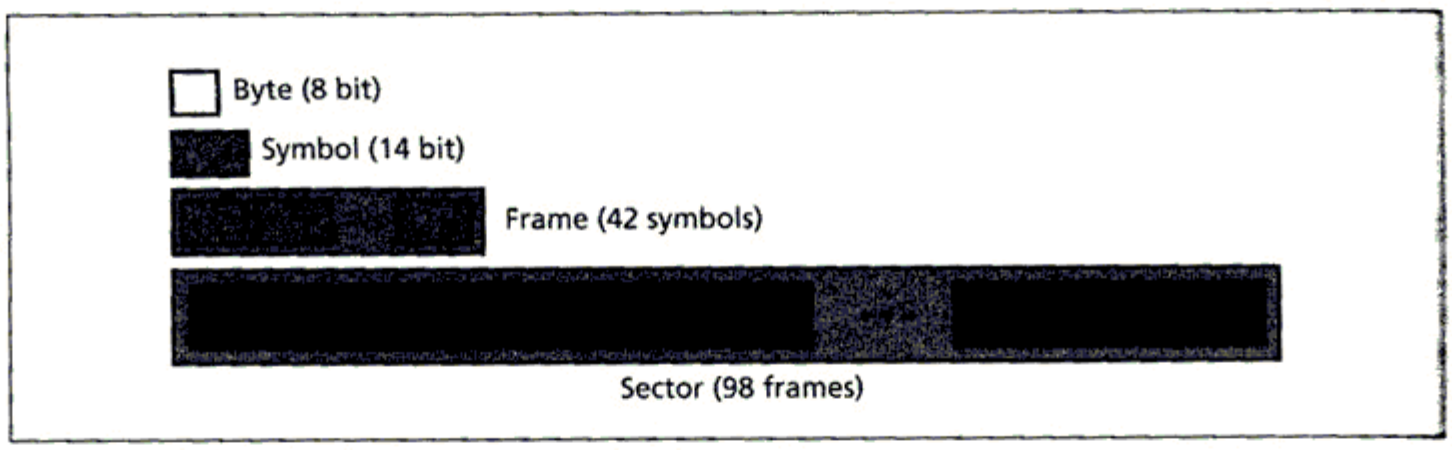


Figure 5.9 CD-ROM format

CD-R

Clearly, CD-ROM technology is justifiable only if the manufacturer can create a large number of disks. On the other hand, the **compact disk recordable (CD-R)** format allows users to create one or more disks without going through the expense involved in creating CD-ROMs. It is particularly useful for making backups. You can write once to CD-R disks, but they can be read many times. This is why the format is sometimes called **write once, read many (WORM)**.

- **Creation**. CD-R technology uses the same principles as CD-ROM to create a disk (Figure 5.10). The following lists the differences:
  a. There is no master disk or mold.
  b. The reflective layer is made of gold instead of aluminum.
  c. There are no physical pits (holes) in the polycarbonate: the pits and lands are only simulated. To simulate pits and lands, an extra layer of dye, similar to the material used in photography, is added between the reflective layer and the polycarbonate.

d. A high-power laser beam, created by the CD burner of the drive, makes a dark spot in the dye, changing its chemical composition, which simulates a pit. The areas not struck by the beam become lands.

- **Reading**. CD-Rs can be read by a CD-ROM or a CD-R drive. This means that any differences should be transparent to the drive. The same low-power laser beam passes in front of the simulated pits and lands. For a land, the beam reaches the reflective layer and is reflected. For a simulated pit, the spot is opaque, so the beam cannot be reflected back.
- **Format and speed**. The format, capacity, and speed of CD-Rs are the same as CD-ROMs.
- **Application**. This technology is very attractive for the creation and distribution of a small number of disks. It is also very useful for making archive files and backups.

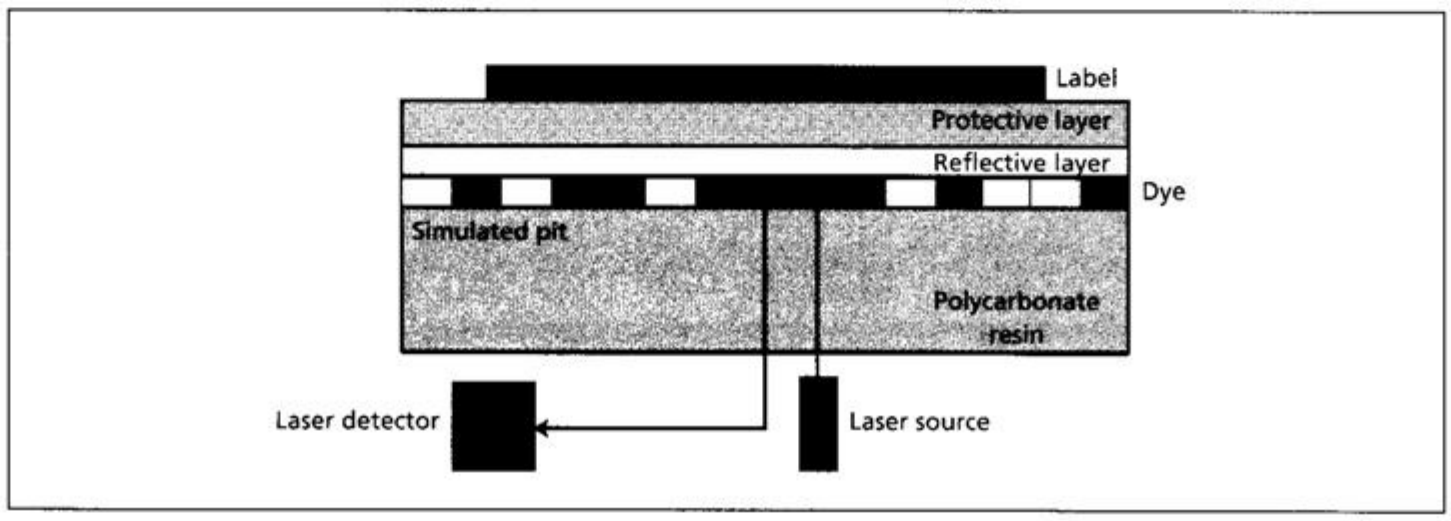


Figure 5.10 Making a CD-R

CD-RW

Although CD-Rs have become very popular, they can be written to only once. To overwrite previous materials, a new technology allows a new type of disk called **compact disk rewritable (CD-RW)**. It is sometimes called an *erasable optical disk*.

- **Creation**. CD-RW technology uses the same principles as CD-R to create the disk (Figure 5.11). The following lists the differences:
  a. Instead of dye, the technology uses an alloy of silver, indium, antimony, and tellurium. This alloy has two stable states: crystalline (transparent) and amorphous (nontransparent).
  b. The drive uses high-power lasers to create simulated pits in the alloy (changing it from crystalline to amorphous).
- **Reading**. The drive uses the same type of low-power laser beam as CD-ROM and CD-R to detect pits and lands.
- **Erasing**. The drive uses a medium-power laser beam to change pits to lands. The beam changes a location from the amorphous state to the crystalline state.
- **Format and speed**. The format, capacity, and speed of CD-RWs are the same as CD-ROMs.

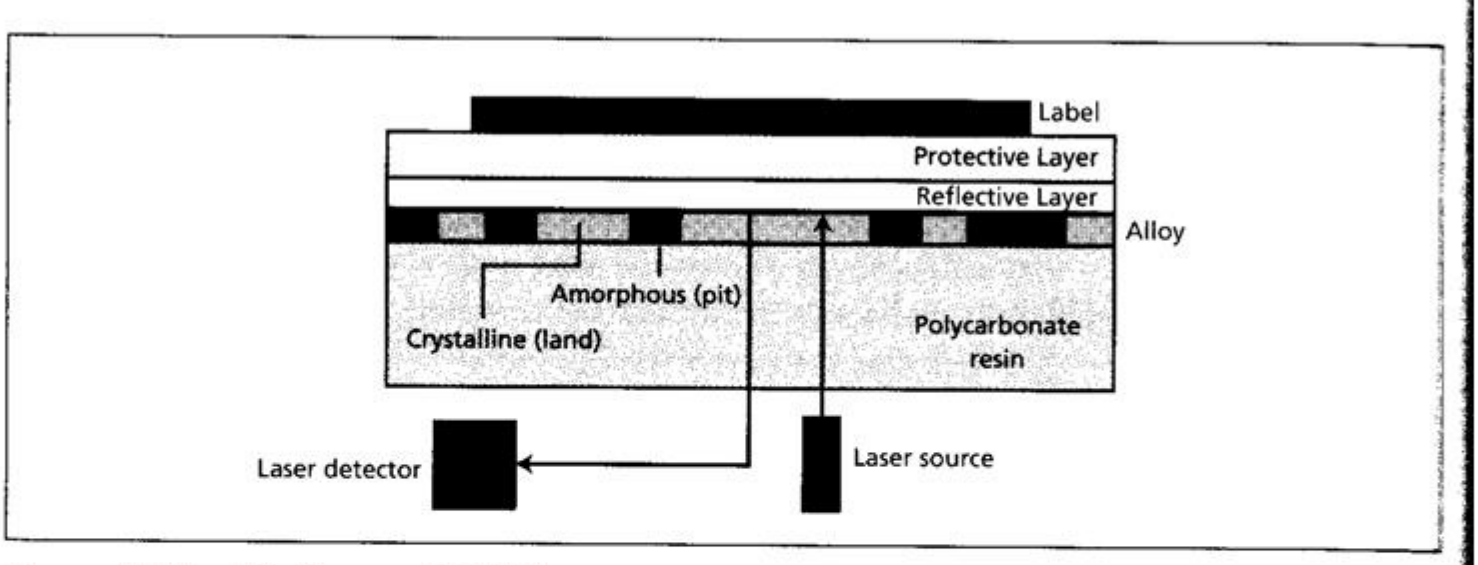


Figure 5.11 Making a CD-RW

- **Application**. The technology is definitely more attractive than CD-R technology. However, CD-Rs are more popular for two reasons. First, blank CD-R discs are less expensive than blank CD-RW discs. Second, CD-Rs are preferable in cases where the created disk must not be changed, either accidentally or intentionally.

DVD

The industry has felt the need for digital storage media with even higher capacity. The capacity of a CD-ROM (650 MB) is insufficient to store video information. The latest optical memory storage device on the market is called a **digital versatile disk (DVD)**. It uses a technology similar to CD-ROM, but with the following differences:

a. The pits are smaller: 0.4 microns in diameter instead of the 0.8 microns used in CDs.

b. The tracks are closer to each other.

c. The beam is a red laser instead of infrared.

d. DVDs use one to two recording layers, and can be single-sided or double-sided.

- **Capacity**. These improvements result in higher capacities (Table 5.3).

Table 5.3 DVD capacities

Feature	*Capacity*
Single-sided, single-layer	4.7 GB
Single-sided, dual-layer	8.5 GB
Double-sided, single-layer	9.4 GB
Double-sided, dual-layer	17 GB

- **Compression**. DVD technology uses MPEG (see Chapter 15) for compression. This means that a single-sided, single-layer DVD can hold 133 minutes of video at high resolution. This also includes both audio and subtitles.

- **Application**. Today, the high capacity of DVDs attracts many applications that need to store a high volume of data.

5.4 SUBSYSTEM INTERCONNECTION

The previous sections outlined the characteristics of the three subsystems (CPU, main memory, and I/O) in a stand-alone computer. In this section, we explore how these three subsystems are interconnected. The interconnection plays an important role because information needs to be exchanged between the three subsystems.

Connecting CPU and memory

The CPU and memory are normally connected by three groups of connections, each called a **bus:** data bus, address bus, and control bus (Figure 5.12).

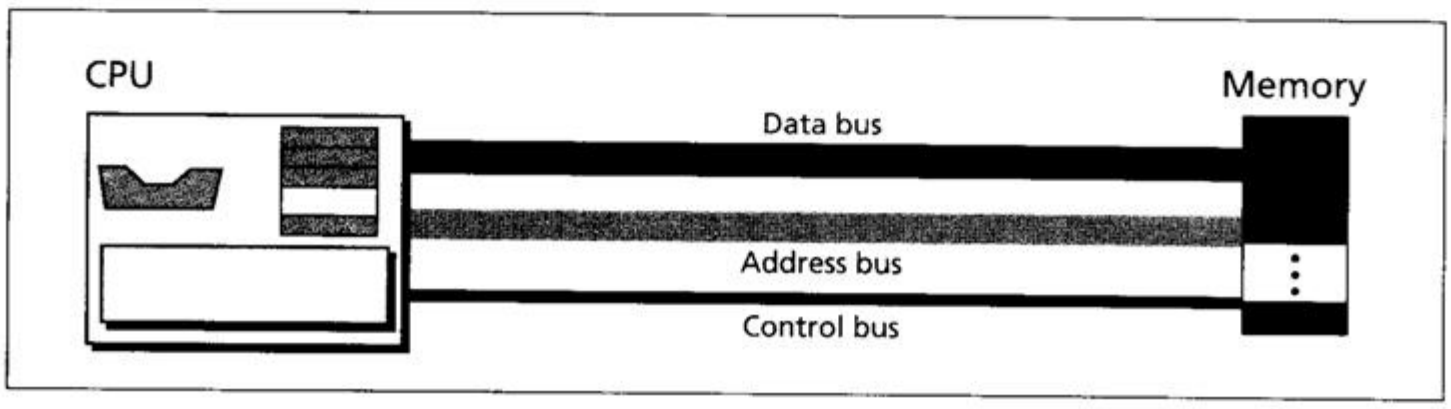


Figure 5.12 Connecting CPU and memory using three buses

bus The **data bus** is made of several connections, each carrying 1 bit at a time. The number of connections depends on the size of the word used by the computer. If the word is 32 bits (4 bytes), we need a data bus with 32 connections so that all 32 bits of a word can be transmitted at the same time.

ess bus The **address bus** allows access to a particular word in memory. The number of connections in the address bus depends on the address space of the memory. If the memory has 2^n words, the address bus needs to carry n bits at a time. Therefore, it must have n connections.

ol bus The **control bus** carries communication between the CPU and memory. For example, there must be a code, sent from the CPU to memory, to specify a read or write operation. The number of connections used in the control bus depends on the total number of control commands a computer needs. If a computer has 2^m control actions, we need m connections for the control bus, because m bits can define 2^m different operations.

Connecting I/O devices

I/O devices cannot be connected directly to the buses that connect the CPU and memory because the nature of I/O devices is different than the nature of CPU and

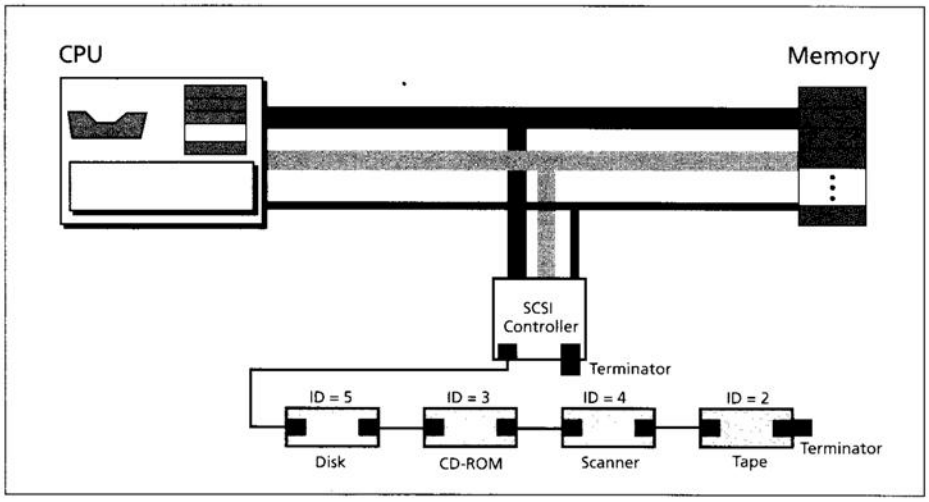


Figure 5.14 SCSI controller

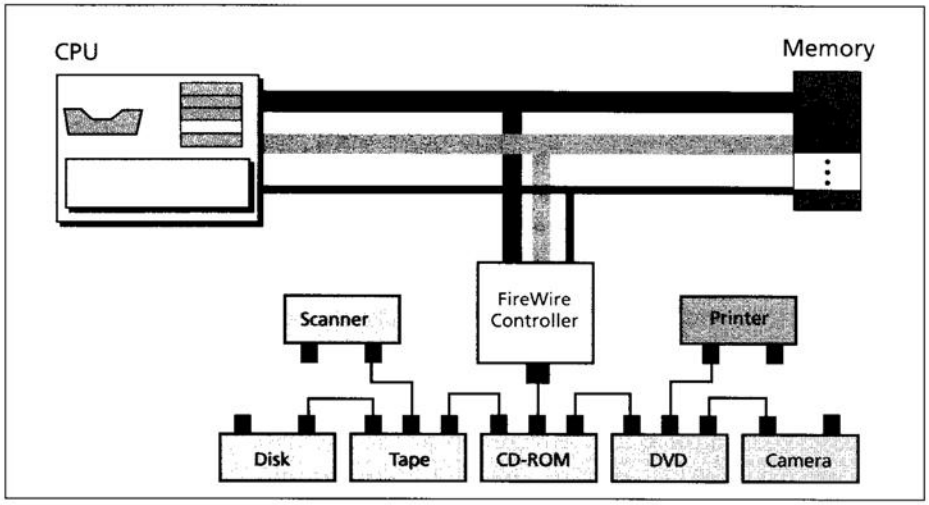


Figure 5.15 FireWire controller

devices to the computer bus. Figure 5.16 show the connection of the USB controller to the bus and the connection of devices to the controller.

Multiple devices can be connected to a USB controller, which is also referred to as a *root hub*. USB-2 (USB Version 2.0) allows up to 127 devices to be connected to a USB controller using a tree-like topology with the controller as the root of the tree, hubs as the intermediate nodes, and the devices as the end nodes. The difference between the controller (root hub) and the other hubs is that the controller is aware of the presence of other hubs in the tree, but other hubs are passive devices that simply pass the data.

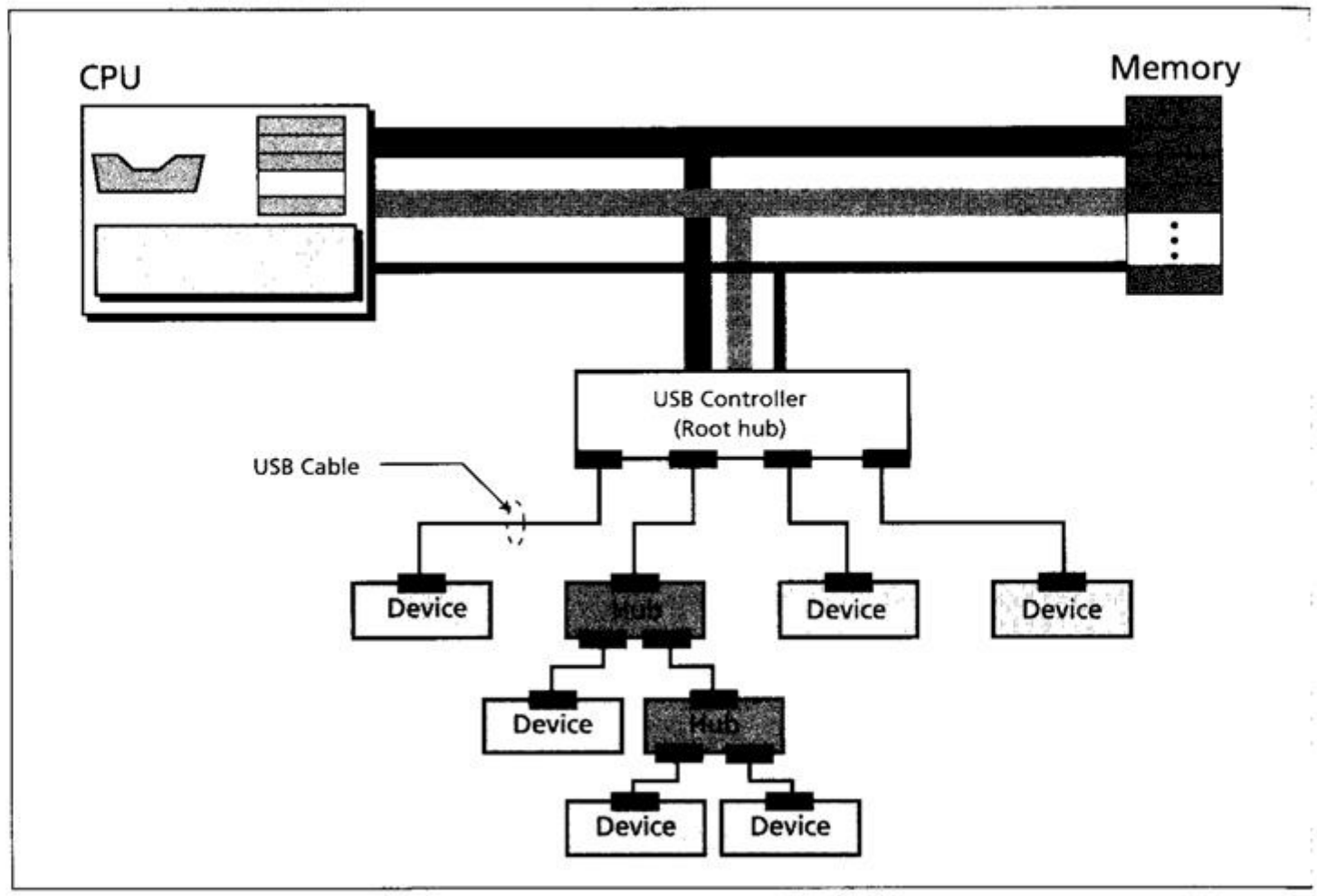


Figure 5.16 USB controller

Devices can easily be removed or attached to the tree without powering down the computer. This is referred to as *hot-swappable*. When a hub is removed from the system, all device and other hubs connected to it are also removed.

USB uses a cable with four wires. Two wires (+5 volts and ground) are used to provide power for low-power devices such as keyboards or mice. A high-power device needs to be connected to a power source. A hub get its power from the bus and can provide power for low-power devices. The other two wires (twisted together to reduce noise) are used to carry data, addresses, and control signals. USB uses two different connectors: A and B. The A connector (downstream connector) is rectangular and is used to connect to the USB controller or the hub. The B connector (upstream connector) is close to square and is used to connect to the device. Recently two new connectors, mini A and mini B, have been introduced that are used for connecting to small devices and laptop computers.

USB-2 provides three data transfer rates: 1.5 Mbps (megabits per second), 12 Mbps, and 480 Mbps. The low data rate can be used with slow devices such as keyboards and mice, the medium data rate with printers, and the high data rate with mass storage devices.

Data is transferred over USB in packets (see Chapter 6). Each packet contains an address part (device identifier), a control part, and part of the data to be transmitted to that device. All devices will receive the same packet, but only those devices with the address defined in the packet will accept it.

Addressing input/output devices

The CPU usually uses the same bus to read data from or write data to main memory and I/O device. The only difference is the instruction. If the instruction refers to a word in main memory, data transfer is between main memory and the CPU. If the instruction identifies an I/O device, data transfer is between the I/O device and the CPU. There are two methods for handling the addressing of I/O devices: isolated I/O and memory-mapped I/O.

ated I/O

In the **isolated I/O** method, the instructions used to read/write memory are totally different than the instructions used to read/write I/O devices. There are instructions to test, control, read from, and write to I/O devices. Each I/O device has its own address. The I/O addresses can overlap with memory addresses without any ambiguity because the instruction itself is different. For example, the CPU can use a command "Read 101" to read from memory word 101, and it can use a command "Input 101" to read from I/O device 101. There is no confusion, because the read command is for reading from memory and the input command is for reading from an I/O device (Figure 5.17).

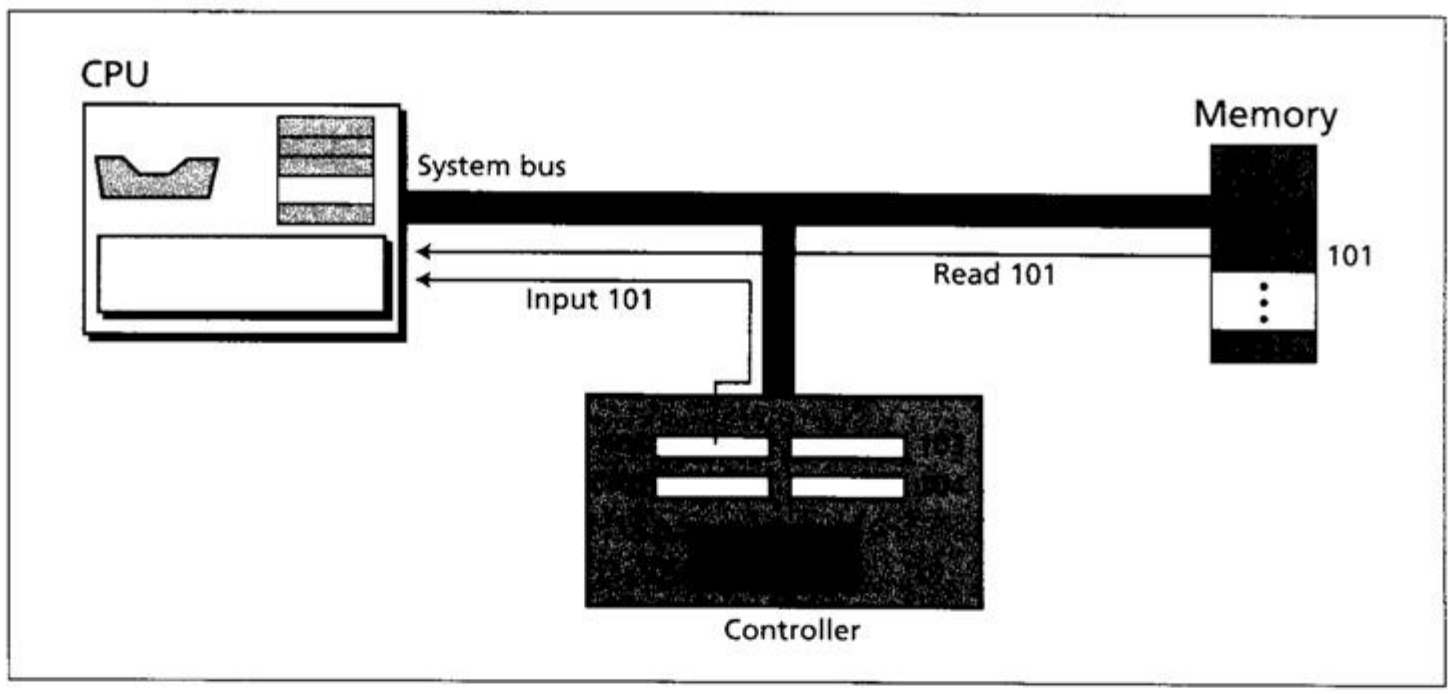


Figure 5.17 Isolated I/O addressing

nory-
ped I/O

In the **memory-mapped I/O** method, the CPU treats each register in the I/O controller as a word in memory. In other words, the CPU does not have separate instructions for transferring data from memory and I/O devices. For example, there is only one "Read" instruction. If the address defines a word from memory, the data is read from that word. If the address defines a register from an I/O device, the data is read from that register. The advantage of the memory-mapped configuration is a smaller number of instructions: all the memory instructions can be used by I/O devices. The disadvantage is that part of the memory address space is allocated to registers in I/O controllers. For example, if we have five I/O controllers and each has four registers, 20 addresses are used for this purpose. The size of the memory is reduced by 20 words. Figure 5.18 shows the memory-mapped I/O concept.

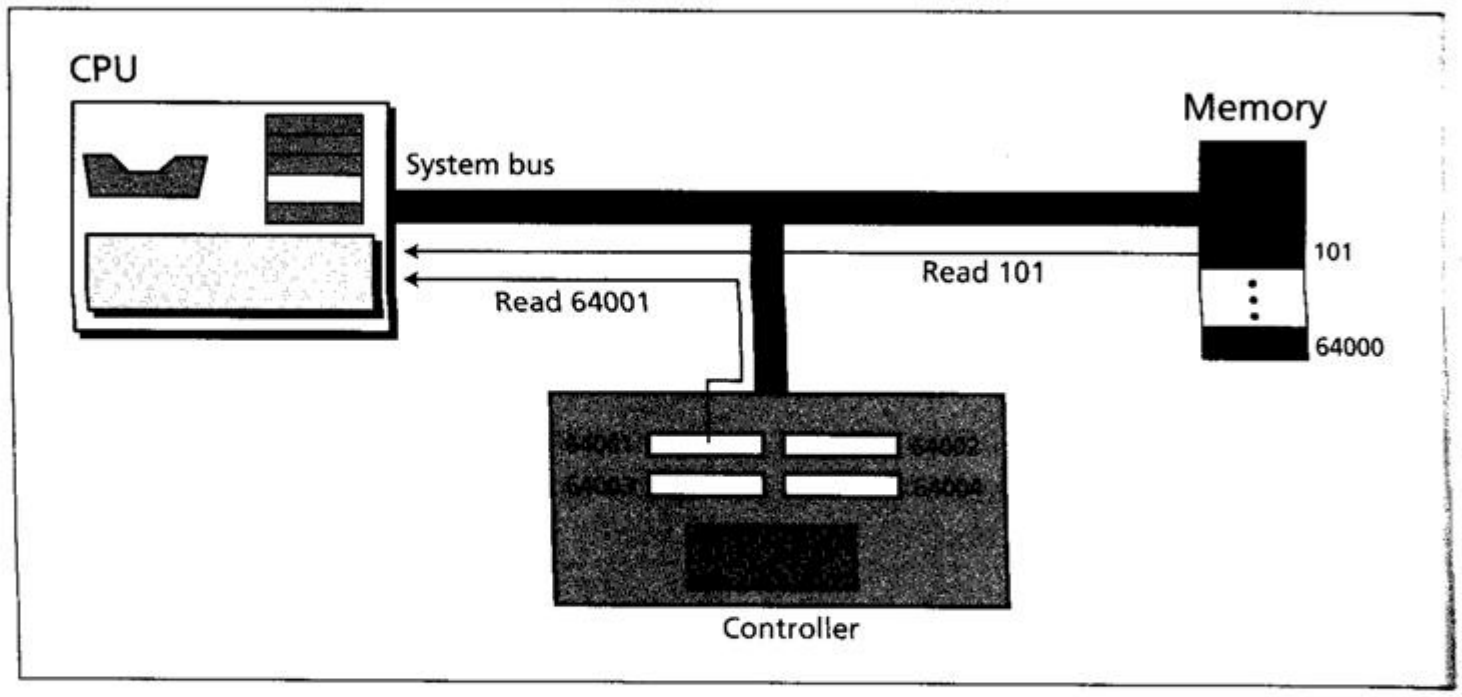


Figure 5.18 Memory-mapped I/O addressing

5.5 PROGRAM EXECUTION

Today, general-purpose computers use a set of instructions called a *program* to process data. A computer executes the program to create output data from input data. Both the program and the data are stored in memory.

At the end of this chapter we give some examples of how a hypothetical simple computer executes a program.

Machine cycle

The CPU uses repeating **machine cycles** to execute instructions in the program, one by one, from beginning to end. A simplified cycle can consist of three phases: *fetch*, *decode*, and *execute* (Figure 5.19).

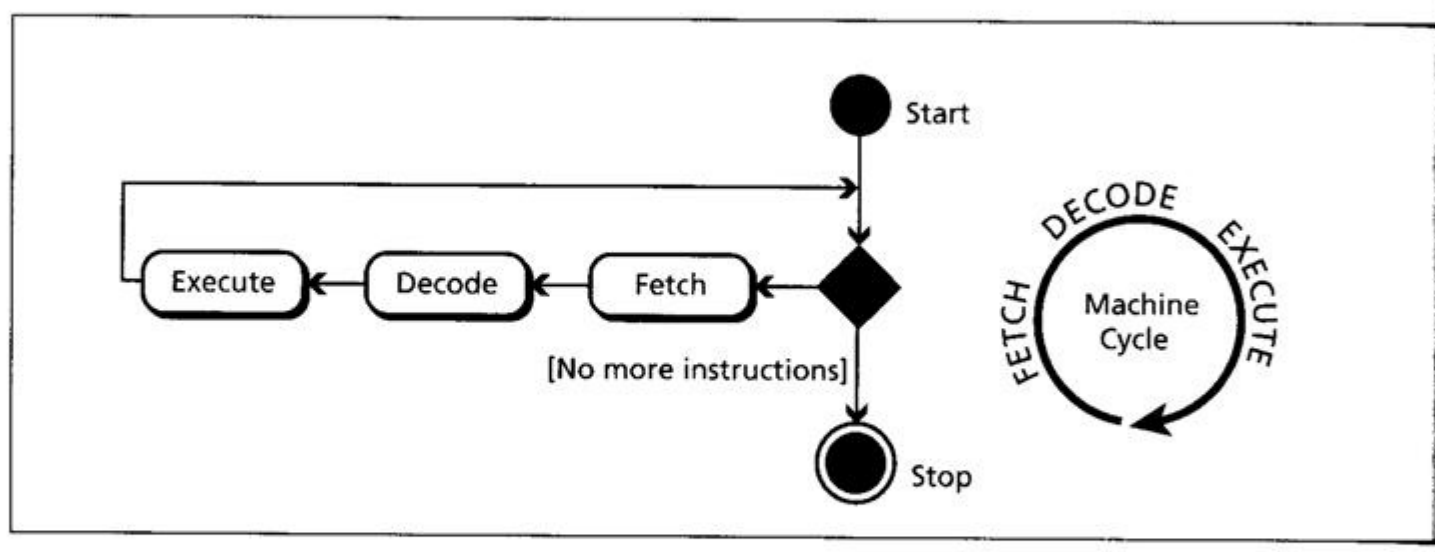


Figure 5.19 The steps of a cycle

In the **fetch** phase, the control unit orders the system to copy the next instruction into the instruction register in the CPU. The address of the instruction to be copied is held in the program counter register. After copying, the program counter is incremented to refer to the next instruction in memory.

The second phase in the cycle is the **decode** phase. When the instruction is in the instruction register, it is decoded by the control unit. The result of this decode step is the binary code for some operation that the system will perform.

After the instruction is decoded, the control unit sends the task order to a component in the CPU. For example, the control unit can tell the system to load (read) a data item from memory, or the CPU can tell the ALU to add the contents of two input registers and put the result in an output register. This is the **execute** phase.

Input/output operation

Commands are required to transfer data from I/O devices to the CPU and memory. Because I/O devices operate at much slower speeds than the CPU, the operation of the CPU must be synchronized with the I/O devices somehow. Three methods have been devised for this synchronization: programmed I/O, interrupt-driven I/O, and direct memory access (DMA).

In the **programmed I/O** method, synchronization is very primitive: the CPU waits for the I/O device (Figure 5.20).

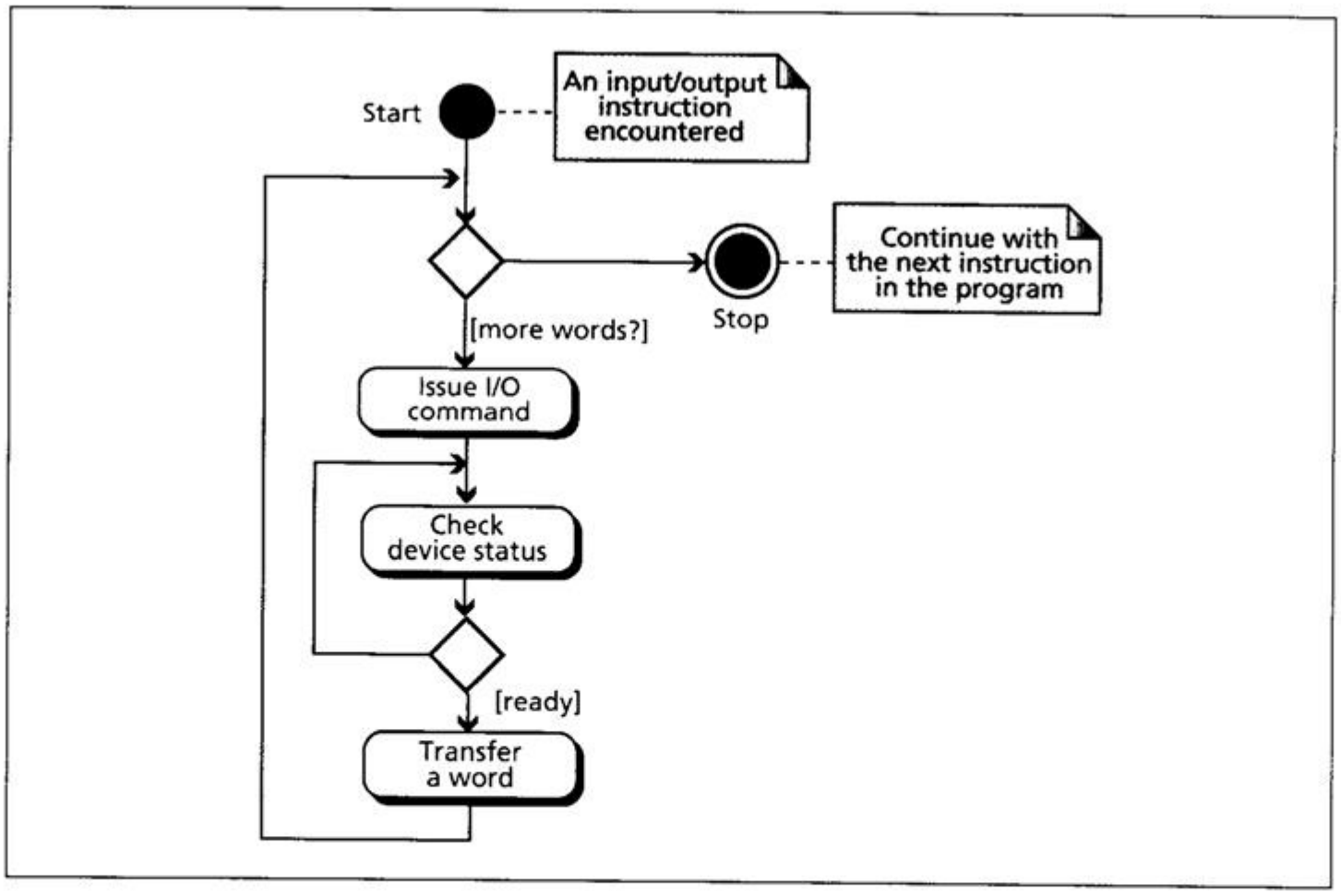


Figure 5.20 Programmed I/O

The transfer of data between the I/O device and the CPU is done by an instruction in the program. When the CPU encounters an I/O instruction, it does nothing else until the data transfer is complete. The CPU constantly checks the status of the I/O device: if the device is ready to transfer, data is transferred to the CPU. If the device is not ready, the CPU continues checking the device status until the I/O device is ready. The big issue here is that CPU time is wasted by checking the status of the I/O device for each unit of data to be transferred. Note that data is transferred to memory after the input operation, while data is transferred from memory before the output operation.

Interrupt-driven i/O

In the **interrupt-driven I/O** method, the CPU informs the I/O device that a transfer is going to happen, but it does not test the status of the I/O device continuously. The I/O device informs (interrupts) the CPU when it is ready. During this time, the CPU can do other jobs such as running other programs or transferring data from or to other I/O devices (Figure 5.21).

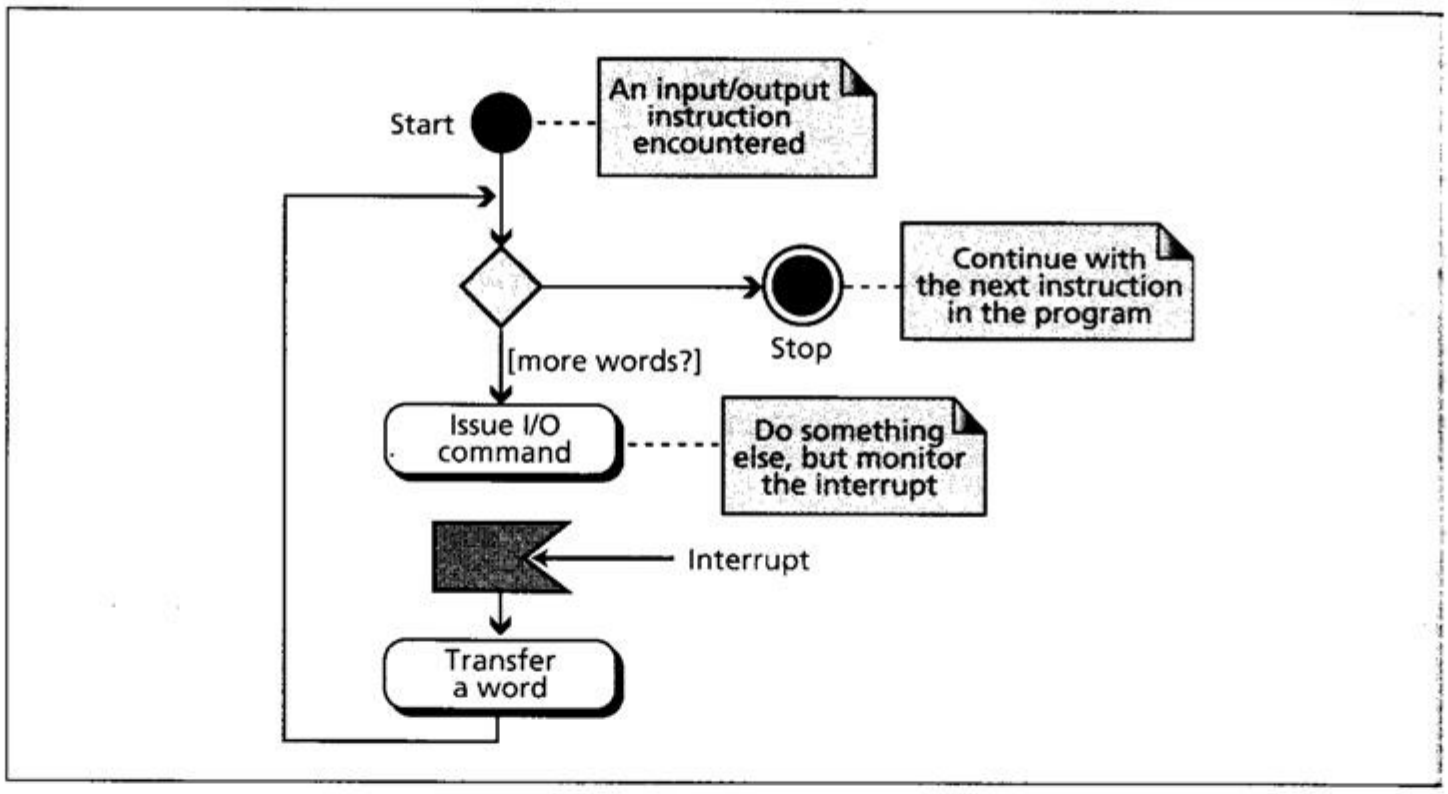


Figure 5.21 Interrupt-driven I/O

In this method, CPU time is not wasted—the CPU can do something else while the slow I/O device is finishing a task. Note that, like programmed I/O, this method also transfers data between the device and the CPU. Data is transferred to memory after the input operation. While data is transferred from memory before the output operation.

Direct memory access (DMA)

The third method used for transferring data is **direct memory access (DMA)**. This method transfers a large block of data between a high-speed I/O device, such as a disk, and memory directly without passing it through the CPU. This requires a DMA controller that relieves the CPU of some of its functions. The DMA controller has registers to hold a block of data before and after memory transfer. Figure 5.22 shows the DMA connection to the data, address and control buses.

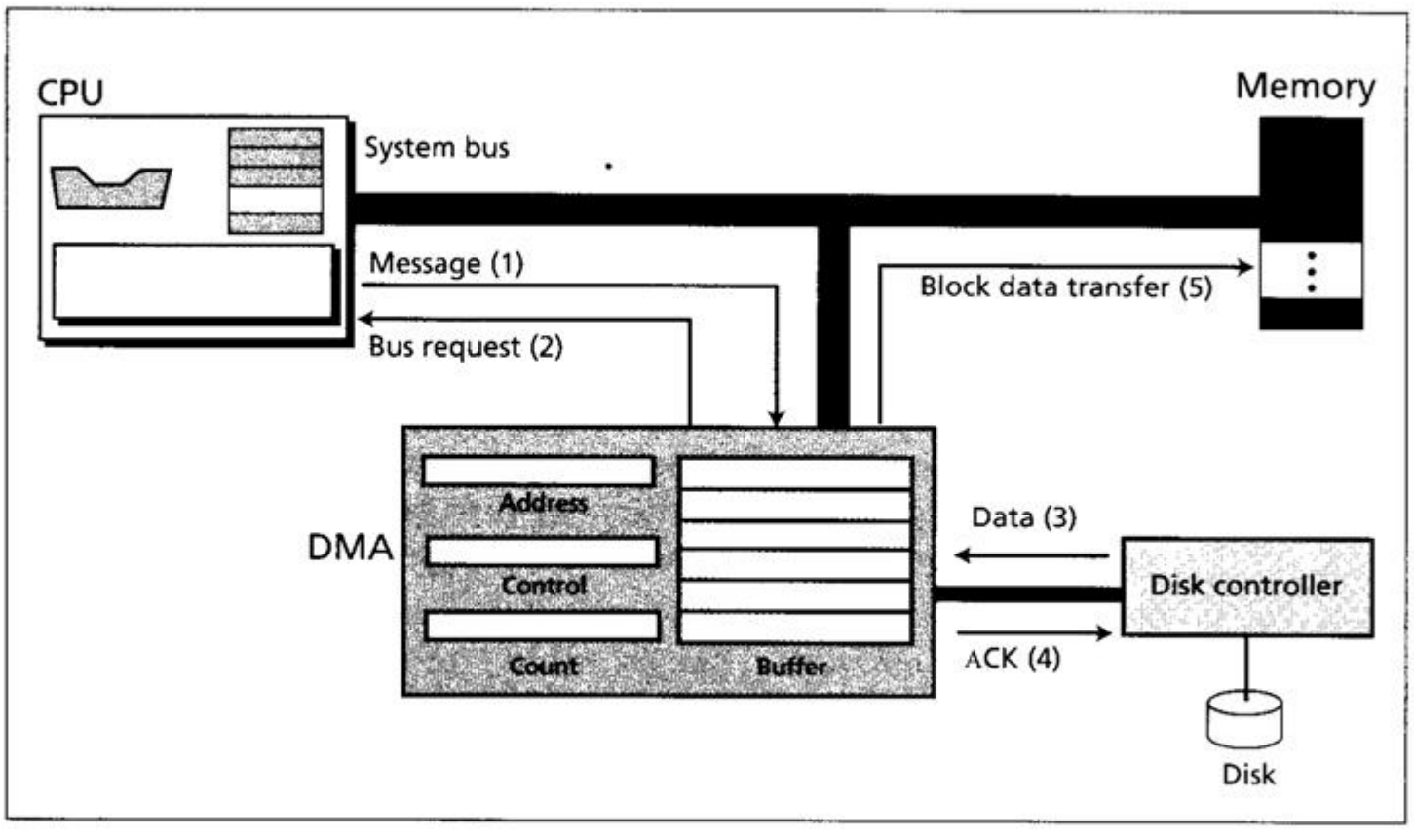


Figure 5.22 DMA connection to the general bus

Using this method for an I/O operation, the CPU sends a message to the DMA. The message contains the type of transfer (input or output), the start address of the memory location, and the number of bytes to be transferred. The CPU is then available for other jobs (Figure 5.23).

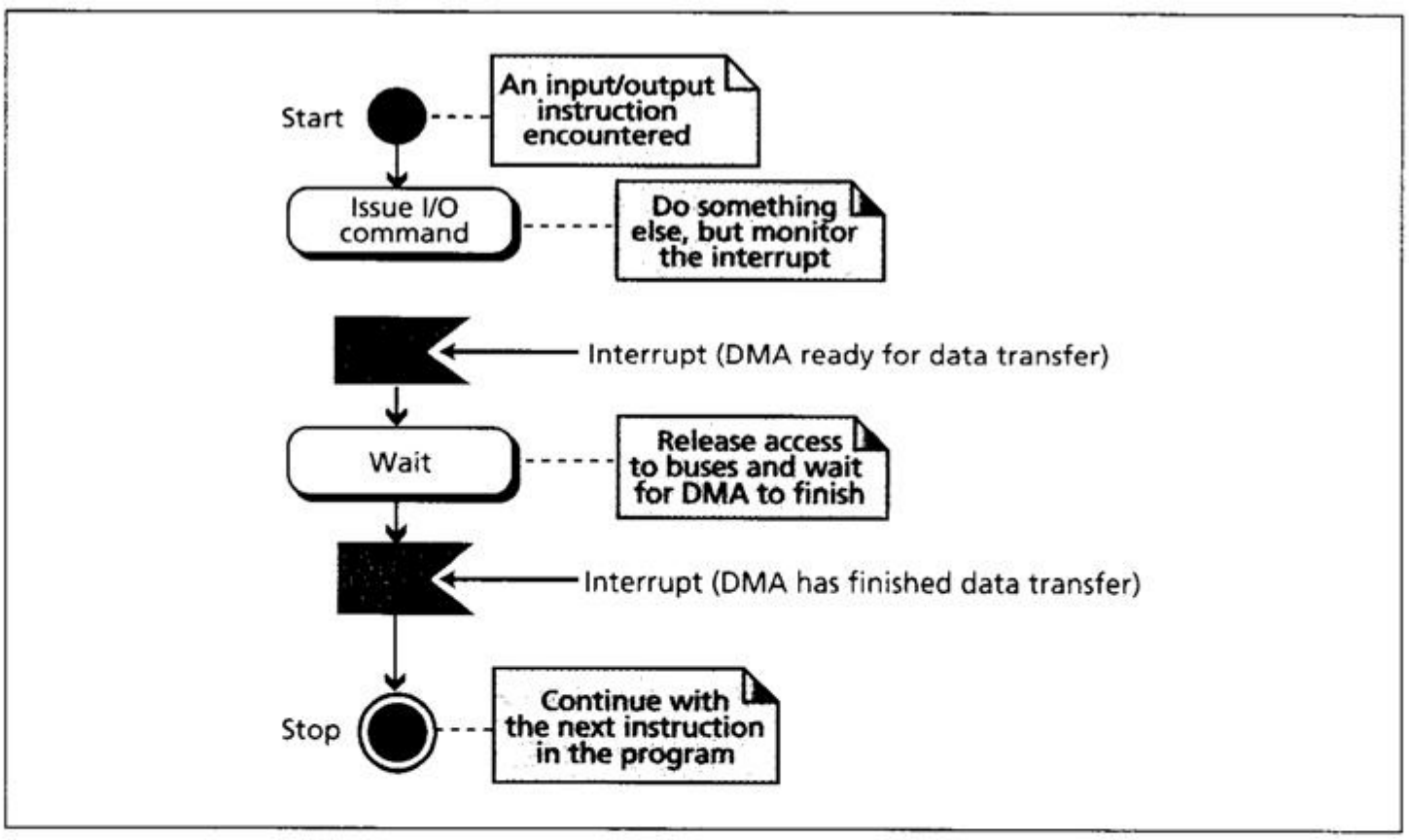


Figure 5.23 DMA input/output

When ready to transfer data, the DMA controller informs the CPU that it needs to take control of the buses. The CPU stops using the buses and lets the controller use

them. After data transfer directly between the DMA and memory, the CPU continues its normal operation. Note that, in this method, the CPU is idle for a time. However, the duration of this idle period is very short compared to other methods—the CPU is idle only during the data transfer between the DMA and memory, not while the device prepares the data.

5.6 DIFFERENT ARCHITECTURES

The architecture and organization of computers has gone through many changes in recent decades. In this section we discuss some common architectures and organizations that differ from the simple computer architecture we discussed earlier.

CISC

CISC (pronounced *sisk*) stands for **complex instruction set computer (CISC)**. The strategy behind CISC architectures is to have a large set of instructions, including complex ones. Programming CISC-based computers is easier than in other designs because there is a single instruction for both simple and complex tasks. Programmers therefore do not have to write a set of instructions to do a complex task.

The complexity of the instruction set makes the circuitry of the CPU and the control unit very complicated. The designers of CISC architectures have come up with a solution to reduce this complexity: programming is done on two levels. An instruction in machine language is not executed directly by the CPU—the CPU performs only simple operations, called *microoperations*. A complex instruction is transformed into a set of these simple operations and then executed by the CPU. This necessitates the addition of a special memory called *micromemory* that holds the set of operations for each complex instruction in the instruction set. The type of programming that uses microoperations is called *microprogramming*.

One objection to CISC architecture is the overhead associated with microprogramming and access to micromemory. However, proponents of the architecture argue that this compensates for smaller programs at the machine level. An example of CISC architecture can be seen in the Pentium series of processors developed by Intel.

RISC

RISC (pronounced *risk*) stands for **reduced instruction set computer**. The strategy behind RISC architecture is to have a small set of instructions that do a minimum number of simple operations. Complex instructions are simulated using a subset of simple instructions. Programming in RISC is more difficult and time-consuming than in the other design because most of the complex instructions are simulated using simple instructions.

Pipelining

We have learned that a computer uses three phases of *fetch*, *decode*, and *execute* for each instruction. In early computers, these three phases needed to be done in series for each instruction. In other words, instruction n needs to finish all of these phases before instruction $n + 1$ can start its own phases. Modern computers use a technique called **pipelining** to improve the **throughput** (the total number of instructions performed in each period of time). The idea is that if the control unit can do two or three of these phases simultaneously, the next instruction can start before the previous one is finished. Figure 5.24.a shows how three consecutive instructions are handled in a computer that uses no pipelining. Figure 5.24.b shows how pipelining can increase the throughput of the computer by allowing different types of phases belonging to different instructions to be done simultaneously. In other words, when the CPU is performing the decode phase of the first instruction, it can also perform the fetch phase of the second instruction. The first computer can perform on average nine phases in the specific period of time, while the pipelined computer can perform 24 phases in the same period of time. If we assume that each phase uses the same amount of time, the first computer has done 9/3 = 3 instructions while the second computer has done 24/3 = 8 instructions. The throughput is therefore increased 8/3 or 266 percent.

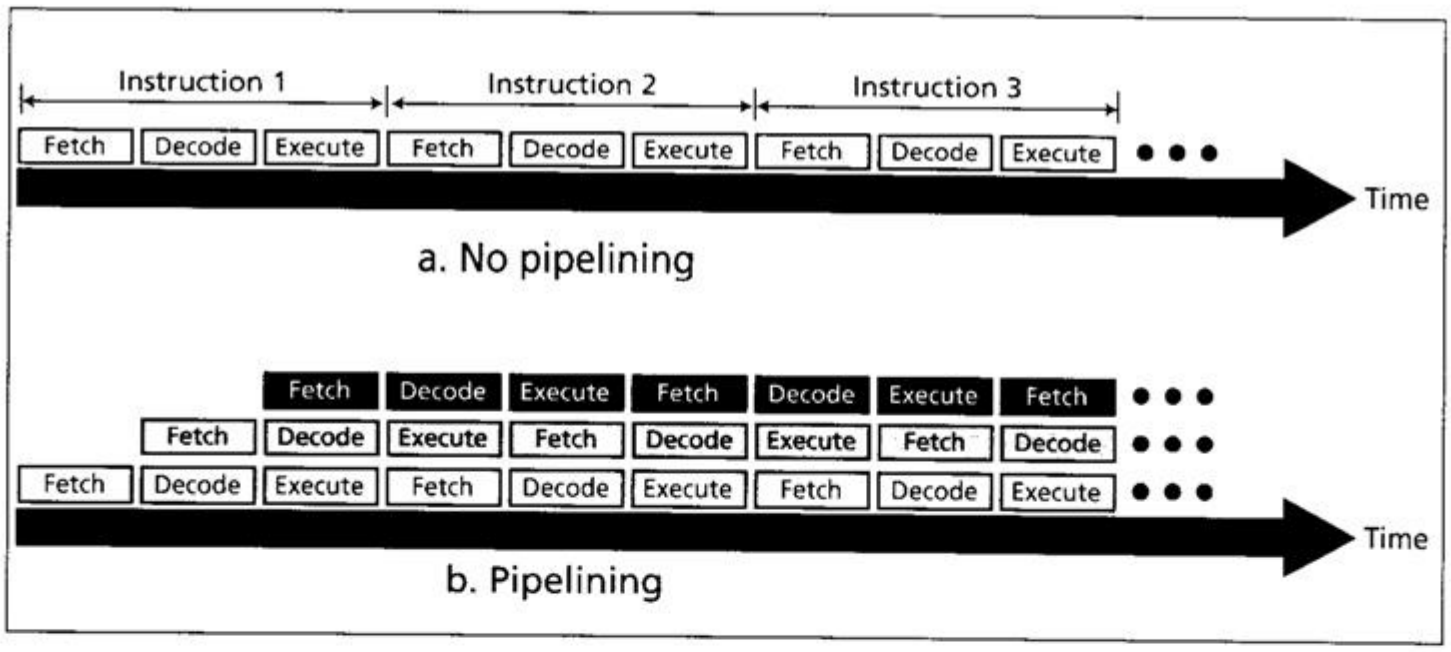


Figure 5.24 Pipelining

Of course, pipelining is not as easy as this. There are some problems, such as when a jump instruction is encountered. In this case, the instruction in the *pipe* should be discarded. However, new CPU designs have overcome most drawbacks. Some new CPU designs can even do several fetch cycles simultaneously.

Parallel processing

Traditionally a computer had a single control unit, a single arithmetic logic unit, and a single memory unit. With the evolution in technology and the drop in the cost of computer hardware, today we can have a single computer with multiple control units, multiple arithmetic logic units and multiple memory units. This idea is

referred to as *parallel processing*. Like pipelining, parallel processing can improve throughput.

Parallel processing involves many different techniques. A general view of parallel processing is given by the taxonomy proposed by M. J. Flynn. This taxonomy divides the computer's organization (in terms of processing data) into four categories, as shown in Figure 5.25. According to Flynn, parallel processing may occur in the data stream, the instruction stream, or both.

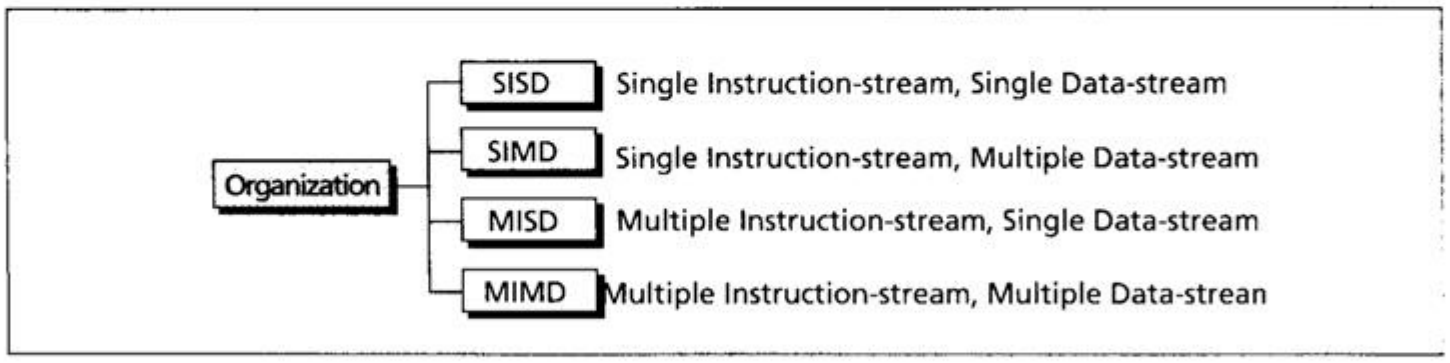


Figure 5.25 A taxonomy of computer organization

SISD organization

A **single instruction-stream, single data-stream (SISD)** organization represents a computer that has one control unit, one arithmetic logic unit, and one memory unit. The instructions are executed sequentially and each instruction may access one or more data item in the data stream. Our simple computer introduced earlier in the chapter is an example of SISD organization. Figure 5.26 shows the concept of configuration for an SISD organization.

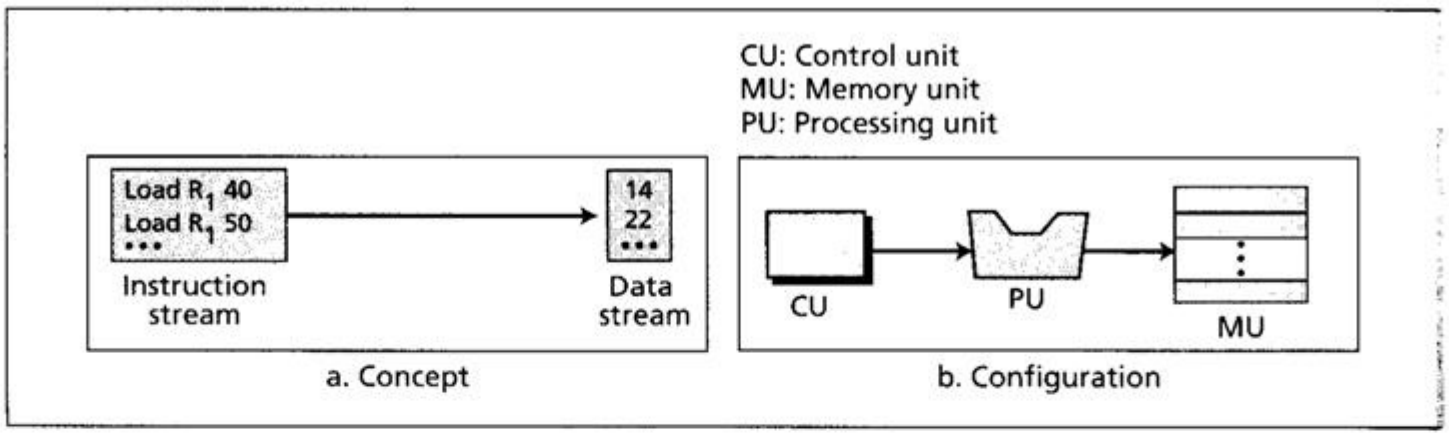


Figure 5.26 SISD organization

SIMD organization

A **single instruction-stream, multiple data-stream (SIMD)** organization represents a computer that has one control unit, multiple processing units, and one memory unit. All processor units receive the same instruction from the control unit, but operate on different items of data. An array processor that simultaneously operates on an array of data belongs to this category. Figure 5.27 shows the concept and implementation of an SIMD organization.

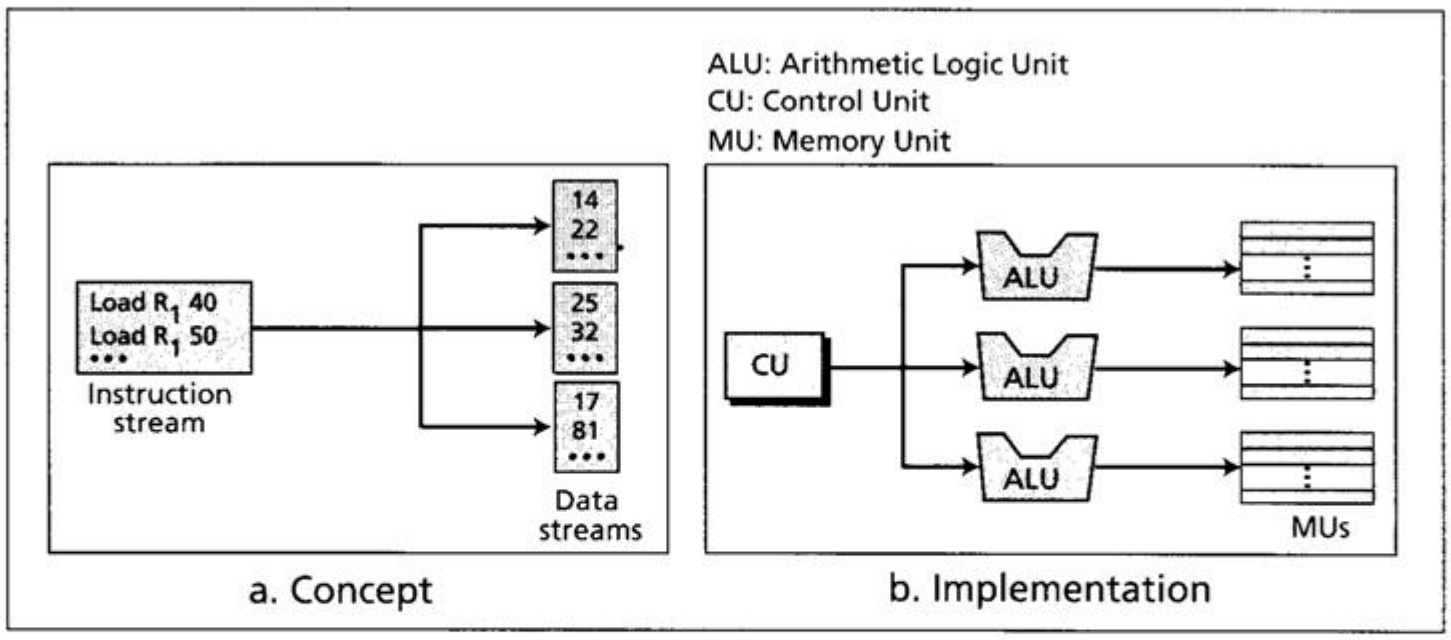


Figure 5.27 SIMD organization

MISD organization

A **multiple instruction-stream, single data-stream (MISD)** architecture is one in which several instructions belonging to several instruction streams simultaneously operate on the same data stream. Figure 5.28 shows the concept, but it has never been implemented.

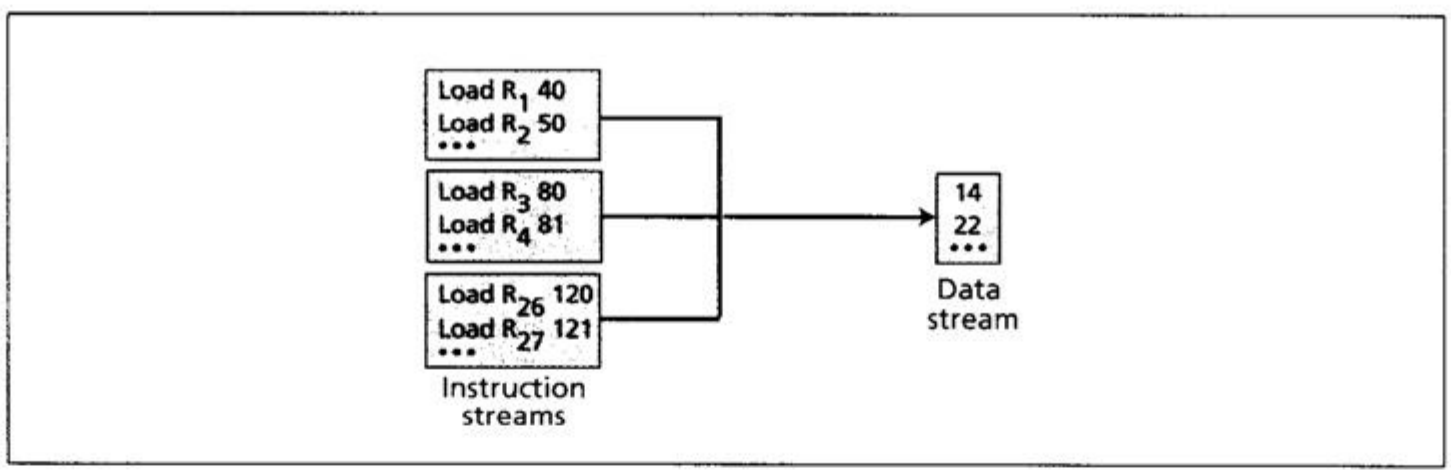


Figure 5.28 MISD organization

MIMD organization

A **multiple instruction-stream, multiple data-stream (MIMD)** architecture is one in which several instructions belonging to several instruction streams simultaneously operate on several data streams (each instruction on one data stream). Figure 5.29 shows the concept and implementation. MIMD organization is considered as a true parallel processing architecture by some experts. In this architecture several tasks can be performed simultaneously. The architecture can use a single shared memory or multiple memory sections.

Parallel processing has found some applications, mostly in the scientific community, in which a task may take several hours or days if done using a traditional computer architecture. Some examples of this can be found in multiplication of very large matrices, in simultaneous processing of large amounts of data for weather prediction, or in space flight simulations.

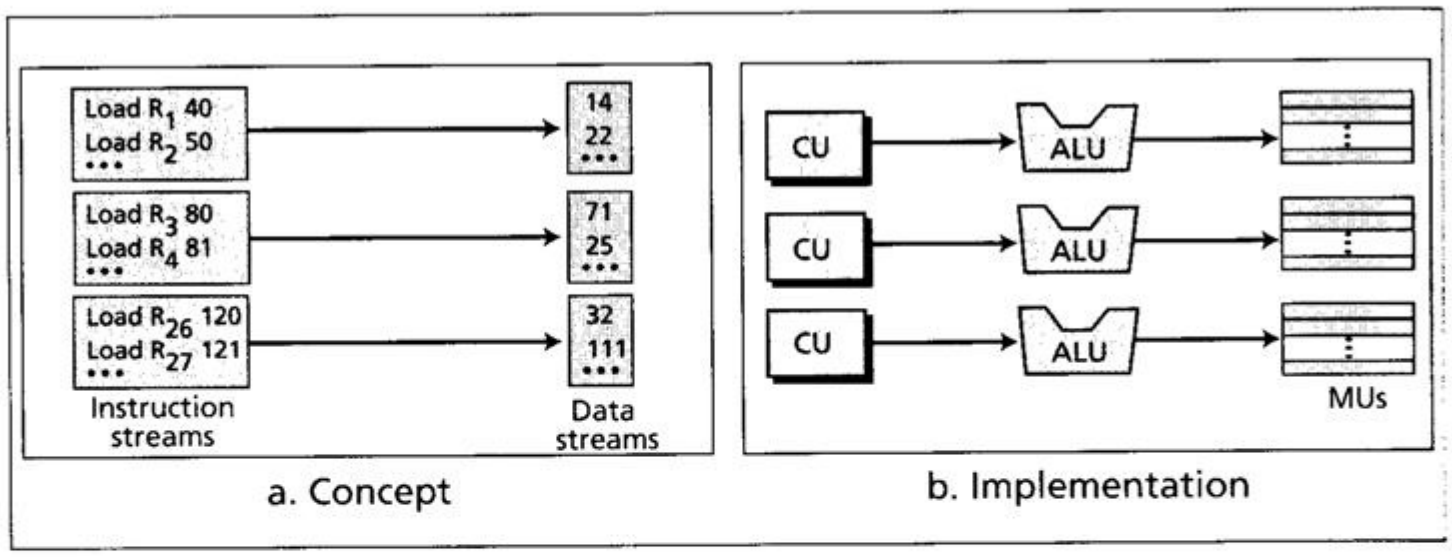


Figure 5.29 MIMD organization

5.7 A SIMPLE COMPUTER

To explain the architecture of computers as well as their instruction processing, we introduce a simple (unrealistic) computer, as shown in Figure 5.30. Our simple computer has three components: CPU, memory, and an input/output subsystem.

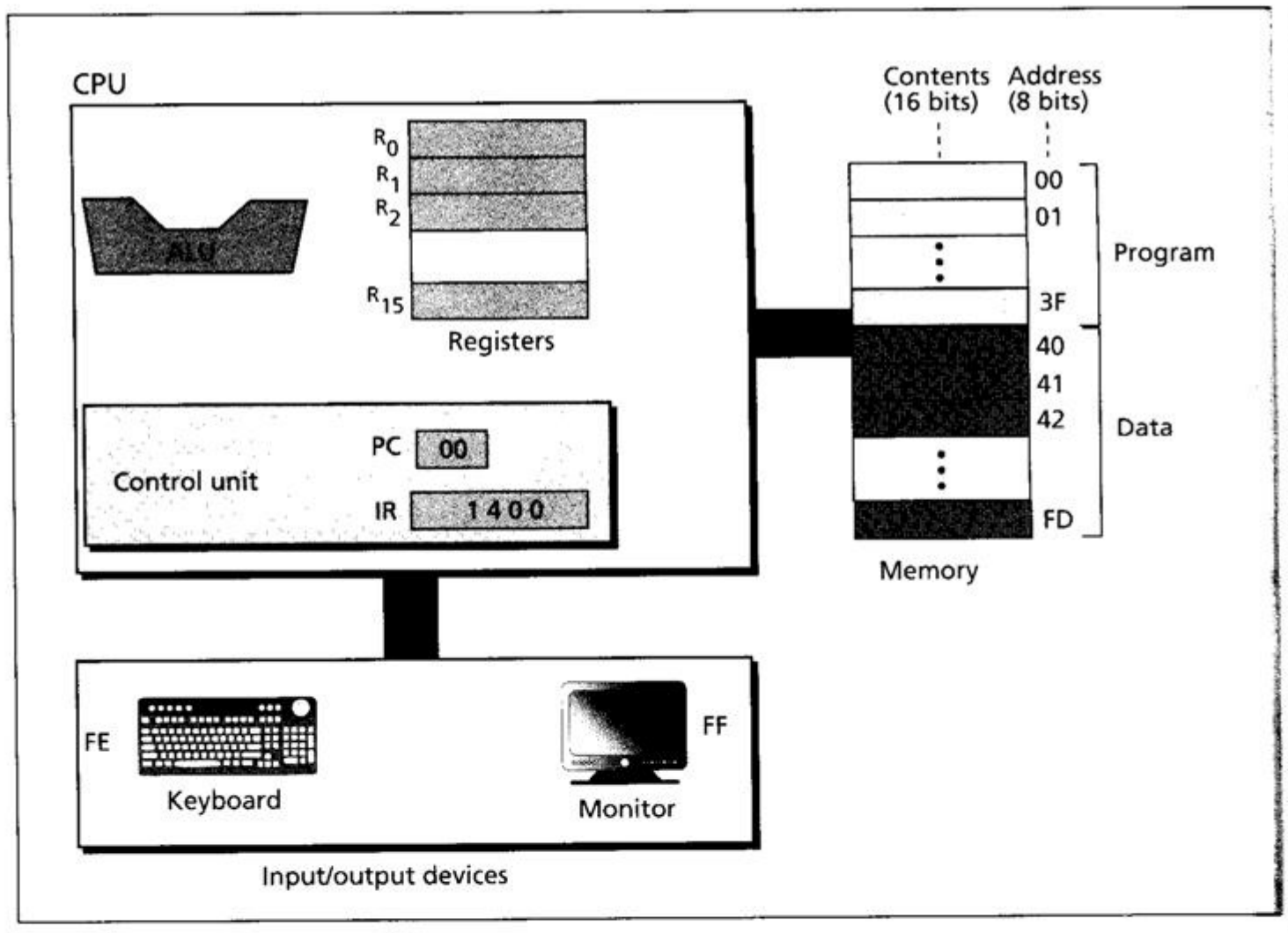


Figure 5.30 The components of a simple computer

CPU

The CPU itself is divided into three sections: data registers, arithmetic logic unit (ALU), and the control unit.

Data registers

There are sixteen 16-bit data registers with hexadecimal addresses $(0, 1, 2, \ldots, F)_{16}$, but we refer to them as R_0 to R_{15}. In most instructions, they hold 16-bit data, but in some instructions they may hold other information.

Control unit

The control unit has the circuitry to control the operations of the ALU, access to memory, and access to the I/O subsystem. In addition, it has two dedicated registers: program counter and instruction register. The program counter (PC), which can hold only eight bits, keeps track of which instruction is to be executed next. The contents of the PC points to the address of the memory location of the main memory that holds the next program instruction. After each machine cycle the program counter is incremented by one to point to the next program instruction. The instruction register (IR) holds a 16-bit value which is the encoded instruction for the current cycle.

Main memory

The main memory has 256 16-bit memory locations with binary addresses $(00000000 \text{ to } 11111101)_2$ or hexadecimal addresses $(00 \text{ to } FD)_{16}$. The main memory holds both data and program instructions. The first sixty-four locations $(00 \text{ to } 3F)_{16}$ are dedicated to program instructions. Program instructions for any program are stored in consecutive memory locations. Memory locations $(40 \text{ to } FD)_{16}$ are used for storing data.

Input/output subsystem

Our simple computer has a very primitive input/output subsystem. The subsystem consists of a keyboard and a monitor. Although we show the keyboard and monitor in a separate box in Figure 5.30, the subsystem is part of the memory address-wise. These devices have memory-mapped addresses, as discussed earlier in the chapter. We assume that the keyboard (as the input device) and monitor (as the only output device) act like memory locations with addresses $(FE)_{16}$ and $(FF)_{16}$ respectively, as shown in the figure. In other words, we assume that they behave as 16-bit registers that interact with the CPU as a memory location would. These two devices transfer data from the outside world to the CPU and vice versa.

Instruction set

Our simple computer is capable of having a set of sixteen instructions, although we are using only fourteen of these instructions. Each computer instruction consists of two parts: the operation code (*opcode*) and the *operand(s)*. The opcode specifies the type of operation to be performed on the operand(s). Each instruction consists of sixteen bits divided into four 4-bit fields. The leftmost field contains the opcode and the other three fields contains the operand or address of operand(s), as shown in Figure 5.31. The instruction are listed in Table 5.4 on page 131.

Note that not every instruction requires three operands. Any operand field not needed is filled with $(0)_{16}$. For example, all three operand fields of the halt instruction, and the last field of the move and NOT instructions, are filled with $(0)_{16}$. Also note that a register address is described by a single hexadecimal digit and thus uses a single field, but a memory location is described by two hexadecimal digits and uses two fields.

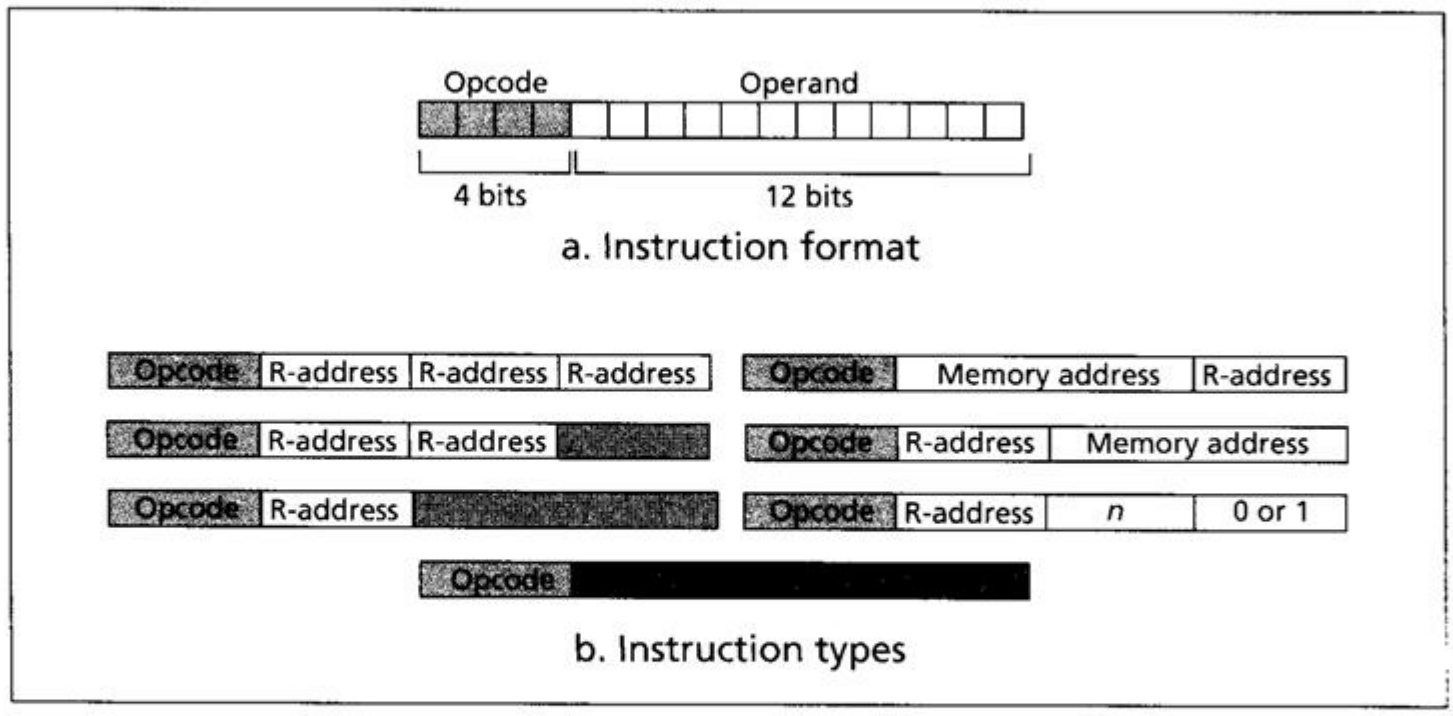


Figure 5.31 Format and different instruction types

There are two add instructions: one for adding integers (ADDI) and one for adding floating point numbers (ADDF). The simple computer can take input from a keyboard if we use address $(FE)_{16}$ as the second operand of the LOAD instruction. Similarly, the computer sends output to the monitor if we use the address $(FF)_{16}$ as the first operand of the STORE instruction. If the third operand of the ROTATE instruction is 0, the instruction circularly rotates the bit pattern in R to the right ***n*** places: if the third operand is 1, it rotates it to the left. We have also included one increment (INC) and one decrement (DEC) instruction.

Processing the instructions

Our simple computer, like most computers, uses machine cycles. A cycle is made of three phases: *fetch*, *decode*, and *execute*. During the *fetch* phase, the instruction whose address is determined by the PC is obtained from the memory and loaded into the IR. The PC is then incremented to point to the next instruction. During the *decode* phase, the instruction in IR is decoded and the required operands are fetched from the register or from memory. During the *execute* phase, the instruction is executed and the results are placed in the appropriate memory location or the register. Once the third phase is completed, the control unit starts the cycle again, but now the PC is pointing to the next instruction. The process continues until the CPU reaches a HALT instruction.

Table 5.4 List of instructions for the simple computer

Instruction	*Code*	*Operands*			*Action*
	d_1	d_2	d_3	d_4	
HALT	0				Stops the execution of the program
LOAD	1	R_D	M_S		$R_D \leftarrow M_S$
STORE	2	M_D		R_S	$M_D \leftarrow R_S$
ADDI	3	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1} + R_{S2}$
ADDF	4	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1} + R_{S2}$
MOVE	5	R_D	R_S		$R_D \leftarrow R_S$
NOT	6	R_D	R_S		$R_D \leftarrow \overline{R}_S$
AND	7	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1}$ AND R_{S2}
OR	8	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1}$ OR R_{S2}
XOR	9	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1}$ XOR R_{S2}
INC	A	R			$R \leftarrow R + 1$
DEC	B	R			$R \leftarrow R - 1$
ROTATE	C	R	n	0 or 1	Rot_n R
JUMP	D	R	n		IF $R_0 \neq R$ then PC = *n*, otherwise continue

Key: R_S, R_{S1}, R_{S2}: Hexadecimal address of source registers
R_D: Hexadecimal address of destination register
M_S: Hexadecimal address of source memory location
M_D: Hexadecimal address of destination memory location
n: hexadecimal number
d_1, d_2, d_3, d_4: First, second, third, and fourth hexadecimal digits

An example

Let us show how our simple computer can add two integers A and B and create the result as C. We assume that integers are in two's complement format. Mathematically, we show this operation as:

$$C = A + B$$

To solve this problem with the simple computer, it is necessary for the first two integers to be held in two registers (for example, R_0 and R_1) and the result of the operation to be held in a third register (for example R_2). The ALU can only operate on the data that is stored in data registers in the CPU. However, most computers, including our simple computer, have a limited number of registers in the CPU. If the number of data items is large and they are supposed to stay in the computer for the duration of the program, it is better to store them in memory and only bring them to the registers temporarily. So we assume that the first two integers are stored in memory locations $(40)_{16}$ and $(41)_{16}$ and the result should be stored in memory location $(42)_{16}$. This means that two integers need to be loaded into the CPU and the result needs to be stored in the memory. Therefore, a simple program to do the simple addition needs five instructions, as shown below:

1. Load the contents of M_{40} into register R_0 ($R_0 \leftarrow M_{40}$).
2. Load the contents of M_{41} into register R_1 ($R_1 \leftarrow M_{41}$).
3. Add the contents of R_0 and R_1 and place the result in R_2 ($R_2 \leftarrow R_0 + R_1$).
4. Store the contents R_2 in M_{42} ($M_{42} \leftarrow R_2$).
5. Halt.

In the language of our simple computer, these five instructions are encoded as:

Code	*Interpretation*			
$(1040)_{16}$	1: LOAD	0: R_0	40: M_{40}	
$(1141)_{16}$	1: LOAD	1: R_1	41: M_{41}	
$(3201)_{16}$	3: ADDI	2: R_2	0: R_0	1: R_1
$(2422)_{16}$	2: STORE	42: M_{42}		2: R_2
$(0000)_{16}$	0: HALT			

Storing program and data

To follow the von Neumann model, we need to store the program and the data in memory. We can store the five-line program in memory starting from location $(00)_{16}$ to $(04)_{16}$. We already know that the data needs to be stored in memory locations $(40)_{16}$, $(41)_{16}$, and $(42)_{16}$.

Cycles

Our computer uses one cycle per instruction. If we have a small program with five instructions, we need five cycles. We also know that each cycle is normally made up of three steps: *fetch*, *decode*, *execute*. Assume for the moment that we need to add 161 + 254 = 415. The numbers are shown in memory in hexadecimal is, $(00A1)_{16}$, $(00FE)_{16}$, and $(019F)_{16}$.

Cycle 1

At the beginning of the first cycle (Figure 5.32), the PC points to the first instruction of the program, which is at memory location $(00)_{16}$. The control unit goes through three steps:

1. The control unit *fetches* the instruction stored in memory location $(00)_{16}$ and puts it in the IR. After this step, the value of the PC is incremented.
2. The control unit *decodes* the instruction $(1040)_{16}$ as $R_0 \leftarrow M_{40}$.
3. The control unit *executes* the instruction, which means that a copy of the integer stored in memory location $(40)_{16}$ is loaded into register R_0.

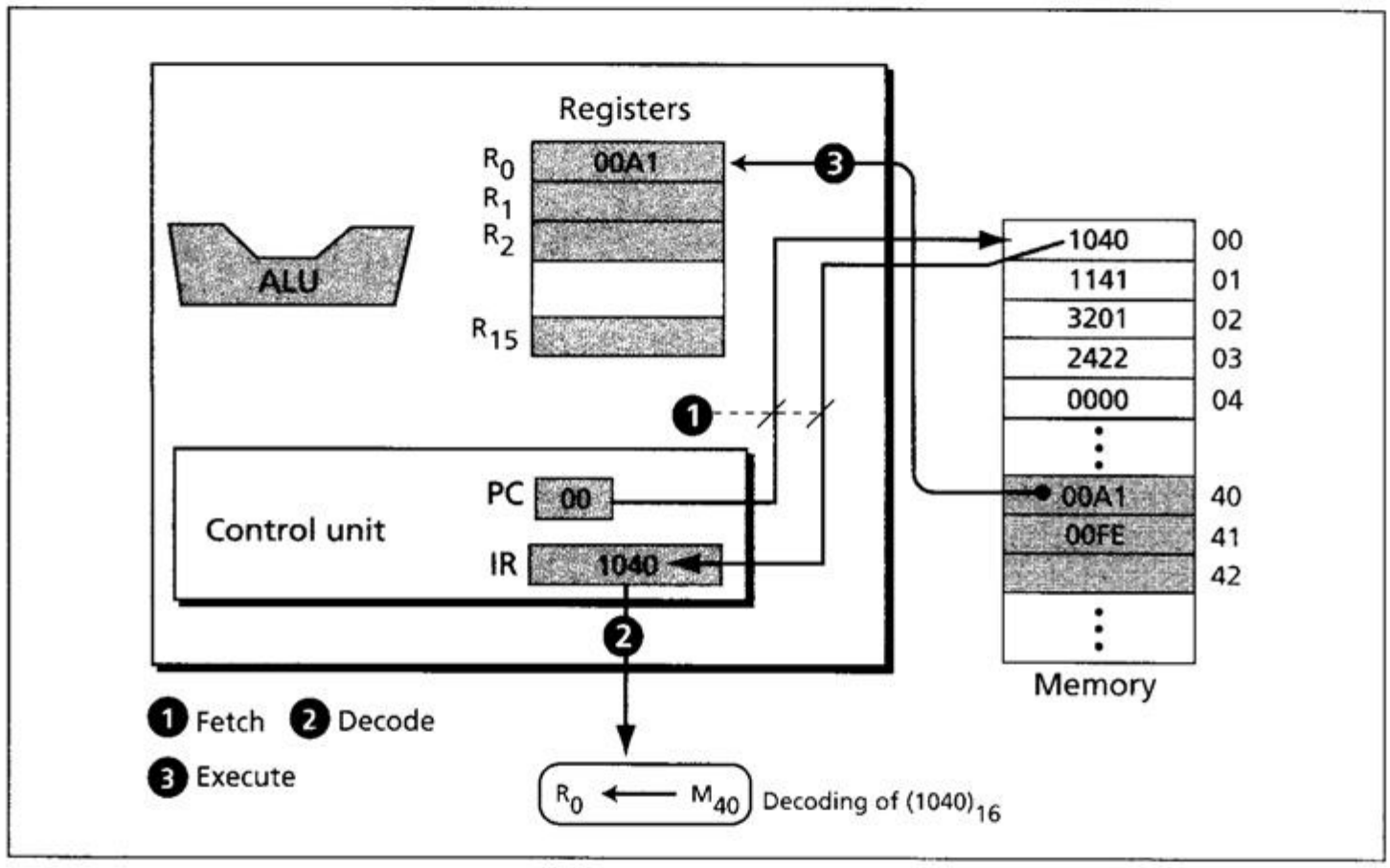


Figure 5.32 Status of cycle 1

Cycle 2

At the beginning of the second cycle (Figure 5.33), the PC points to the second instruction of the program, which is at memory location $(01)_{16}$. The control unit goes through three steps:

1. The control unit *fetches* the instruction stored in memory location $(01)_{16}$ and puts it in the IR. After this step, the value of the PC is incremented.
2. The control unit *decodes* the instruction $(1141)_{16}$ as $R_1 \leftarrow M_{41}$.
3. The control unit *executes* the instruction, which means that a copy of integer stored in memory location $(41)_{16}$ is loaded into register R_1.

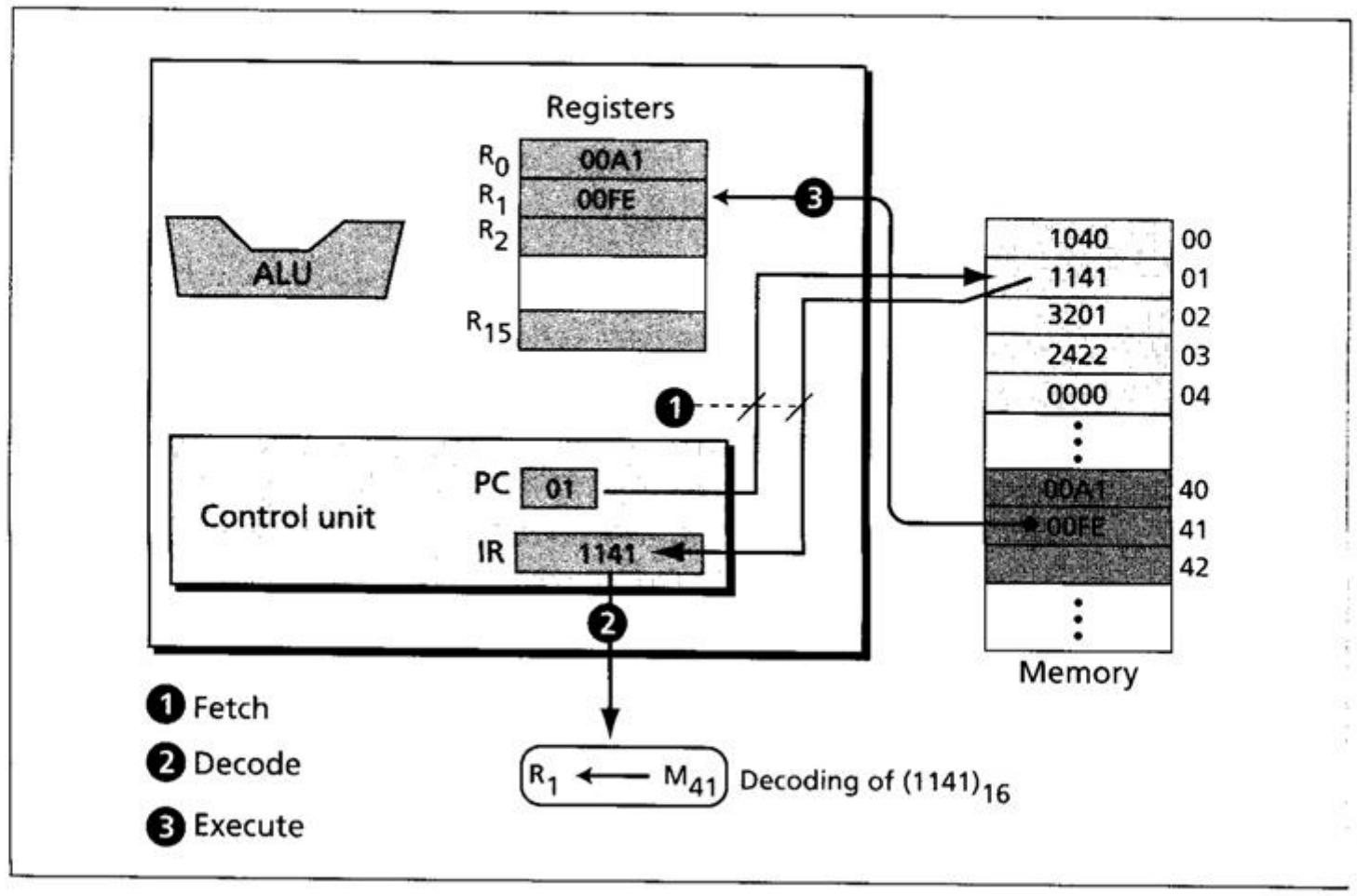


Figure 5.33 Status of cycle 2

Cycle 3

At the beginning of the third cycle (Figure 5.34), the PC points to the third instruction of the program, which is at memory location $(02)_{16}$. The control unit goes through three steps:

1. The control unit *fetches* the instruction stored in memory location $(02)_{16}$ and puts it in the IR. After this step, the value of the PC is incremented.
2. The control unit *decodes* the instruction $(3201)_{16}$ as $R_2 \leftarrow R_0 + R_1$.
3. The control unit *executes* the instruction, which means that the contents of R_0 is added to the content of R_1 (by the ALU) and the result is put in R_2.

Cycle 4

At the beginning of the fourth cycle (Figure 5.35), the PC points to the fourth instruction of the program, which is at memory location $(03)_{16}$. The control unit goes through three steps:

1. The control unit *fetches* the instruction stored in memory location $(03)_{16}$ and puts it in the IR. After this step, the value of the PC is incremented.
2. The control unit *decodes* the instruction $(2422)_{16}$ as $M_{42} \leftarrow R_2$.
3. The control unit *executes* the instruction, which means a copy of integer in register R_2 is stored in memory location $(42)_{16}$.

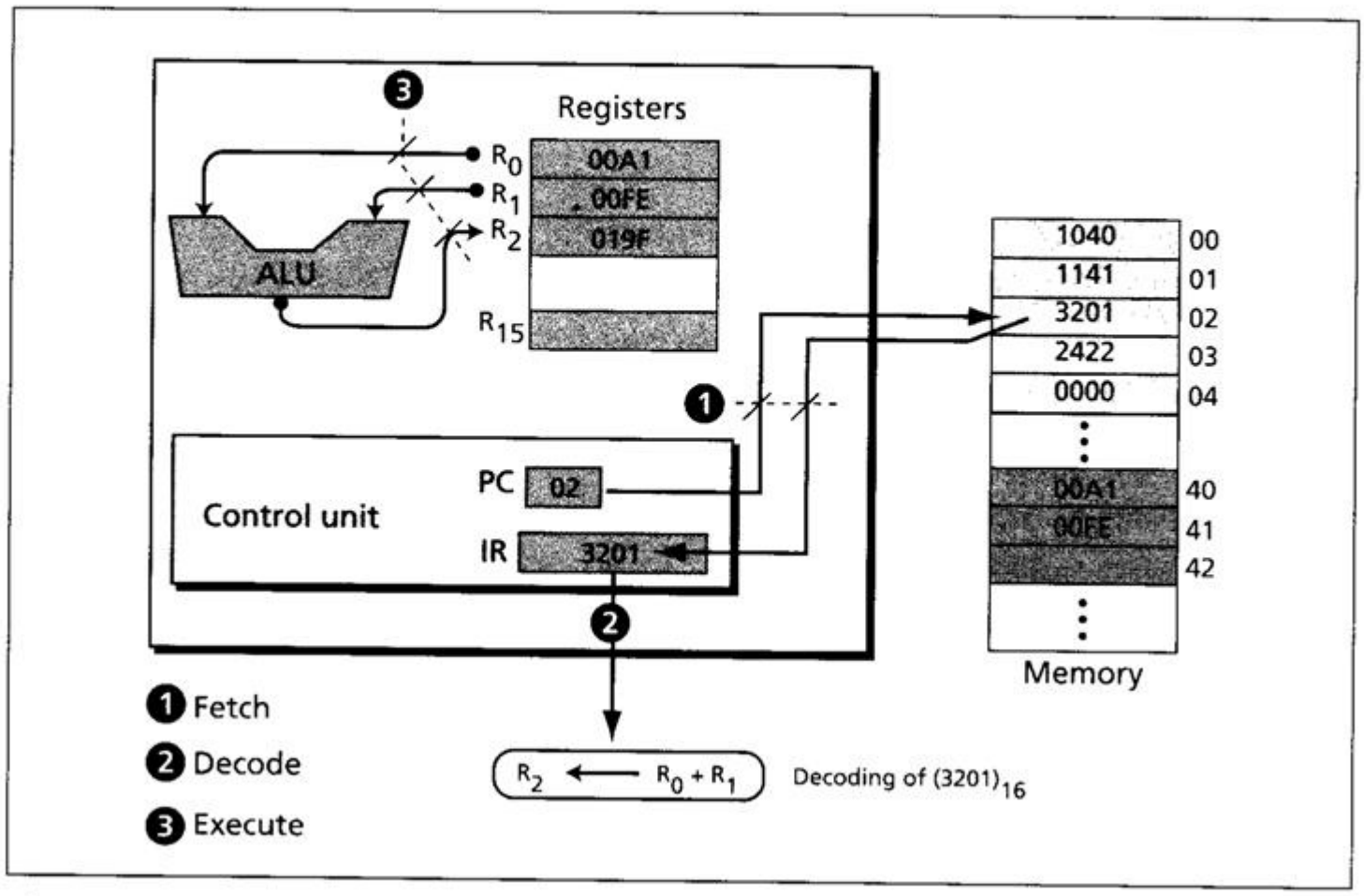


Figure 5.34 Status of cycle 3

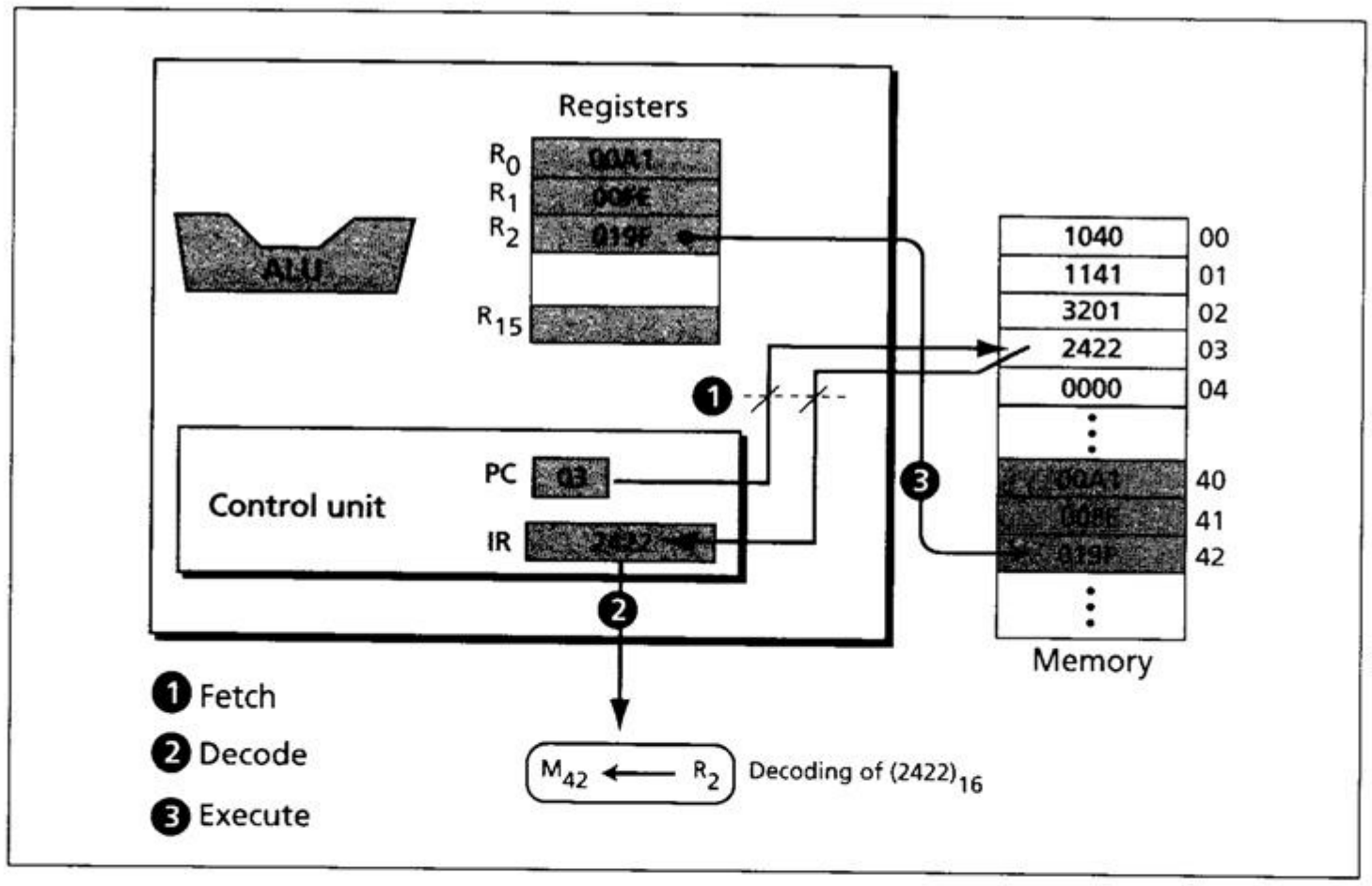


Figure 5.35 Status of cycle 4

Cycle 5

At the beginning of the fifth cycle (Figure 5.36), the PC points to the fifth instruction of the program, which is at memory location $(04)_{16}$. The control unit goes through three steps:

1. The control unit *fetches* the instruction stored in memory location $(04)_{16}$ and puts it in the IR. After this step, the value of the PC is incremented.
2. The control unit *decodes* the instruction $(0000)_{16}$ as Halt.
3. The control unit *executes* the instruction, which means that the computer stops.

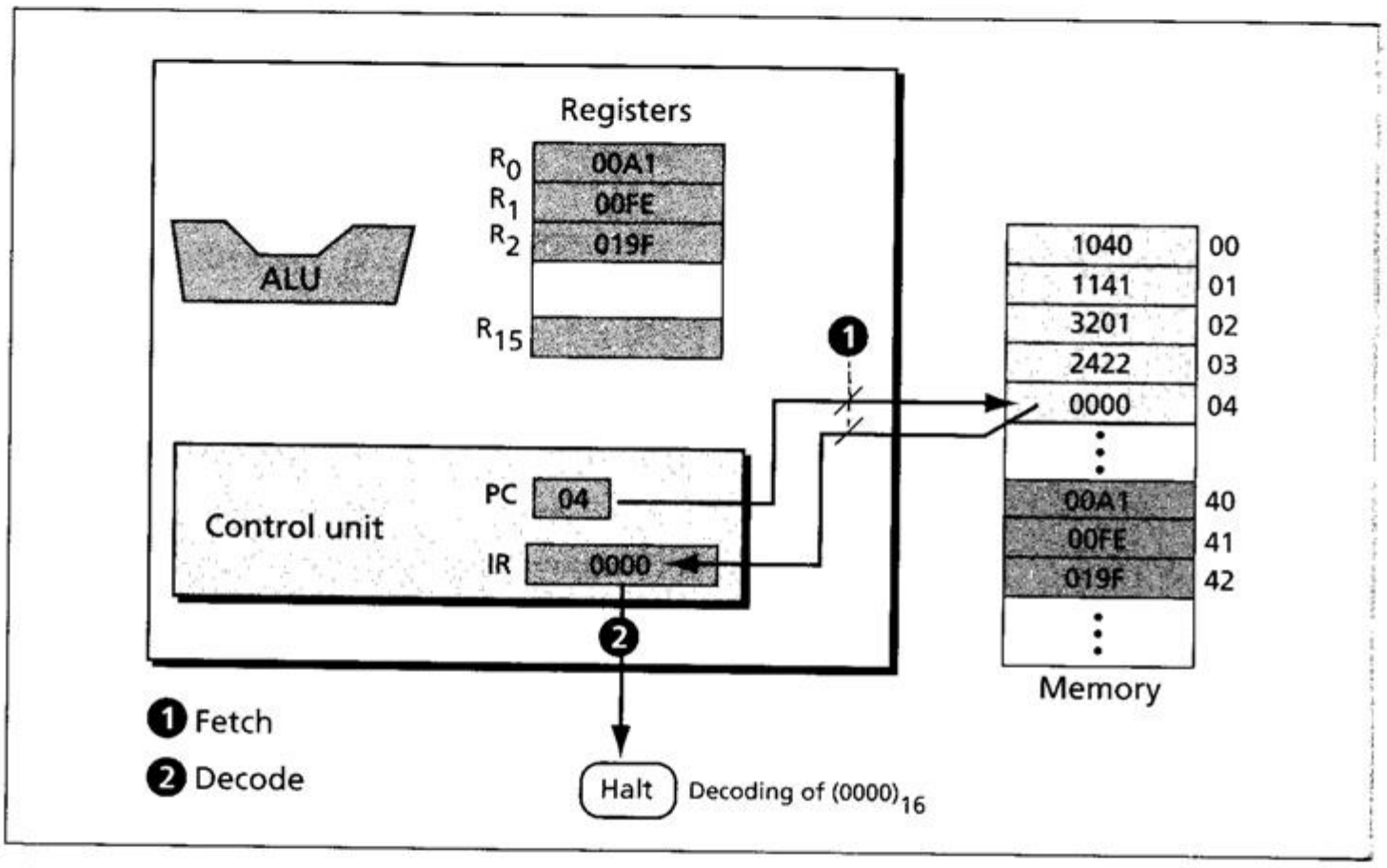


Figure 5.36 Status of cycle 5

Another example

In the previous example we assumed that the two integers to be added were already in memory. We also assumed that the result of addition will be held in memory. You may ask how we can store the two integers we want to add in memory, or how we use the result when it is stored in the memory. In a real situation, we enter the first two integers into memory using an input device such as keyboard, and we display the third integer through an output device such as a monitor. Getting data via an input device is normally called a *read* operation, while sending data to an output device is normally called a *write* operation. To make our previous program more practical, we need modify it as follows:

1. Read an integer into M_{40}.
2. $R_0 \leftarrow M_{40}$.
3. Read an integer into M_{41}.

4. $R_1 \leftarrow M_{41}$.
5. $R_2 \leftarrow R_0 + R_1$.
6. $M_{42} \leftarrow R_2$.
7. Write the integer from M_{42}.
8. Halt.

There are many ways to implement input and output. Most computers today do direct data transfer from an input device to memory and direct data transfer from memory to an output device. However, our simple computer is not one of them. In our computer we can simulate read and write operations using the LOAD and STORE instructions. Furthermore, LOAD and STORE read data input to the CPU and write data from the CPU. We need two instruction to read data into memory or write data out of memory. The read operation is:

$R \leftarrow M_{FE}$ Because the keyboard is assumed to be memory location $(FE)_{16}$
$M \leftarrow R$

The write operation is:

$R \leftarrow M$
$M_{FF} \leftarrow R$ Because the monitor is assumed to be memory location $(FF)_{16}$

You may ask why, if the operations are supposed to be done in the CPU, do we transfer the data from the keyboard to the CPU, then to the memory, then to the CPU for processing? Could we directly transfer data to the CPU? The answer is that we can do this for this small problem, but we should not do it in principle. Think what happens if we need to add 1000 numbers or sort 1,000,000 integers. The number of registers in the CPU is limited (it may be hundreds in a real computer, but still not enough).

The input operation must always read data from an input device into memory: the output operation must always write data from memory to an output device.

With this in mind, the program is coded as:

1	$(1FFE)_{16}$	5	$(1040)_{16}$	9	$(1F42)_{16}$
2	$(240F)_{16}$	6	$(1141)_{16}$	10	$(2FFF)_{16}$
3	$(1FFE)_{16}$	7	$(3201)_{16}$	11	$(0000)_{16}$
4	$(241F)_{16}$	8	$(2422)_{16}$		

Operations 1 to 4 are for input and operations 9 and 10 are for output. When we run this program, it waits for the user to input two integers on the keyboard and

press the enter key. The program then calculates the sum and displays the result on the monitor.

Reusability One of the advantages of a computer over a non-programmable calculator is that we can use the same program over and over. We can run the program several times and each time enter different inputs and obtain a different output.

5.8 RECOMMENDED READING

For more details about the subjects discussed in this chapter, the following books are recommended:

- Englander I: *The Architecture of Computer Hardware and Systems Software*, Hoboken, NJ: Wiley, 2003
- Mano M: *Computer System Architecture*, Upper Saddle River, NJ: Prentice Hall, 1993
- Null L and Lobur J: *Computer Organization and Architecture*, Sudbury, MA: Jones and Bartlett, 2003
- Hamacher C, Vranesic Z, and Zaky S: *Computer Organization*, New York: McGraw-Hill, 2002
- Warford S: *Computer Systems*, Sudbury, MA: Jones and Bartlett, 2005
- Ercegovac M, Lang T, and Moreno J: *Introduction to Digital Systems*, Hoboken, NJ: Wiley, 1998
- Cragon H: *Computer Architecture and Implementation*, Cambridge: Cambridge University Press, 2000
- Stallings W: *Computer Organization and Architecture*, Upper Saddle River, NJ: Prentice Hall, 2002

5.9 KEY TERMS

This chapter has introduced the following key terms, which are listed here with the pages on which they first occur:

address bus 115

address space 104

arithmetic logic unit (ALU) 102

bus 115

cache memory 107

central processing unit (CPU) 102

compact disk read-only memory (CD-ROM) 110

compact disk rewritable (CD-RW) 113

complex instruction set computer (CISC) 124

control bus 115

control unit 103

data bus 115

decode 121

digital versatile disk (DVD) 114

direct memory address (DMA) 122

dynamic RAM (DRAM) 106

electronically erasable programmable read-only memory (EEPROM) 106

erasable programmable read-only memory (EPROM) 106

execute 121

fetch 121

FireWire 116
input/output subsystem 108
interrupt-driven I/O 122
intertrack gap 109
keyboard 108
machine cycle 120
magnetic tape 109
master disk 111
monitor 108
multiple instruction-stream, single data-stream (MISD) 127
optical storage device 110
pipelining 125
polycarbonate resin 111
program counter 103
programmed I/O 121
read-only memory (ROM) 106
reduced instruction set computer (RISC) 124
rotational speed 109
seek time 109
small computer system interface (SCSI) 116
storage device 108
track 108
Universal Serial Bus (USB) 116
input/output controller 116
instruction register 103
intersector gap 109
isolated I/O 119
land 111
magnetic disk 108
main memory 103
memory mapped I/O 119
multiple instruction-stream, multiple data-stream (MIMD) 127
non-storage device 108
parallel processing 126
pit 111
printer 108
programmable read-only memory (PROM) 106
random access memory (RAM) 105
read/write head 108
register 103
sector 108
single instruction-stream, multiple data-stream (SIMD) 126
static RAM (SRAM) 105
throughput 125
transfer time 109
write once, read many (WORM) 112

5.10 SUMMARY

- The parts that make up a computer can be divided into three broad categories or subsystems: the central processing unit (CPU), the main memory, and the input/output subsystem.
- The central processing unit (CPU) performs operations on data. It has three parts: an arithmetic logic unit (ALU), a control unit, and a set of registers. The arithmetic logic unit (ALU) performs logic, shift, and arithmetic operations on data. Registers are fast stand-alone storage locations that hold data temporarily. The control unit controls the operation of each part of the CPU.
- Main memory is a collection of storage locations, each with a unique identifier called the *address*. Data is transferred to and from memory in groups of bits called *words*. The total number of uniquely identifiable locations in memory is called the *address space*. Two types of memory

are available: random access memory (RAM) and read-only memory (ROM).

- The collection of devices referred to as the input/output (I/O) subsystem allows a computer to communicate with the outside world and to store programs and data even when the power is off. Input/output devices can be divided into two broad categories: nonstorage and storage devices. Nonstorage devices allow the CPU/memory to communicate with the outside world. Storage devices can store large amounts of information to be retrieved at a later time. Storage devices are categorized as either magnetic or optical.
- The interconnection of the three subsystems of a computer plays an important role, because information needs to be exchanged between these subsystems. The CPU and memory are normally connected by three groups of connections, each called a *bus*: data bus, address bus, and control bus. Input/output devices are attached to the buses through an *input/output controller* or interface. Several kinds of controllers are in use. The most common ones today are SCSI, FireWire, and USB.
- There are two methods of handling the addressing of I/O devices: isolated I/O and memory-mapped I/O. In the isolated I/O method, the instructions used to read/write to and from memory are different than the instructions used to read/write to and from input/output devices. In the memory-mapped I/O method, the CPU treats each register in the I/O controller as a word in memory.
- Today, general-purpose computers use a set of instructions called a *program* to process data. A computer executes the program to create output data from input data. Both the program and the data are stored in memory. The CPU uses repeating machine cycles to execute instructions in the program, one by one, from beginning to end. A simplified cycle can consist of three phases: *fetch*, *decode*, and *execute*.
- Three methods have been devised for synchronization between I/O devices and the CPU: programmed I/O, interrupt-driven I/O, and direct memory access (DMA).
- The architecture and organization of computers have gone through many changes during recent decades. We can divide computers architecture into two broad categories: CISC (complex instruction set computers) and RISC (reduced instruction set computers).
- Modern computers use a technique called *pipelining* to improve their throughput. The idea is to allow the control unit to perform two or three phases simultaneously, which means that processing of the next instruction can start before the previous one is finished.
- Traditionally, a computer had a single control unit, a single arithmetic logic unit, and a single memory unit. Parallel processing can improve throughput by using multiple instruction streams to handle multiple data streams.

5.11 PRACTICE SET

Review questions

1. What are the three subsystems that make up a computer?
2. What are the components of a CPU?
3. What is the function of the ALU?
4. What is the function of the control unit?
5. What is the function of main memory?
6. Define RAM, ROM, SRAM, DRAM PROM, EPROM, and EEPROM.
7. What is the purpose of cache memory?
8. Describe the physical components of a magnetic disk.
9. How are the surfaces of a magnetic disk and magnetic tape organized?
10. Compare and contrast CD-R, CD-RW, and DVD.
11. Compare and contrast SCSI, FireWire and USB controllers.
12. Compare and contrast the two methods for handling the addressing of I/O devices.

Compare and contrast the three methods for handling the synchronization of the CPU with I/O devices.

Compare and contrast CISC architecture with RISC architecture.

Describe pipelining and its purpose.

Describe parallel processing and its purpose.

Multiple-choice questions

The _____ is a computer subsystem that performs operations on data.

a. CPU
b. memory
c. I/O hardware
d. none of the above

_____ is a stand-alone storage location that holds data temporarily.

a. An ALU
b. A register
c. A control unit
d. A tape drive

_____ is a unit that can add two inputs.

a. An ALU
b. A register
c. A control unit
d. A tape drive

A register in a CPU can hold _____.

a. data
b. instructions
c. program counter values
d. all of the above

A control unit with five wires can define up to _____ operations.

a. 5
b. 10
c. 16
d. 32

A word is _____ bits.

a. 8
b. 16
c. 32
d. any of the above

23. If the memory address space is 16 MB and the word size is 8 bits, then _____ bits are needed to access each word.
    a. 8
    b. 16
    c. 24
    d. 32
24. The data in _____ is erased if the computer is powered down.
    a. RAM
    b. ROM
    c. a tape drive
    d. a CD-ROM
25. _____ is a memory type with capacitors that need to be refreshed periodically.
    a. SRAM
    b. DRAM
    c. ROM
    d. all of the above
26. _____ is a memory type with traditional flip-flop gates to hold data.
    a. SRAM
    b. DRAM
    c. ROM
    d. all of the above
27. There are _____ bytes in 16 Terabytes.
    a. 2^{16}
    b. 2^{40}
    c. 2^{44}
    d. 2^{56}
28. _____ can be programmed and erased using electronic impulses but can remain in a computer during erasure.
    a. ROM
    b. PROM
    c. EPROM
    d. EEPROM
29. _____ is a type of memory in which the user, not the manufacturer, stores programs that cannot be overwritten.
    a. ROM
    b. PROM
    c. EPROM
    d. EEPROM

30. Main memory in a computer usually consists of large amounts of _____ speed memory.
    a. high
    b. medium
    c. low
    d. any of the above
31. A _____ is a storage device to which the user can write information only once.
    a. CD-ROM
    b. CD-R
    c. CD-RW
    d. all of the above
32. A _____ is a storage device that can undergo multiple writes and erasures.
    a. CD-ROM
    b. CD-R
    c. CD-RW
    d. all of the above
33. The smallest storage area on a magnetic disk that can be accessed at one time is a _____.
    a. track
    b. sector
    c. frame
    d. head
34. If the memory has 2^{32} words, the address bus needs to have _____ wires.
    a. 8
    b. 16
    c. 32
    d. 64
35. A control bus with eight wires can define _____ operations.
    a. 8
    b. 16
    c. 256
    d. 512
36. A _____ controller is a high-speed serial interface that transfers data in packets.
    a. SCSI
    b. USB
    c. FireWire
    d. both b and c
37. The three steps in the running of a program on a computer are performed in the specific order _____.
    a. fetch, execute, and decode
    b. decode, execute, and fetch
    c. fetch, decode, and execute
    d. decode, fetch, and execute
38. In the _____ method for synchronizing the operation of the CPU with an I/O device, the I/O device informs the CPU when it is ready for data transfer.
    a. programmed I/O
    b. interrupt-driven I/O
    c. DMA
    d. isolated I/O
39. In the _____ method for synchronizing the operation of the CPU with an I/O device, the CPU is idle until the I/O operation is finished.
    a. programmed I/O
    b. interrupt-driven I/O
    c. DMA
    d. isolated I/O
40. In the _____ method for synchronizing the operation of the CPU with an I/O device, a large block of data can be passed from an I/O device to memory directly.
    a. programmed I/O
    b. interrupt-driven I/O
    c. DMA
    d. isolated I/O

Exercises

41. A computer has 64 MB (megabytes) of memory. Each word is 4 bytes. How many bits are needed to address each single word in memory?
42. How many bytes of memory are needed to store a full screen of data if the screen is made of 24 lines with 80 characters in each line? The system uses ASCII code, with each ASCII character stored as a byte.
43. An imaginary computer has sixteen data registers (R0 to R15), 1024 words in memory, and 16 different instructions (add, subtract, and so on). What is the minimum size of an instruction in bits if a typical instruction uses the following format: *Instruction M R2*.

If the computer in Exercise 43 uses the same size of word for data and instructions, what is the size of each data register?

What is the size of the instruction register in the computer in Exercise 43?

What is the size of the program counter in the computer in Exercise 43?

What is the size of the data bus in the computer in Exercise 43?

What is the size of the address bus in the computer in Exercise 43?

What is the minimum size of the control bus in the computer in Exercise 43?

A computer uses isolated I/O addressing. Its memory has 1,024 words. If each controller has 16 registers, how many controllers can be accessed by this computer?

A computer uses memory-mapped I/O addressing. The address bus uses 10 lines (10 bits). If memory is made up of 1,000 words, how many four-register controllers can be accessed by the computer?

52. Using the instruction set of the simple computer in Section 5.7, write the code for a program that performs the following calculation:

$$D \leftarrow A + B + C$$

A, B, C, and D are integers in two's complement format. The user types the value of A, B, and, C, and the value of D is displayed on the monitor.

53. Using the instruction set of the simple computer in Section 5.7, write the code for a program that performs the following calculation:

$$B \leftarrow A + 3$$

A and 3 are integers in two's complement format. The user types the value of A and the value of B is displayed on the monitor. (Hint: use the increment instruction.)

54. Using the instruction set of the simple computer in Section 5.7, write the code for a program that performs the following calculation:

$$B \leftarrow A - 2$$

A and 2 are integers in two's complement format. The user types the value of A and the value of B is displayed on the monitor. (Hint: use the decrement instruction.)

55. Using the instruction set of the simple computer in Section 5.7, write the code for a program that adds n integers typed on the keyboard and displays their sum. You need first to type the value of n. (Hint: use decrement and jump instructions and repeat the addition n times.)

56. Using the instruction set of the simple computer in Section 5.7, write the code for a program that accepts two integers from the keyboard. If the first integer is 0, the program increment the second integer, and if the first integer is 1, the programs decrement the second integer. The first integer must be only 0 or 1 otherwise the program fails. The program displays the result of the increment or decrement.

6.1 INTRODUCTION

A **network** is a combination of hardware and software that sends data from one location to another. The hardware consists of the physical equipment that carries signals from one point in the network to another. The software consists of instructions that make the services that we expect from a network possible.

Network criteria

A network must be able to meet a number of criteria. The most important of these are performance, reliability, and security.

Performance

Performance can be measured in many ways, including transit time and response time. Transit time is the amount of time required for a message to travel from one device to another. Response time is the elapsed time between an enquiry and a response. The performance of a network depends on a number of factors, including the number of users, the type of transmission medium, the capabilities of the connected hardware, and the efficiency of the software.

Reliability

In addition to accuracy of delivery, network **reliability** is measured by the frequency of failure, the time it takes to recover from a failure, and the network's robustness in a catastrophe.

Security

Network **security** issues include protecting data from unauthorized access, damage and change, and implementing policies and procedures for recovery from breaches and data losses. We discuss security in Chapter 16.

Physical structures

Before discussing networks, we need to define some network attributes.

Type of connection

A network consists of two or more devices connected through links. A link is a communications pathway that transfers data from one device to another. For visualization purposes, it is simplest to imagine any link as a line drawn between two points. For communication to occur, two devices must be connected in some way to the same link at the same time. There are two possible types of connections: point-to-point and multipoint.

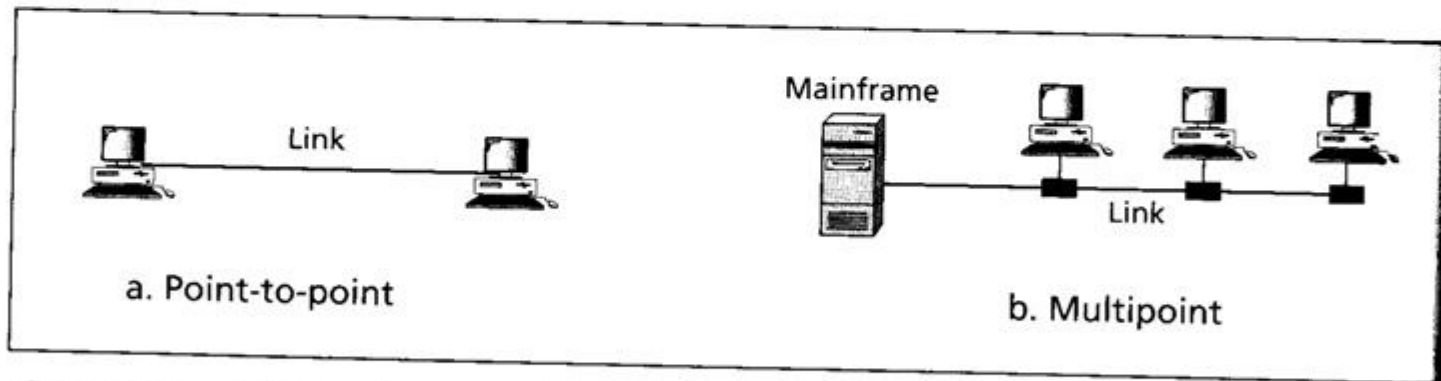


Figure 6.1 Types of connections: point-to-point and multipoint

A **point-to-point connection** provides a dedicated link between two devices, and the entire capacity of the link is reserved for transmission between these two devices. A **multipoint connection** (also called **multidrop connection**) is one in which more than two specific devices share a single link. In a multipoint environment, the capacity of the channel is shared, either spatially or temporally.

ical
logy

The term *physical topology* refers to the way in which a network is laid out physically. Two or more devices connect to a link: one or more links form a topology. The topology of a network is the geometric representation of the relationship of all the links and devices (usually called **nodes**) to one another. There are four possible basic topologies: mesh, star, bus, and ring (see Figure 6.2).

In a **mesh topology**, every device has a dedicated point-to-point link to every other device. In a **star topology**, each device has a dedicated point-to-point link only to a central controller, usually called a **hub**. A **bus topology** uses a multipoint link. One long cable, called the *bus*, acts as a **backbone** to link all the devices in a network. Nodes are connected to the bus cable by drop lines and taps (connectors). In a **ring topology**, each device has a dedicated point-to-point connection with only the two devices on either side of it. A signal is passed along the ring in one direction, from device to device, until it reaches its destination. Each device in the ring incorporates a repeater. When a device receives a signal intended for another device, its repeater regenerates the bits and passes them along.

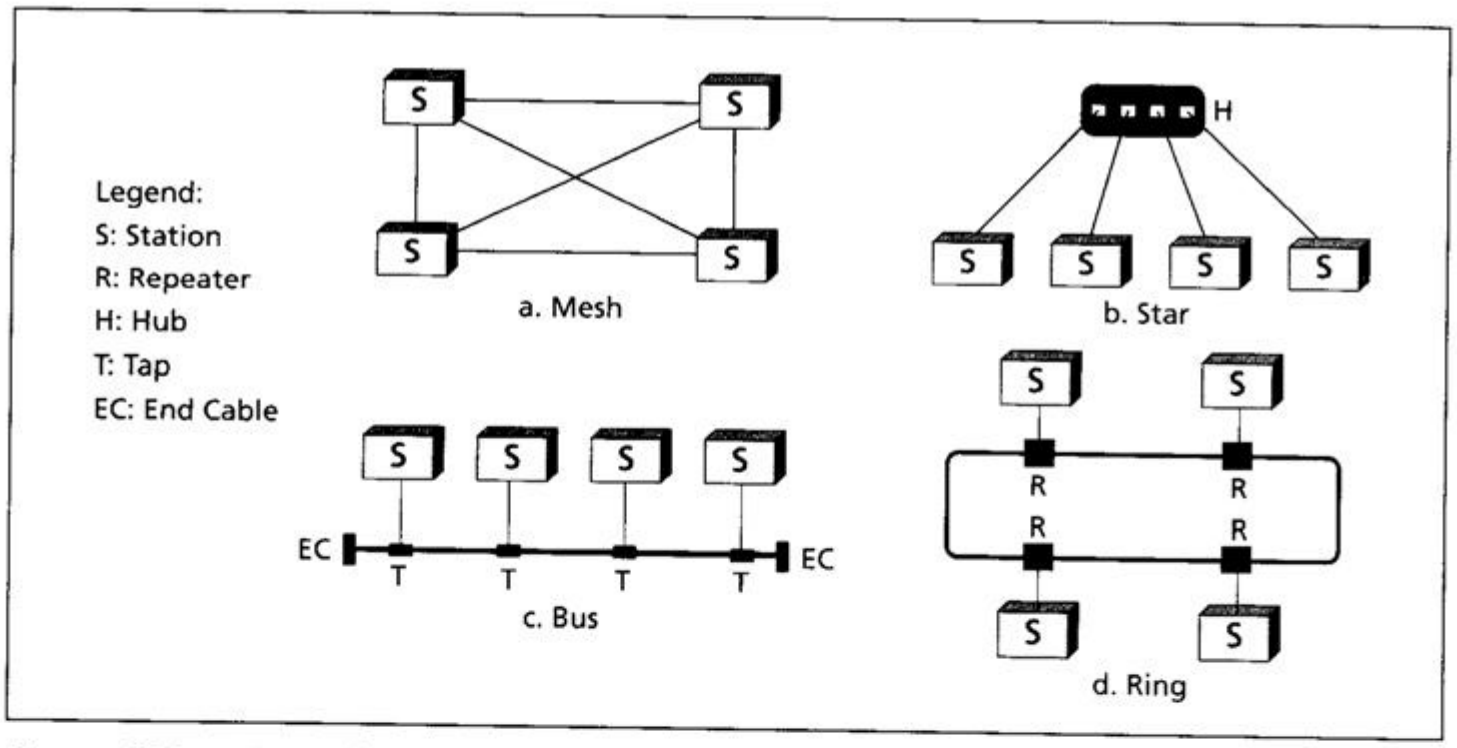


Figure 6.2 Four physical topologies

Each topology has its own advantages and disadvantages. A mesh guarantees that each connection can carry its own data load, eliminating traffic problems. It is also robust in the sense that if one link becomes unusable, it does not incapacitate the entire network. The main disadvantages of a mesh topology are related to the amount of cabling and the number of input/output ports required.

A ring is relatively easy to install and reconfigure. In addition, fault isolation is simplified. However, a break in the ring (such as a disabled station) can disable the

entire network. This weakness can be solved by using a dual ring or a switch capable of closing off the break. Ring topology was prevalent when IBM introduced its local-area network Token Ring. Today, the need for higher-speed networks has made this topology less popular.

A bus topology can be installed easily. Backbone cable can be laid along the most efficient path, then connected to the nodes by drop lines of various lengths. However, a fault or break in the bus cable stops all transmission, even between devices on the same side of the problem section.

Today the most common topology in high-speed local-area networks is a star. A star topology is less expensive than a mesh topology, but has most of the advantages of the latter. One big disadvantage of a star topology is the dependency of the whole topology on one single point, the hub. If the hub goes down, the whole system is dead. However, the low cost of a star topology and ease of installation and scalability has made the star topology the only common topology.

Categories of networks

Today networks can be divided into three broad categories: local-area networks (LANs), wide-area networks (WANs), and metropolitan area networks (MANs). The category into which a network falls is determined by its size. A LAN normally covers a small area, a WAN can be worldwide, while a MAN covers a small city or town.

Local area networks

A **local area network (LAN)** is usually privately owned and links the devices in a single office, building, or campus (see Figure 6.3). Depending on the needs of an organization and the type of technology used, a LAN can be as simple as two PCs and a printer in someone's home office, or it can extend throughout a company and include audio and video peripherals. Currently, LAN size is limited to a few kilometers.

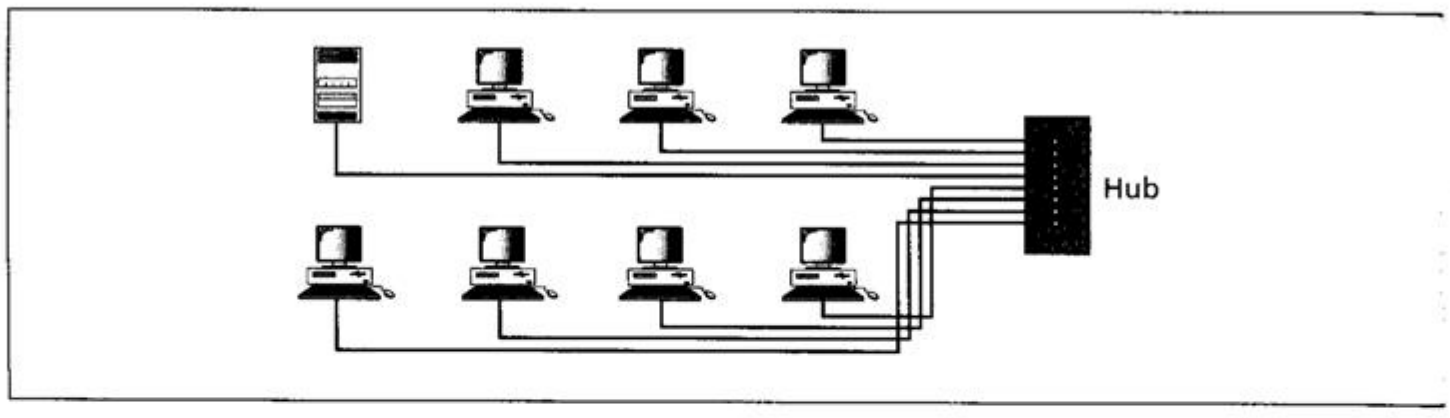


Figure 6.3 An isolated LAN connecting eight computers to a hub

LANs are designed to allow resources to be shared between personal computers or workstations. The resources to be shared can include hardware, such as a printer, software, such as an application program, or data. A common example of a LAN found in many business environments links a workgroup of task-related computers, for example, engineering workstations or accounting PCs. One of the computers may be given a large-capacity disk drive and may become a server to clients. Software can

be stored on this central server and used by the whole group. In this example, the size of the LAN may be determined by licensing restrictions on the number of users per copy of software, or by restrictions on the number of users licensed to access the operating system.

area
orks

A **wide area network (WAN)** provides long-distance transmission of data, images, audio, and video information over large geographic areas that may comprise a country, a continent, or even the whole world. A WAN can be as complex as the backbones that connect the Internet or as simple as a dial-up line that connects a home computer to the Internet. A point-to-point WAN is normally a single line connection between two devices such as a dial-up line or a cable line. A backbone WAN is a complex network operated by a service provider and normally connects Internet service providers (ISPs) (see Figure 6.4).

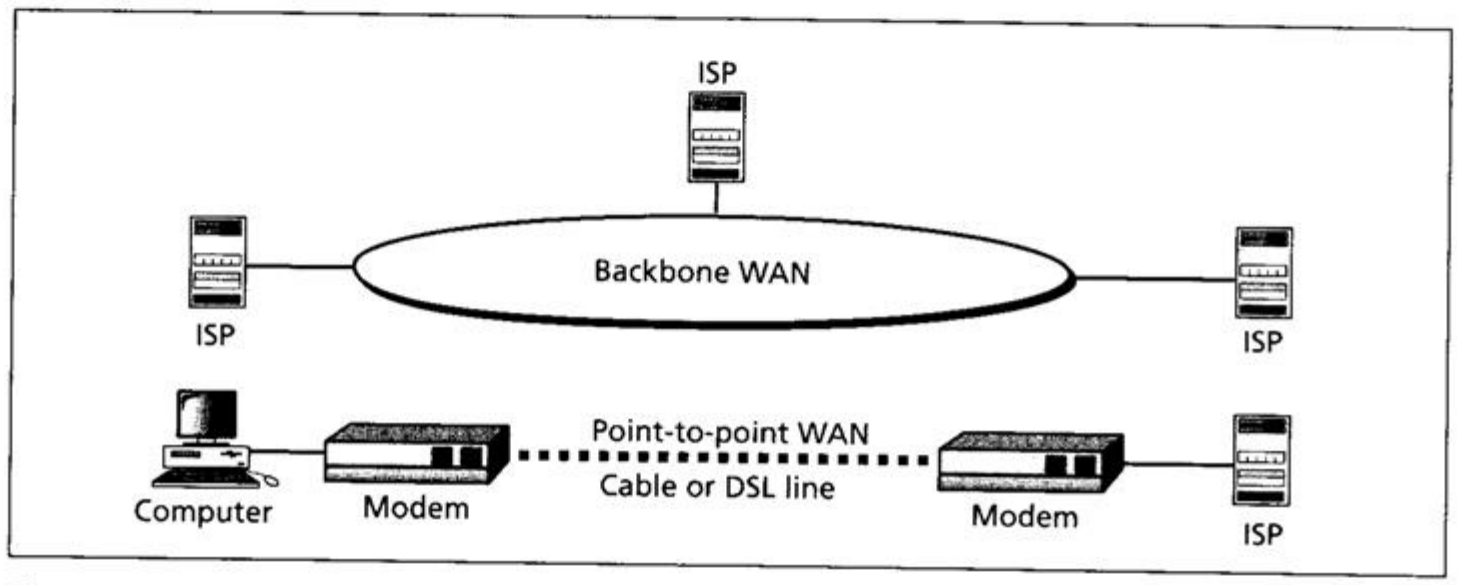


Figure 6.4 A point-to-point WAN and a backbone WAN

etropolitan
a
etworks

A **metropolitan area network (MAN)** is a network with a size between a LAN and a WAN. It normally covers the area inside a town or a city. It is designed for customers who need high-speed connectivity, normally to the Internet, and has endpoints spread over a city or part of city. A good example of a MAN is that part of a telephone company's network that can provide a high-speed DSL line to the customer. Another example is the cable network that was originally designed for cable television, but today can also be used for high-speed data connection to the Internet.

An internet

Today, it is very rare to see a network in isolation: networks are connected to one another. When two or more networks are connected, they become an **internetwork**, or an **internet** (lowercase "i"). Figure 6.5 shows an example of an internet.

In other words, a network is a group of connected communicating devices such as computers and printers, while an internet is two or more networks that can communicate with each other. In Figure 6.5 we have seven LANs and two WANs. Each LAN is a local network connecting computers, printers, and other devices inside a

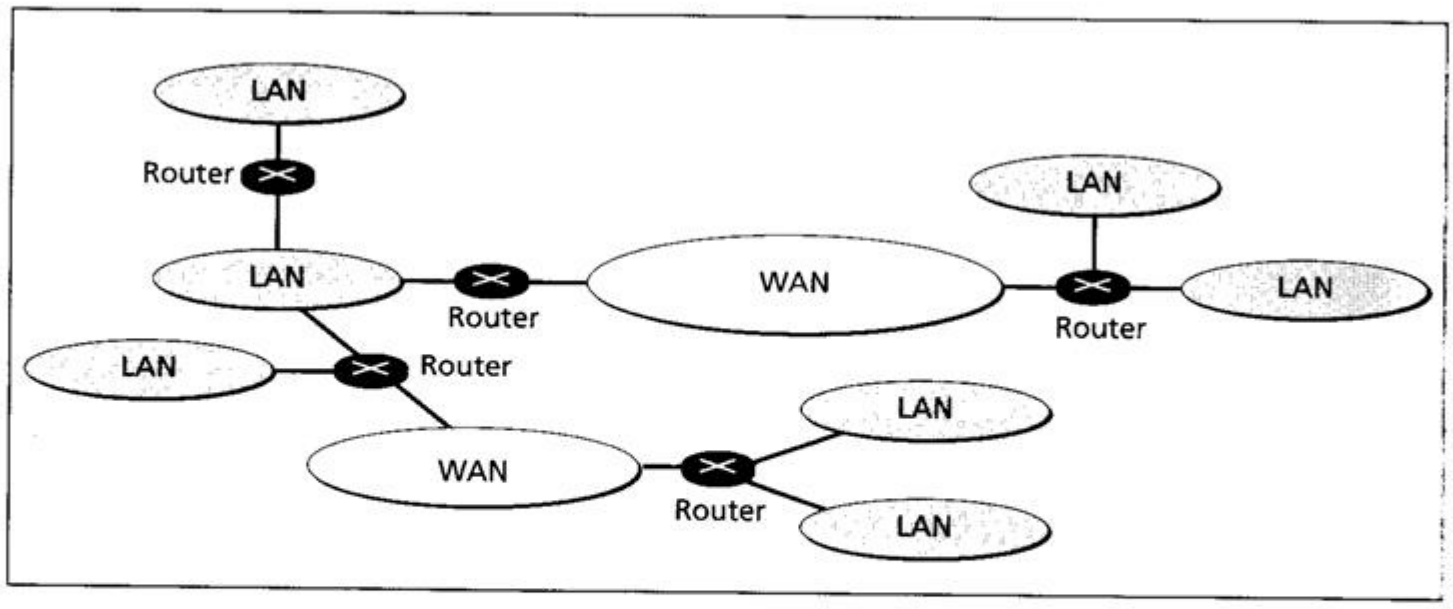


Figure 6.5 An internet made of WANs, LANs, and routers

building or a campus. The WANs are either point-to-point (a leased line, for example) or a backbone operated by a network service provider. The **routers** are connecting devices that route the *packets* (messages) travelling through the internet.

The Internet

The most notable internet is the Internet (uppercase "I"), a collaboration of hundreds of thousands of interconnected networks. Private individuals, as well as various organizations such as government agencies, schools, research facilities, corporations, and libraries in more than 100 countries use the Internet. Millions of people are users.

It is difficult to give an accurate representation of the Internet, because it is continually changing—new networks are being added and networks of defunct organizations removed. Today most end users who want an Internet connection use the services of **Internet service providers (ISPs)**. An ISP is an organization with one or more *servers* (high capacity computers) that are connected to the Internet through high-speed links. Individual Internet users or small companies can be connected to the servers of a local ISP by establishing a service contract and paying a fee. There are international service providers, national service providers, regional service providers, and local service providers. The Internet today is run by private companies, not the government. Figure 6.6 shows a conceptual (but not geographic) view of part of the Internet.

The Internet has revolutionized many aspects of our daily lives. It has affected the way we do business as well as the way we spend our leisure time. Count the ways you've used the Internet recently. Perhaps you've sent electronic mail (e-mail) to a business associate, paid a utility bill, read a newspaper from a distant city, or looked up a local movie schedule—all by using the Internet. Or maybe you researched a medical topic, booked a hotel reservation, chatted with a colleague, or comparison-shopped for a car. The Internet is a communication system that has brought a wealth of information to our fingertips and organized it for our use.

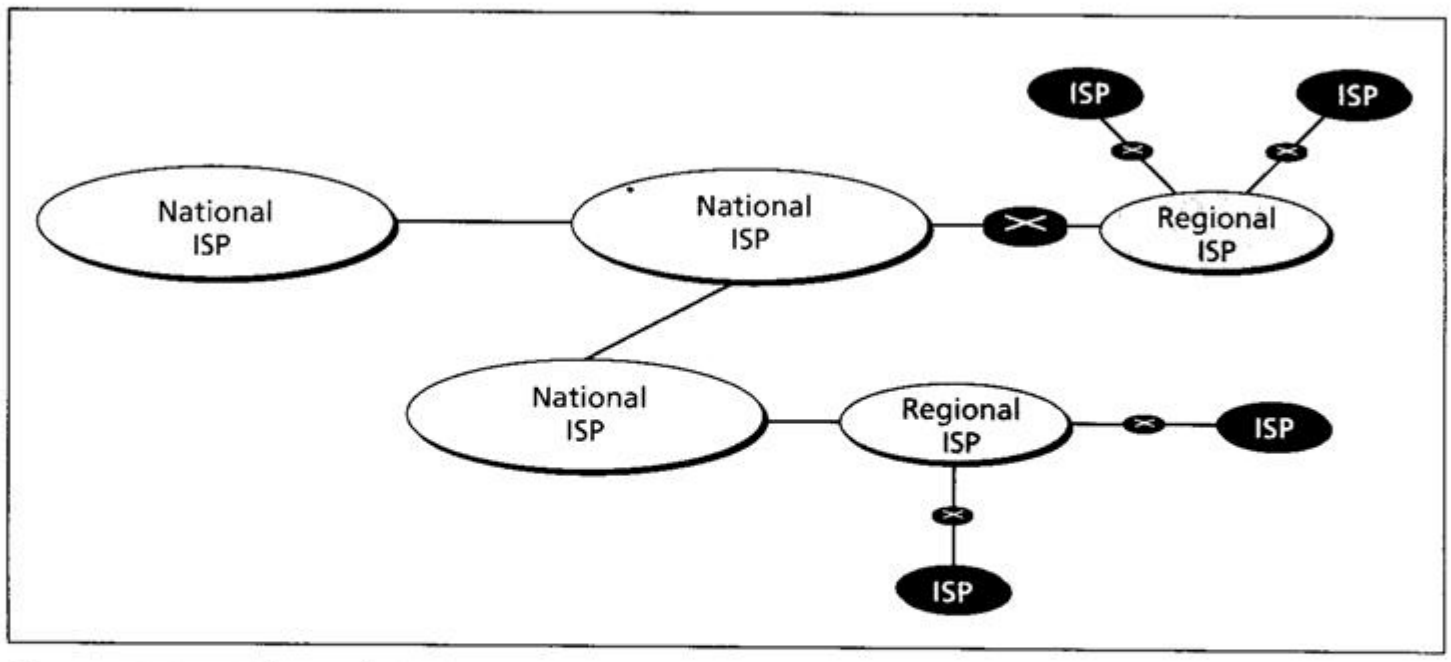


Figure 6.6 Hierarchical organization of the Internet

6.2 TCP/IP PROTOCOL SUITE

We can compare the task of networking to the task of solving a math problem with a computer. The fundamental job of solving the problem with a computer is done by computer hardware. However, this is a very tedious task if only hardware is involved. We would need switches for every memory location to store and manipulate data. The task is much easier if software is available. At the highest level, a program can direct the problem-solving process: the details of how this is done by the actual hardware can be left to the layers of software that are called by the higher levels.

Compare this to a service provided by a computer network. For example, the task of sending an e-mail from one place in the world to another can be broken down into several tasks, each performed by a layer. Each layer uses the services of a lower layer. At the lowest layer, a signal, or a set of signals, is sent from the source computer to the destination computer.

To divide the services required to perform a task, the Internet has created a set of rules called *protocols*. These allow different local and wide area networks, using different technologies, to be connected together and carry a message from one point to another. The set, or *suite*, of protocols that controls the Internet today is referred to as the **TCP/IP protocol suite**. The abbreviations (TCP and IP) will become clear as we explain different protocols.

The original TCP/IP protocol suite was defined as having four layers: host-to-network (or link), internet (network), transport, and application. However, the TCP/IP protocol suite today is normally considered as a five-layer model, as shown in Figure 6.7.

Figure 6.8 shows the layers involved when a message is sent from device A to device B. As the message travels from A to B, it may pass through many routers. Routers use only the first three layers.

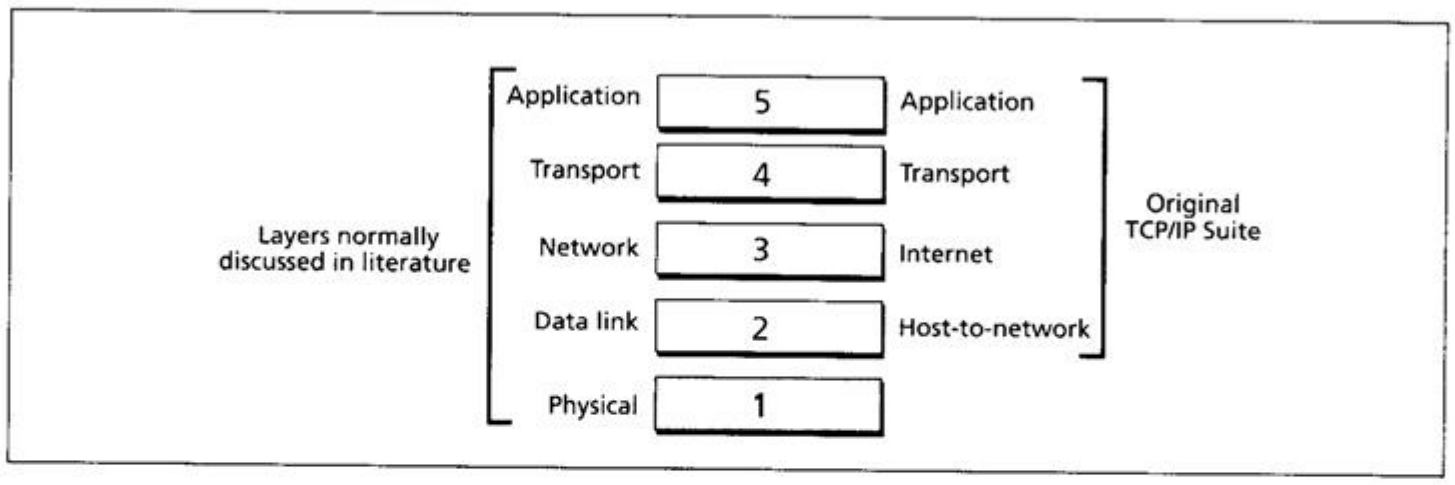


Figure 6.7 The TCP/IP protocol suite

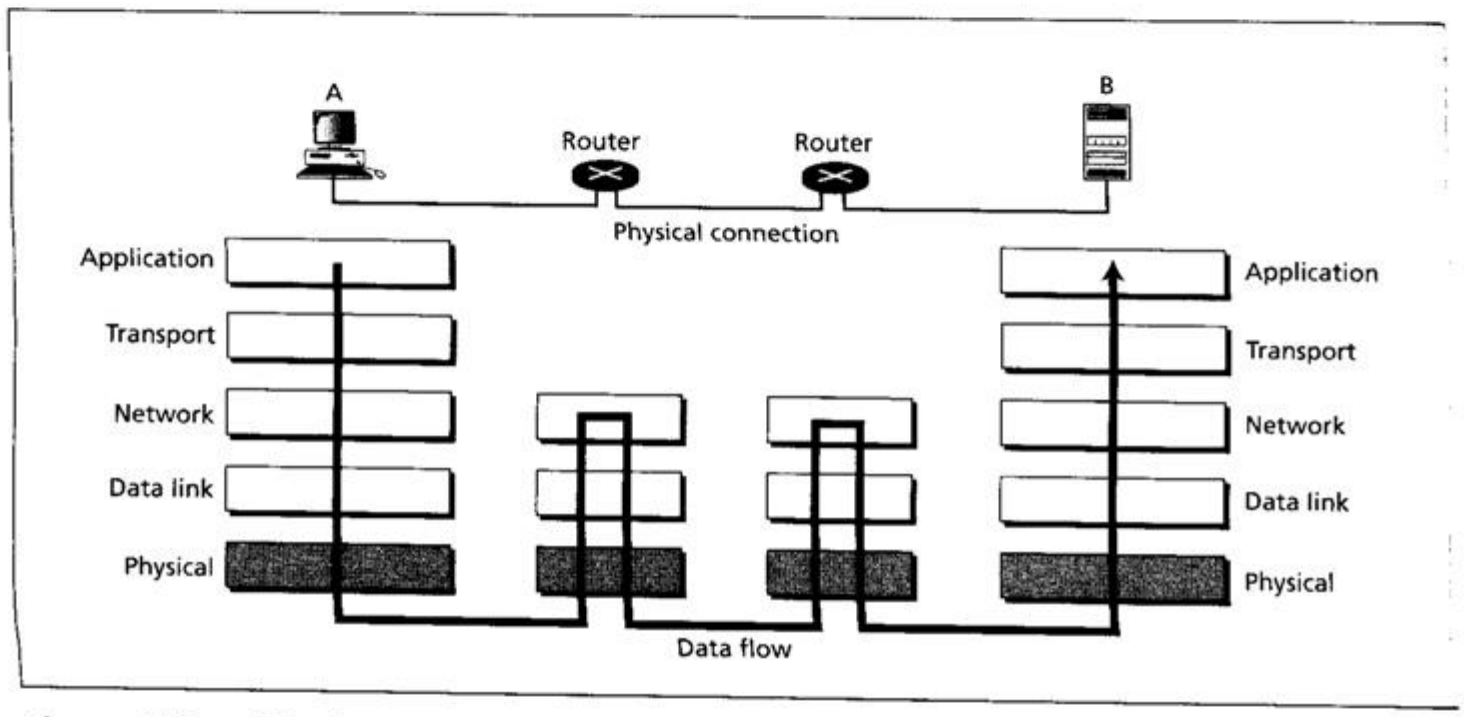


Figure 6.8 The interaction between layers in the TCP/IP protocol suite

Within a single machine, each layer calls on the services of the layer immediately below it. Layer 3, for example, uses the services provided by layer 2 and provides services for layer 4.

6.3 LAYERS

This section briefly describes the function of each layer in the TCP/IP protocol suite. We show how a message travels through the different layers until it reaches the physical layer and is sent by the transmission media.

Application layer

The **application layer** enables a user, whether human or software, to access the network. It provides support for services such as electronic mail, remote file access and transfer, browsing the World Wide Web, and so on. Later in this chapter we

discuss some common applications of the Internet. The application layer is the only layer seen by most users of the Internet.

The application layer is responsible for providing services to the user.

t-server
ecture

Although there are two architectures (designs) that allow two application programs, running on two remote computers, to communicate with each other, **client-server architecture** is more common. The other architecture, **peer-to-peer architecture**, is becoming popular, but we only discuss client-server architecture in this chapter. In client-server architectures, each application is made up of two separate but related programs: a *client* program and a *server* program. The server program must be running all the time: the client program can be run only when needed. This implies that the computer on which the server program is running must be on all the time, while the computer that runs the client program may also only be on when needed. In practice, the computer that runs the server program is referred to as the *server* and the computer that runs the client program is referred to as the *client* (Figure 6.9).

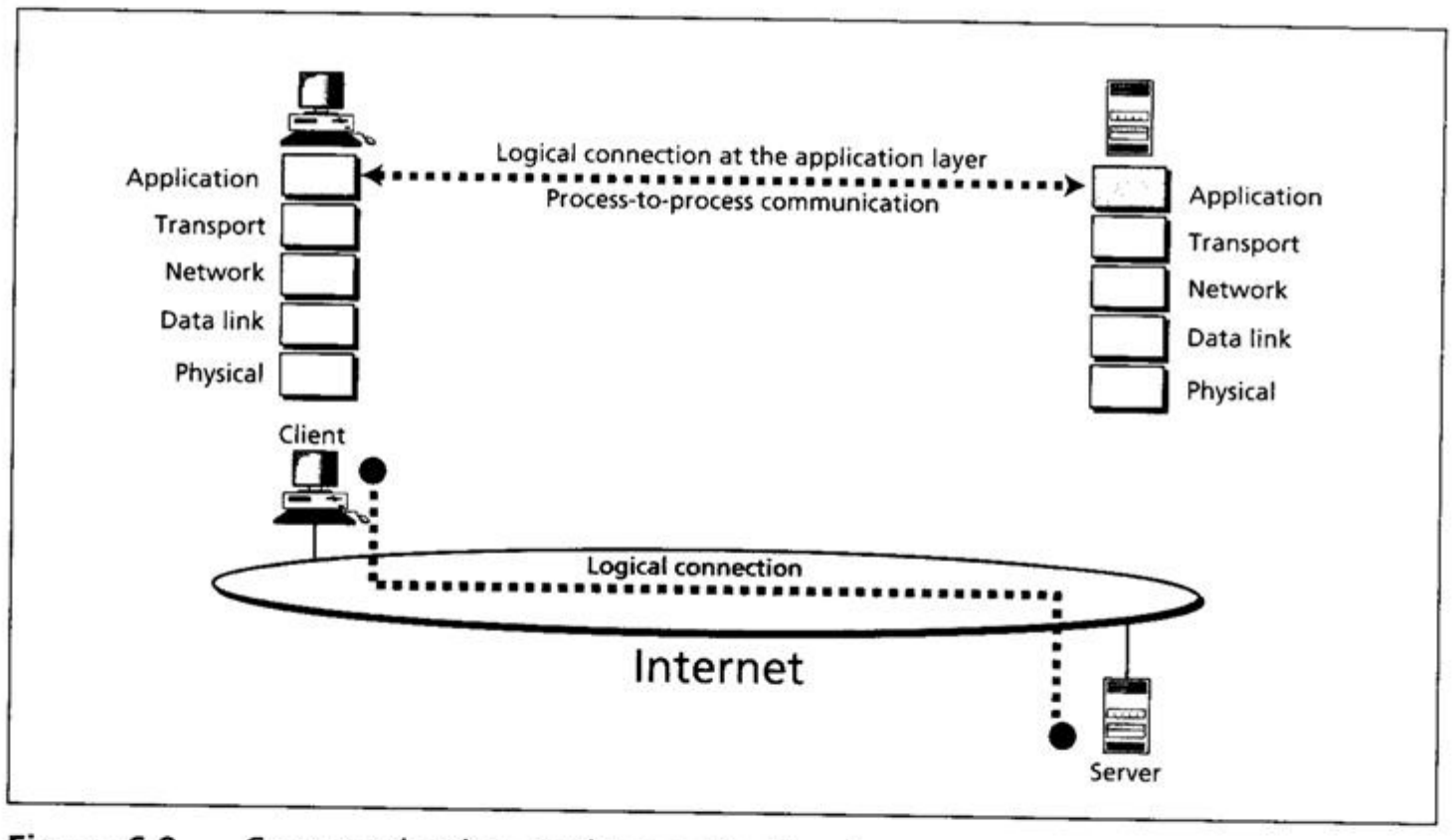


Figure 6.9 Communication at the application layer

The communication between a client program and a server program is referred to as **process-to-process communication**, because a running program in this architecture is called a **process**. In Figure 6.9 the server process is the server program that is running all the time waiting to receive a request from a client process.

The user (human or software) that needs the services of the server process runs the dormant client program (changing it to a client process) and lets the client process request the service, which will be responded to by the server process. Note that when a server process is running many client processes can request services and obtain responses.

Application-layer address

When a client needs to send a request to a server, it needs the server application-layer address. Although the number of Internet applications is limited, the number of sites that run a server for a specific application is huge. For example, there are many sites that run an HTTP (defined later) server, and HTTP clients can access these sites to browse or download information stored at the site. To identify one particular HTTP site, the client uses a **Uniform Resource Locator (URL)**. For example, to access the publisher of this book, the user needs to use the URL *http://www.cengage.co.uk*. As we will see later, the server application-layer address is not used for delivery of messages, it only helps the client to find the actual address of the server computer. Note that a client site needs no identification at the application layer because it is not a service provider—it only receives services. Different sites will have different application-layer addresses, although they may be running the same server type (for example, HTTP).

The address at the application layer cannot be used to send a message. The client needs the actual address of the server in the network. The situation is similar to sending a letter to a person if we only know their name. The post office cannot deliver a letter based only on the name of the recipient: the actual address is needed. The application layer address can help the client to find the actual address of the server in the network. Each computer in the network has an address, called a *logical address* or *IP address*, as we see shortly.

The server application-layer address can help the client to find the **IP address** of the server computer. The client process should already know the address of a **domain name server (DNS)**. These servers, which are spread over the Internet, each have a directory that matches domain names to IP addresses. The client prepares and sends a message to a DNS server and asks for the actual IP address of the desired server. After receiving the response, the client server knows the IP address of the desired server. Figure 6.10 shows the idea. In the figure, d_5 is the application-layer address and d_3 is the network-layer or IP address.

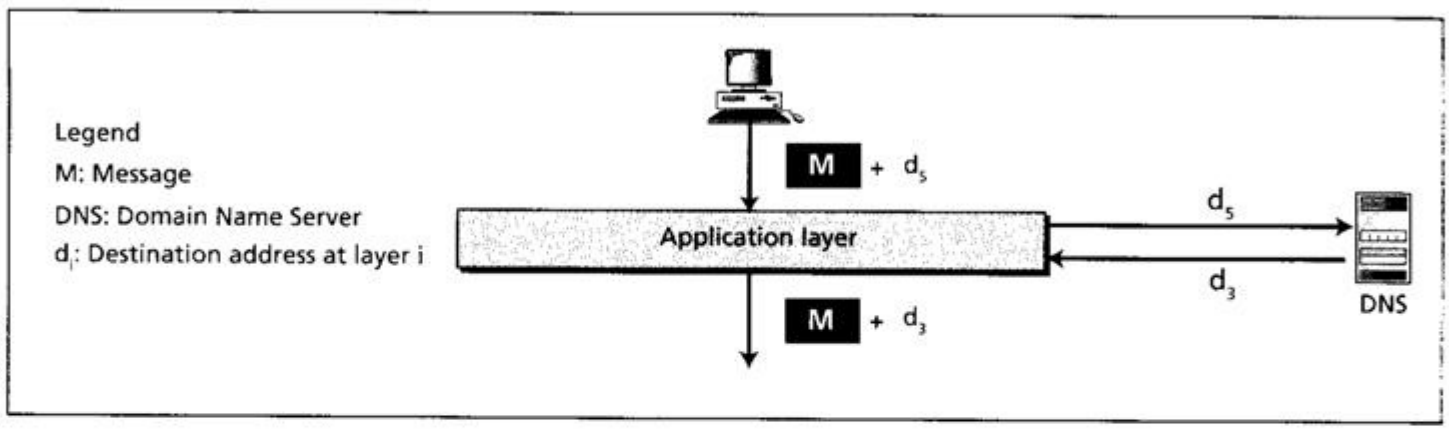


Figure 6.10 Addresses at the application layer

Transport layer

The **transport layer** is responsible for **process-to-process delivery** of the entire message: logical communication is created between the transport layer of the client and the server computer. In other words, although physical communication is between two physical layers (through many possible links and routers), the two application layers consider the transport layer as the agent that takes responsibility

for delivering the messages. Figure 6.11 shows the logical connection between the two transport layers through the Internet.

The transport layer is responsible for the logical delivery of a message between client and server processes.

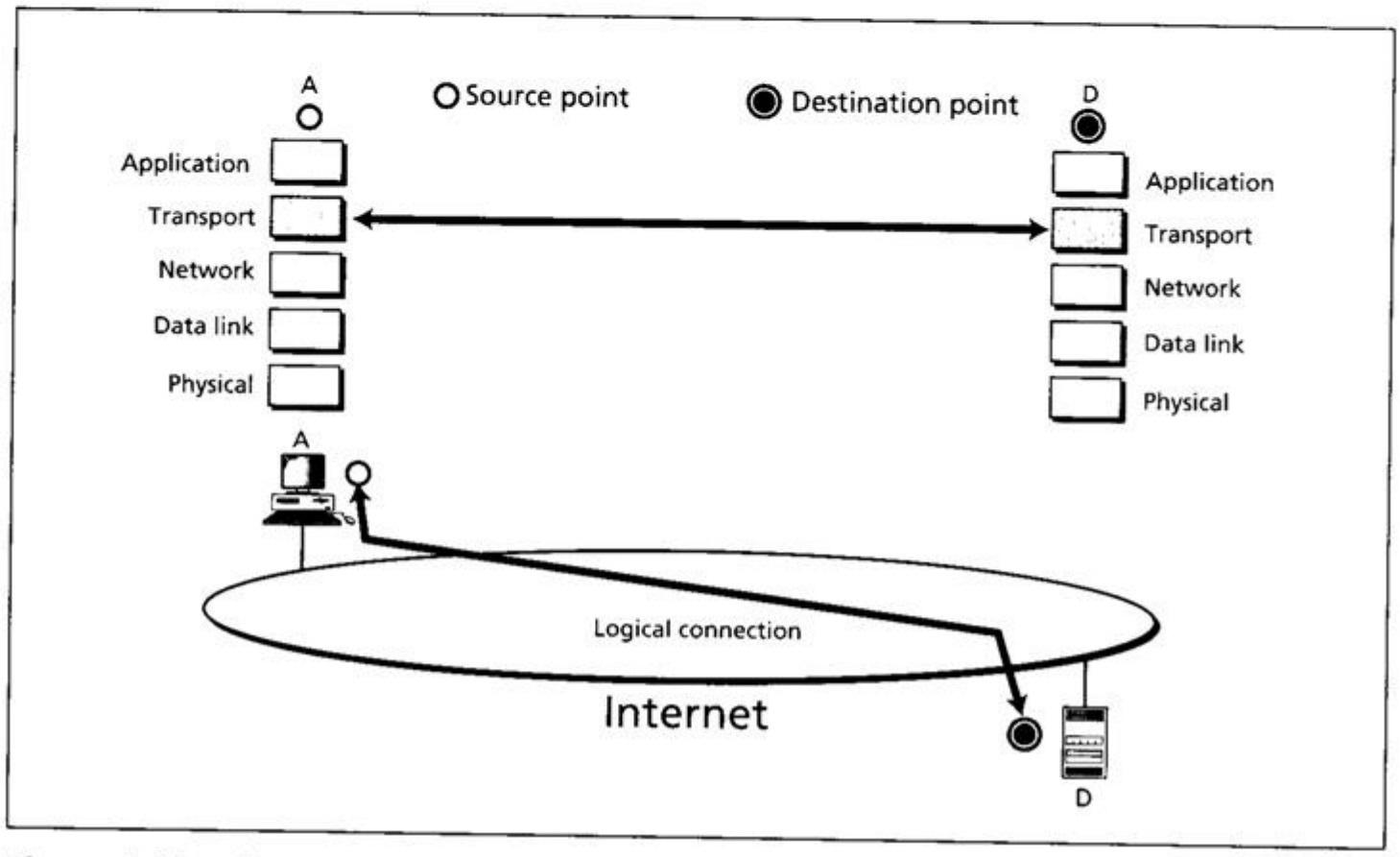


Figure 6.11 Communication at the transport layer

**sport-
r
resses
t
bers)**

The IP address of the server is necessary for communication, but more is required. The server computer may be running several processes at the same time, for example an FTP server process and an HTTP server process. When the message arrives at the server, it must be directed to the correct process. We need another address for server process identification, called a **port number**. An analogy may help here. Assume that everyone in a community lives in apartment buildings. To send a message to a specific person, we need to know both the building address and the apartment number. The IP address is similar to the building address, and the port address is similar to apartment number. Server port numbers are well-known: most computers have a file that gives server port addresses, while the client port number can be temporarily assigned by the computer running the client process. However, the Internet restricts the range of temporary port numbers to avoid infringing the range of well-known port addresses (Figure 6.12). In the figure, d_4 is the well-known port number and s_4 is the temporary port number.

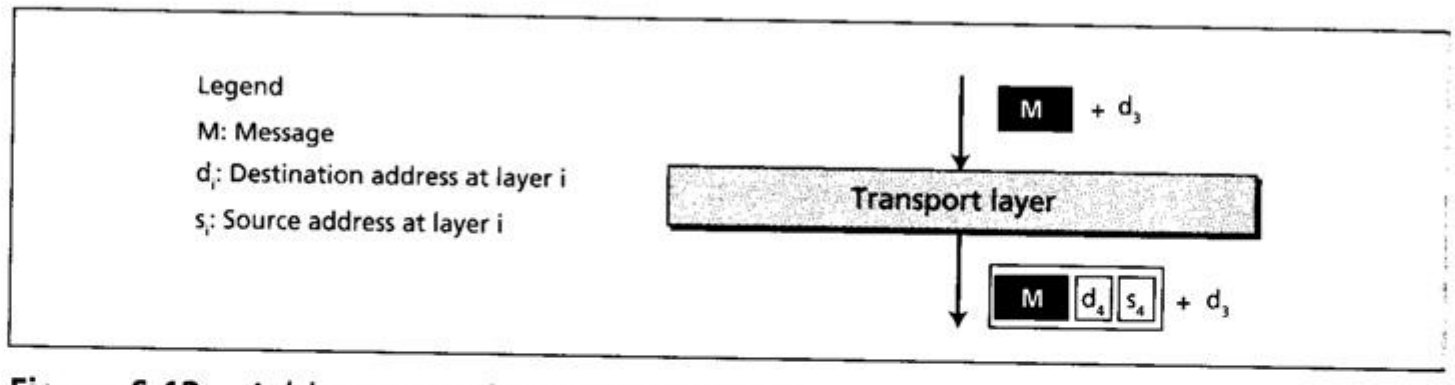


Figure 6.12 Addresses at the transport layer

Multiplexing and demultiplexing

One of the duties of the transport layer is **multiplexing** and **demultiplexing**. To continue with our analogy, in large cities where people live in big apartment buildings, the mailman cannot collect the mail from each individual tenant and cannot deliver mail to each individual tenant. This duty is normally taken by a concierge. The concierge collects outgoing mail from tenants and delivers them to the mailman (multiplexing), and distributes incoming mail to the tenants after it has arrived (demultiplexing). The transport layer does the same for the different processes, collecting outgoing messages from processes and distributing incoming messages to processes. The transport layer uses port numbers (similar to the apartment numbers used by the concierge or apartment manager) to do the multiplexing and demultiplexing.

This implies that port numbers should be unique in a computer. Server processes use well-known port numbers, but client processes use temporary port numbers that are assigned by the transport layer.

Congestion control

The transport layer carries out **congestion control**. The underlying networks that physically carry the packets may become congested with traffic. This may cause the network to drop—lose—some packets. Some protocols use a buffer for each process. The messages are stored in the buffer before being sent. If the transport layer detects that there is congestion in the network, it slows transmission. This is similar to the effect of traffic lights installed at highway junctions in some countries. Only one car is allowed to pass the lights with each green signal: if there is traffic in the junction, the interval between green lights increases.

Flow control

The transport layer also carries out flow control. The transport layer at the sender site can monitor the transport layer at the receiver site to check that the receiver is not overwhelmed with received packets. This is achieved if the system uses acknowledgment from the receiver. The receiver can acknowledge receipt of each, or a group of, packets to allow the sender to check that the receiver is not overwhelmed.

Error control

During a message's journey it can be corrupted, lost, duplicated, or received out of order. The sending transport layer is responsible for ensuring that the message is received correctly by the destination transport layer. The acknowledgment system described above can also provide error control. The transport layer can keep a copy of the message in a buffer (temporary storage) until it receives acknowledgement from the receiver that the packet has arrived uncorrupted and in the right order. If

no acknowledgment arrives in due time, or a negative acknowledgment arrives (denoting the packet is corrupted), the sender resends the packet. To be able to check the ordering of packets, the transport layer can add a sequence number to the packet and an acknowledgement number to the acknowledgment.

During the life of the TCP/IP protocol suite three transport layer protocols have been designed: UDP, TCP, and SCTP.

UDP

The **User Datagram Protocol (UDP)** is the simplest of all three protocols. UDP does multiplexing and demultiplexing (by adding source and destination ports to the packet). It also does a type of error control by adding a checksum (see Appendix H) to the packet. Error control in this case is only a "yes or no" procedure—the receiver recalculates the checksum to see if an error has occurred during transmission. If the receiver concludes that the packet is corrupted, it silently drops the packet without warning the sender to resend it. UDP is not a perfect protocol, but it is useful in some circumstances when other duties are not required of the transport layer, or are already carried out by the application layer. UDP, however, being simple, has the advantage of being fast. UDP is also more efficient: it carries less information overhead than other protocols. Some application programs prefer to use UDP because they either do flow and error control themselves, or need a fast and efficient response. An example of this is the DNS server discussed before. UDP is also suitable for applications whose timing is more important than accuracy. For example, when we are dealing with real-time transmission of video on the Internet, it is important that all packets that form an image arrive on time, and if a small number of packets are lost or corrupted, the viewer cannot detect a small and transient error in the image.

UDP is referred to as a **connectionless protocol** because it does not provide a logical connection between packets belonging to a single message. Each packet is an individual entity in UDP, due to the lack of sequence numbers. This service is similar to that provided by the regular mail system. Assume you need to send a set of ordered packets to a destination: the post office cannot guarantee that they are delivered in the requested order. Each packet, as far as the post office is concerned, is an individual entity with no relationship to other packets.

TCP

Transmission Control Protocol (TCP) is a protocol that supports all the duties of a transport layer. However, it is not as fast and as efficient as UDP. TCP uses sequence numbers, acknowledgment numbers, and checksums. It also uses buffers at the sender's site. This combination of provisions provides multiplexing, demultiplexing, flow control, congestion control, and error control.

TCP is referred to as a **connection-oriented protocol** because it provides a logical connection between the two transport layers: one at the source, the other at the destination. The use of sequence numbers maintains a connection: if a packet arrives out of order or lost, it will be resent. The transport layer at the receiver does not deliver the out of sequence packet to the application process, but holds all packets in a message until they are received in the correct order.

Although TCP is a perfect transport-layer protocol for data communication, it is not suitable for real-time transmission of audio or video. If a packet is lost, TCP needs to resend it, which destroys the synchronization of packets.

SCTP

Stream Control Transmission Protocol (SCTP) is a new protocol that is designed for new services expected from the Internet, such as Internet telephony and video streaming. This protocol combines the advantages of both UDP and TCP. Like UDP, it is suitable for real-time transmission of audio and video, but like TCP, it provides error and flow control.

The network layer

The **network layer** is responsible for the source-to-destination (computer-to-computer or host-to-host) delivery of a packet, possibly across multiple networks (links). The network layer ensures that each packet gets from its point of origin to its final destination.

The network layer is responsible for the delivery of individual packets from the source host to the destination host.

The services expected from a network layer are similar to the services mentioned above for a "perfect" transport layer, with the exception of multiplexing/demultiplexing. In other words, a network layer protocol should provide congestion, flow and error control. However, this does not happen in the Internet, as we will see shortly.

Network-layer addresses

The packet travelling from the client to the server and the packet returning from the server need a network-layer address. The server address is provided by the server, as discussed above, while the client address is known by the client computer. Figure 6.13 shows the idea. In addition, the network layer uses its routing table to find the logical address of the next hop (router), which is passed to the data link layer. This logical address (n_3) is needed by the data link layer to find the data link layer address of the next router.

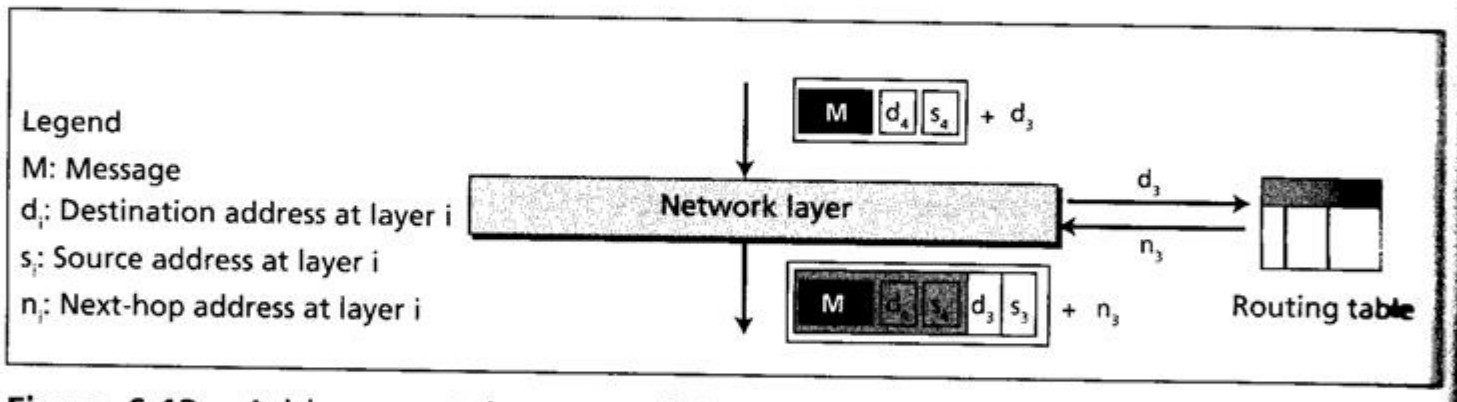


Figure 6.13 Addresses at the network layer

Routing

The network layer has a specific duty: **routing**. Routing means determination of the partial or total path of a packet. As the Internet is a collection of networks (LANs,

WANs, and MANs), the delivery of a packet from its source to its destination may be a combination of several deliveries: a source-to-router delivery, several router-to-router deliveries, and finally a router-to-destination delivery.

We can again compare the delivery of a packet on the Internet to the delivery of a letter using the regular postal service. Imagine that a letter is to go from a small town in California to a small town in Florida. The sender drops the letter at the post office of the town in California (the first router). The local post office collects the letter and sends it to the main post office in Los Angeles (a router). The letter is probably delivered to Los Angeles airport (another router). The postal airplane delivers the letter to Miami airport (another router). The letter then is delivered to the main post office in Miami (another router). The letter is then sent to the local post office of the destination town (the final router). Finally it will be delivered to the recipient of the letter.

This is probably the normal scenario. But what happens if, due to bad weather, the postal airplane cannot fly from Los Angeles to Miami? The office at Los Angeles airport (acting as a router) may decide to choose another route. The letter might be sent to New York. After arrival, it could be carried to the Miami main post office by road. Note that in this case the source and destination addresses of the letter remain the same, but the route is changed. The post office router at Los Angeles airport decides that the best route at this moment, although probably longer, is via New York.

Note that in both scenarios, the source and destination address of the letter (as written on it) remain the same: the only thing that is changed is the route. The route is selected based on the destination address and the available best paths. For example, in the second scenario, when the letter arrives at Los Angeles airport, the post office sends it by air to New York, not Miami.

The situation is the same with routing in the network layer. When a router receives a packet, it consults its routing table to determine the best route for the packet to reach its final destination. The routing table provides the IP address of the next router. When the packet reaches the next router, a new decision is made. In other words, the routing decision is made at each router.

Figure 6.14 shows the route of a packet from source to destination through some networks. The source is computer A, the destination is computer D. When the packet arrives at router R1, this router selects router R4 (instead of R3) as the next router (probably WAN 1 is either down or congested). Router R4 selects router R5 as the next router. Finally the packet is delivered to its destination, computer D. Note that the routers use only the first three layers of the TCP/IP protocol suite: they do not need the transport layer, which is responsible for the end-to-end delivery.

How does a router update its routing table? This is done by other protocols, *routing protocols*. A routing protocol sends its own messages to all routers on the Internet to update them about routes. Several routing protocols are used on the Internet, such as RIP, OSPF, and BGP, but a discussion of these is beyond the scope of this chapter.

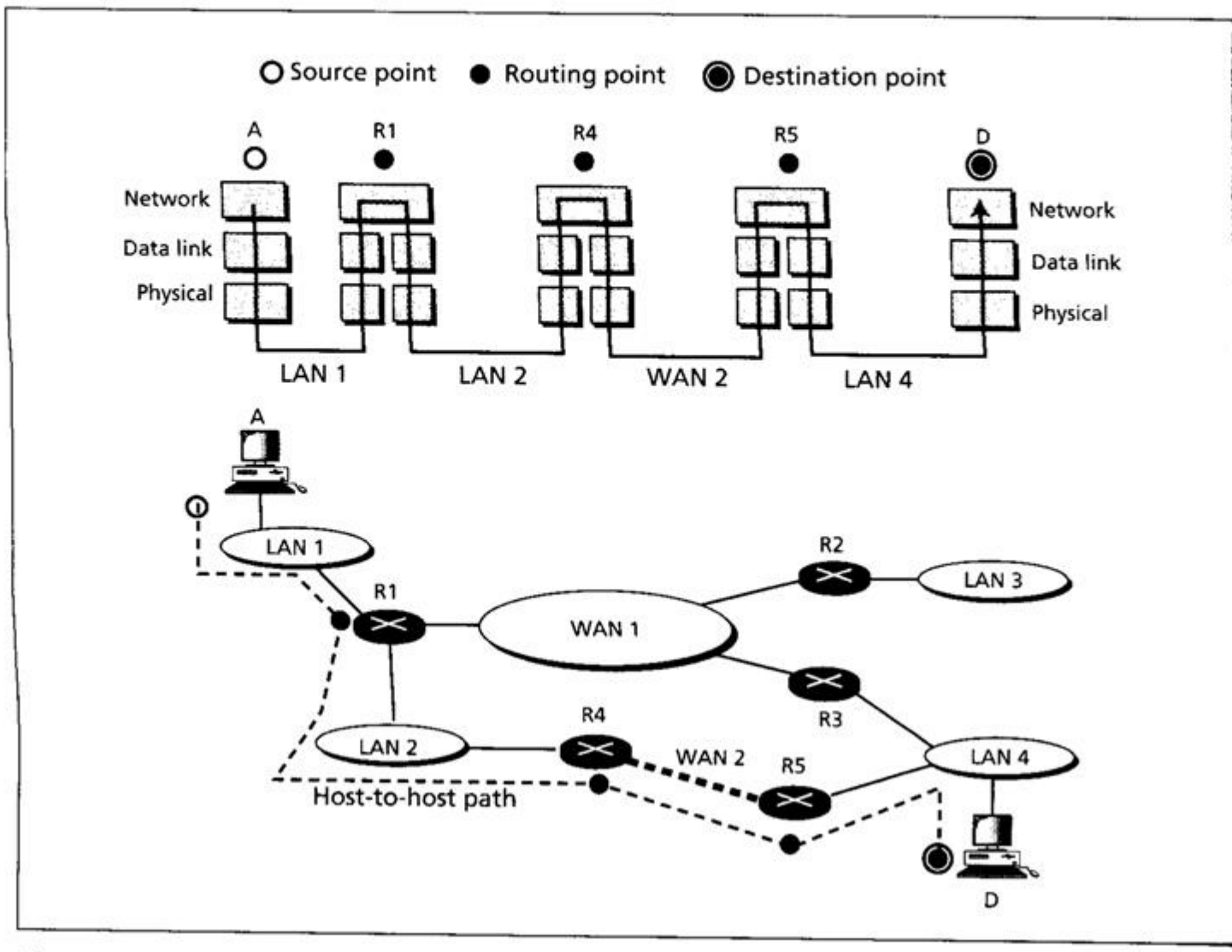


Figure 6.14 Routing at the network layer

Network layer protocols

The TCP/IP protocol suite supports one main protocol (IP) and several auxiliary protocols to help IP to perform its duties.

IP

In the TCP/IP protocol suite, the main protocol at the network layer is **Internet Protocol (IP)**. The current version is IPv4 (version 4) although IPv6 (version 6) is also in use, though not ubiquitously. IPv4 is responsible for the delivery of a packet from the source computer to the destination computer. For this purpose, every computer and router in the world is identified by a 32-bit IP address, which is presented in **dotted decimal notation**. The notation divides the 32-bit address into four 8-bit sections and writes each section as a decimal number between 0 and 255 with three dots separating the sections. For example, an IPv4 address 00001010 00011001 10101100 00001111 is written as 10.25.172.15 in dotted decimal notation.

At a message's source the IPv4 protocol adds the source and destination IP addresses to the packet passed from the application. The packet is then ready for its journey. However, the actual delivery is done by the data link and physical layer, as described shortly.

The address range of IPv4 (32 bits) can define up to 2^{32} (more than 4 billion) different devices. However, the way in which addresses have been allocated in the past has created *address depletion*. Several remedies have been designed to solve this problem, but experts foresee an imminent change in the structure of IP

addresses. The ultimate solution is IPv6 (partially in use today) in which the address is made up from 128 bits.

IP provides a **best-effort service**. It does not guarantee that packets will arrive error-free or in the order intended by the sender. It does not even guarantee that any packet is delivered: a packet may be lost for ever. Although this looks superficially like a terrible service, we can see that the Internet is working and doing its job. The situation is the same as the basic service offered by the post office. If we use the normal services offered by the post office, it does not guarantee that our letters will be received by the destination, but the system works almost all the time. If we need a guaranteed delivery, we can use other services offered by the post office, such as certified mail or a return receipt. The situation is the same with the Internet. Although IP is unreliable, an application can use the services of a reliable transport protocol, such as TCP, or implement error control itself to complement the service provided by IP.

Auxiliary protocols

Internet Protocol uses other auxiliary protocols to compensate for its deficiencies to some extent. Internet Control Message Protocol (ICMP) can be used to report a limited number of errors to the source computer. For example, if a router drops a packet because of congestion, ICMP can send a packet to the source to warn it of the congestion. ICMP can also be used to check the status of nodes in the Internet.

Internet Group Management Protocol (IGMP) can be used to add *multicasting* capability to IP. IP per se is a *unicast* protocol: one source, one destination. Multicasting enables IP to do multicasting: one source, many destinations. Other auxiliary protocols are used in the network layer, such as Address Resolution Protocol (ARP) and Reverse Address Resolution Protocol (RARP), that give other help to IP, but we must leave the discussion of these protocols to textbooks devoted to networking.

Data link layer

As we saw in the previous section, the network layer packet may pass through several routers in its journey from its source to its destination. Carrying the packet from one node to another (where a node can be a computer or a router) is the responsibility of the **data link layer**.

In our previous analogy, after the post office at Los Angeles airport decides that the letter should go to the post office at New York airport, the letter is delivered to the crew of the corresponding aircraft. Note however that the crew is only responsible for carrying the letter from Los Angeles to New York. The letter is probably put in a bag or a box with a label such as "from Los Angeles to New York".

The situation is the same on the Internet. In Figure 6.15, after the source decides that the packet should be sent to router R1 (the only possible router in this case), it encapsulates the packet in a **frame** and adds the link-layer address of router R1 in the packet's header as the destination address, and the link-layer address of computer A as the source address, then sends the packet. Every device connected to LAN 1 receives the frame, but only R1 opens it, because it recognizes its data-link layer address. This procedure is repeated between R1 and R4, between R4 and R5, and between R5 and D. Note that the reason we show separate boxes

for the data-link and physical layers in the upper part of the diagram is because each network (LAN and WAN) may operate under different protocols with different address or frame formats.

The data-link layer is responsible for node-to-node delivery of frames.

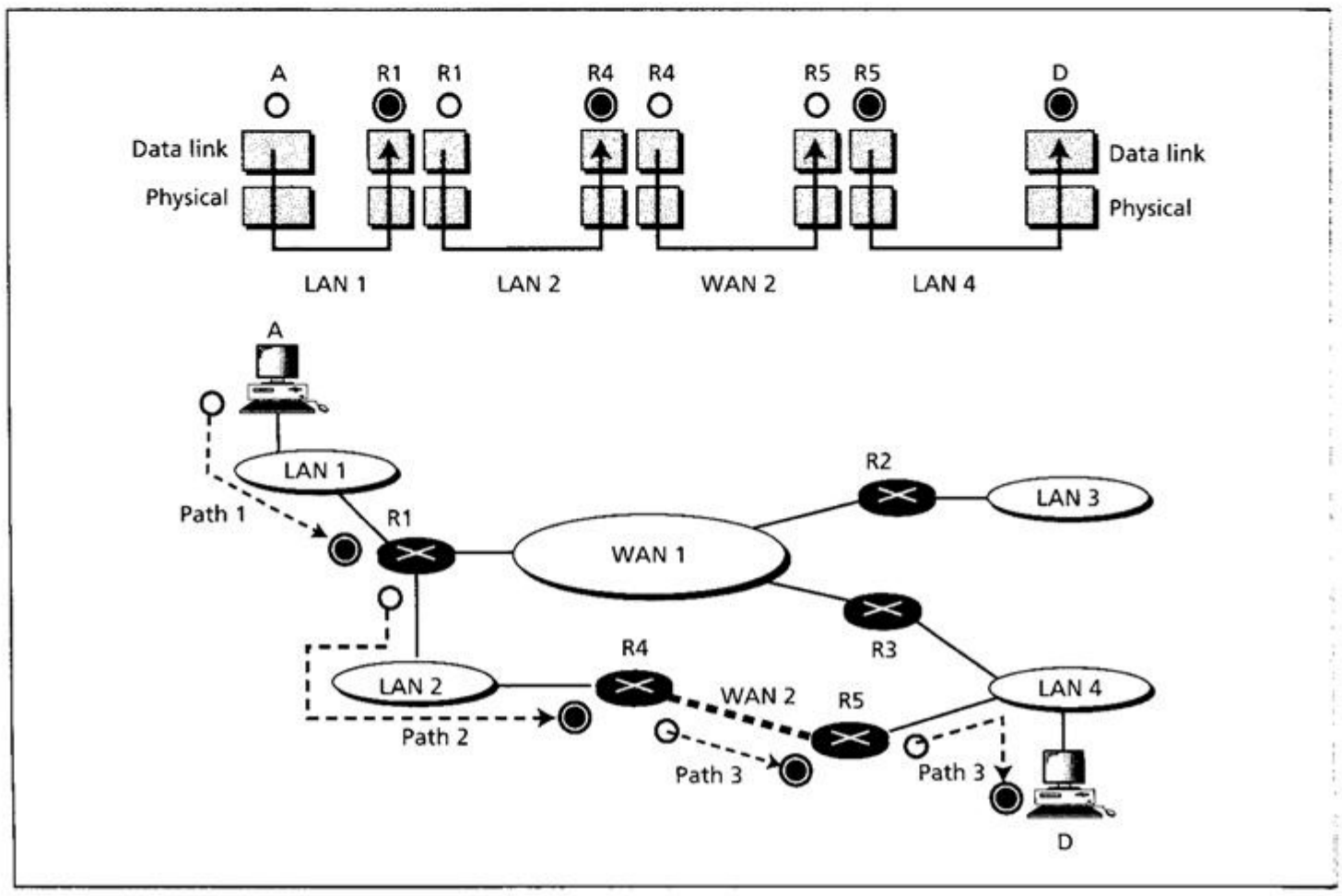


Figure 6.15 Communication at the data-link layer

Data-link layer addresses

Two questions that come to mind are how computer A knows the data-link layer address of router R1, or router R1 knows the data-link layer address of router R4. A device can find the data-link address of another device either statically or dynamically. In the static method, a device can create a table with two columns and store the network-layer link-layer addresses pairs. In the dynamic method, a device can broadcast a special packet containing the IP address of the next device and ask the neighbor with this IP address to send back its link address. Figure 6.16 shows the idea of data-link layer. In the figure, the data-link layer uses a static or dynamic process to find the data-link layer address of the next hop (router).

Unlike IP addresses, addresses at the data-link layer cannot be universal. Each data-link protocol may have a different address format and size. The Ethernet protocol, the most prevalent local area network in use today, uses a 48-bit address,

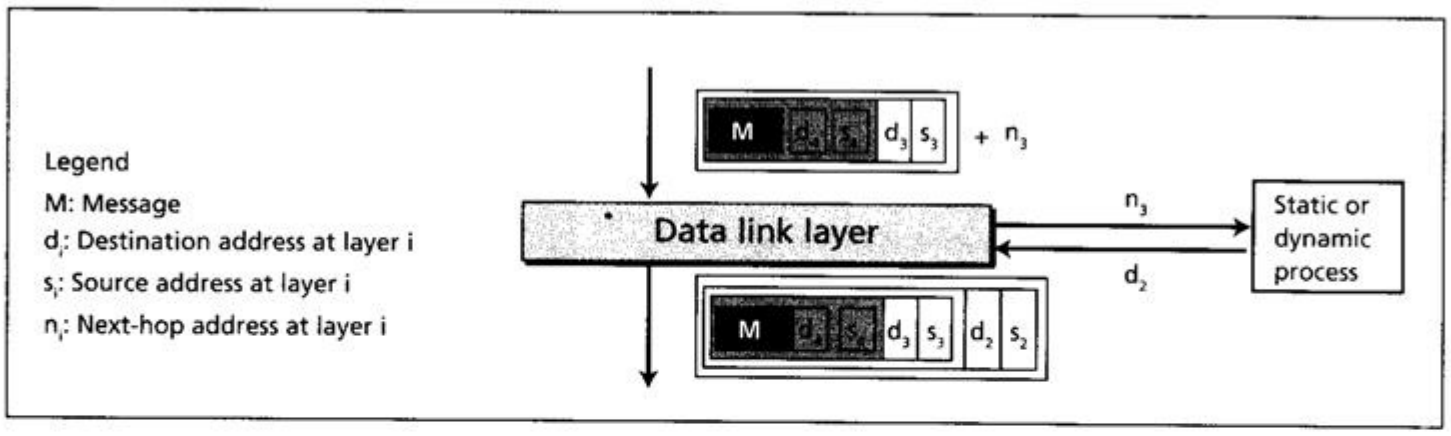


Figure 6.16 Addresses at the data-link layer

which is normally written in hexadecimal format (grouped in six sections, each with two hexadecimal digits) as shown below:

07:01:02:11:2C:5B

Data-link layer addresses are often called **physical addresses** or **media access control (MAC) addresses**.

d ntrol

Some data-link layer protocols use error control and flow control at the data-link layer. The procedure is the same as we discussed for the transport layer. However, it is implemented only between the outgoing point of a node and the incoming point of another node. This means that, for example, errors are checked several times, but none of the error checking covers errors that may happen inside a router. There may be no error from the outgoing port of router R1 to the incoming port of router R2, but what if the error happens within either of the routers? This is the reason why we need error control at the transport layer: to check the error from end to end.

Physical layer

The **physical layer** coordinates the functions required to carry a bit stream over a physical medium. Although the data-link layer is responsible for moving a frame from one node to another, the physical layer is responsible for moving the individual bits that make up the frame to the next node. In other words, the unit of transfer in the data link layer is a frame, while the unit of transfer in the physical layer is a bit. Each bit in the frame is transformed to an electromagnetic signal and propagated through the physical medium (wireless or cable). Note that there is no need for addressing in the physical layer: the propagation is broadcast. The signal sent from a device is received by other devices that are connected to the sending device by some means as long as there is no filtering to filter out the signal. For example, in a local area network, when a signal is sent by a computer or a router, all other computers and routers will receive it. Figure 6.17 shows the concept of the physical layer.

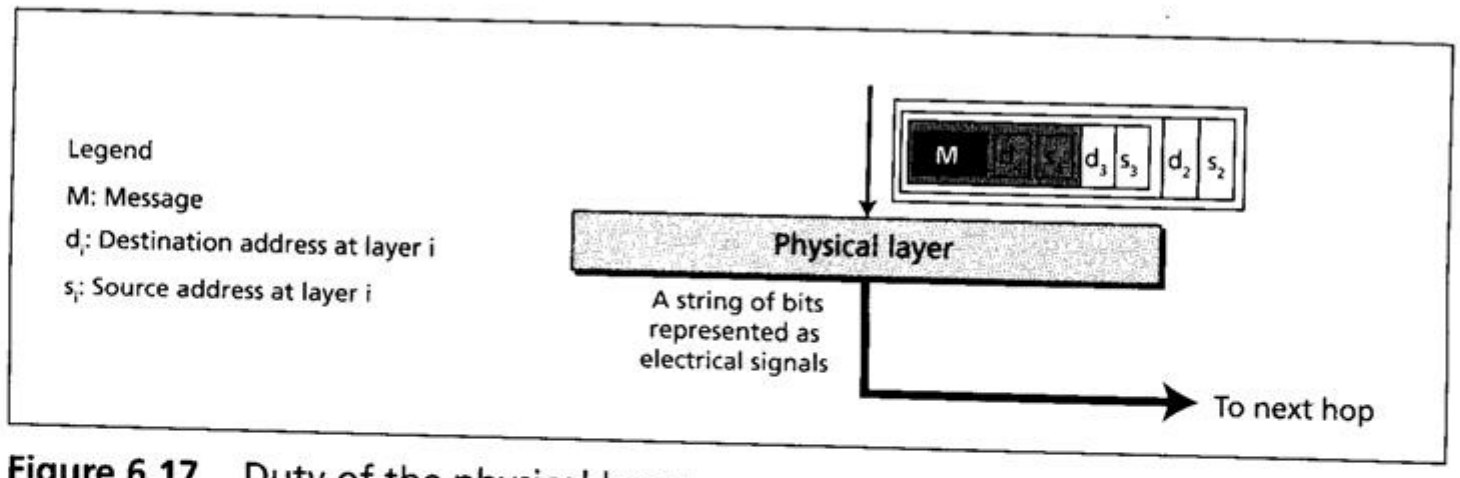


Figure 6.17 Duty of the physical layer

Summary of layers

Figure 6.18 summarizes the duties of each layer in the TCP/IP protocol and the addresses involved in each layer.

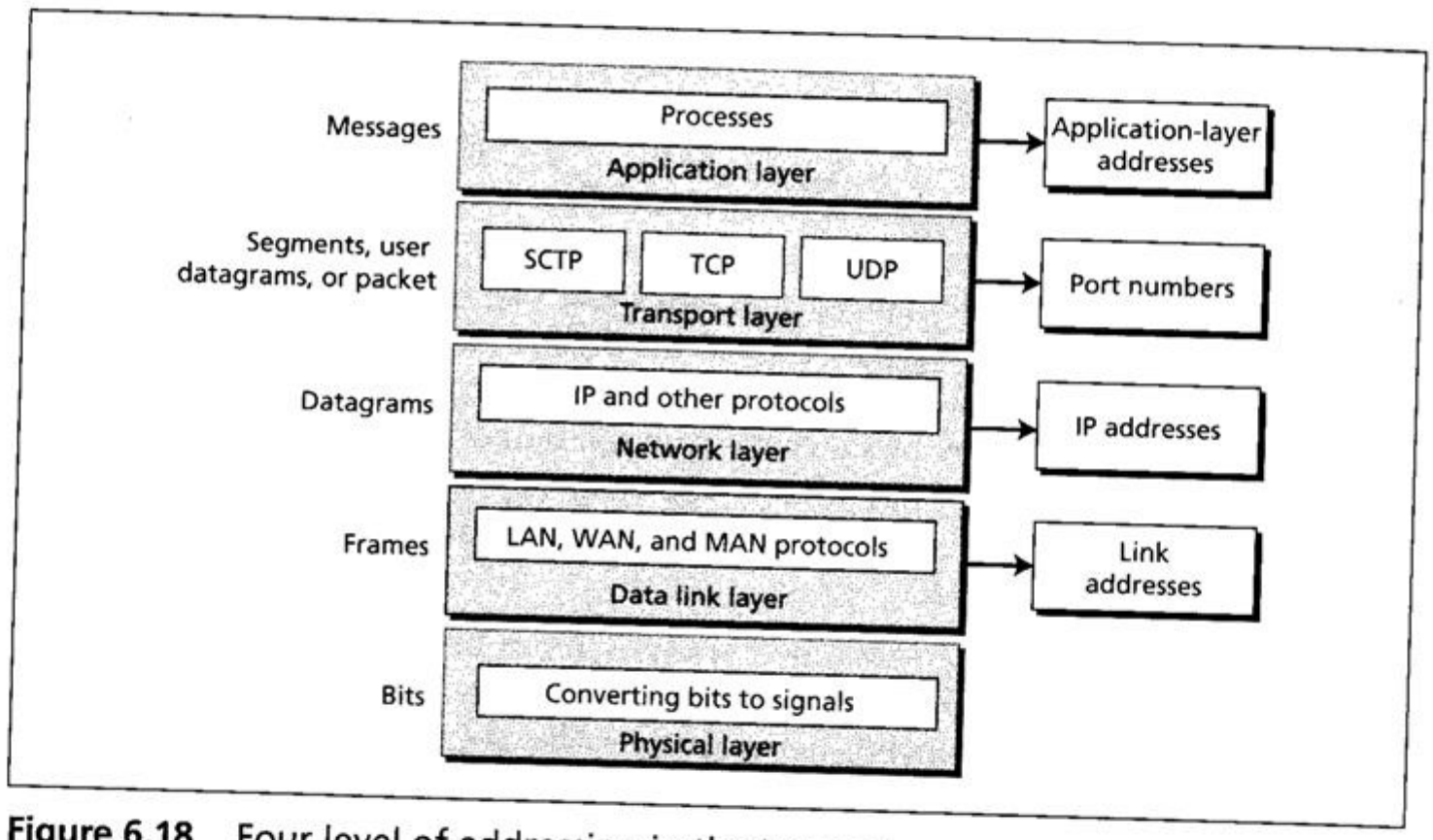


Figure 6.18 Four level of addressing in the Internet

The figure shows the unit of data in each layer. In the application layer, processes exchange messages. At the transport layer, the unit of data is referred to as a *segment* (TCP), a *user datagram* (UDP), or a packet (SCTP). At the network layer, the unit of data is referred to as a datagram. At the data-link layer, the unit of data is referred to as a frame. Finally, in the physical layer, the unit of data is a bit.

Figure 6.19 shows another aspect of the various layers: **encapsulation**. In the figure, D5 denotes the data unit at layer 5, D4 denotes the data unit at layer 4, and so on. The process starts at layer 5 (the application layer), then moves from layer to layer in descending, sequential order. At each layer, a **header**, or possibly a **trailer**, can be added to the data unit. Commonly, the trailer is added only at layer 2. When the formatted data unit passes through the physical layer (layer 1) it is changed into an electromagnetic signal and transported via a physical link.

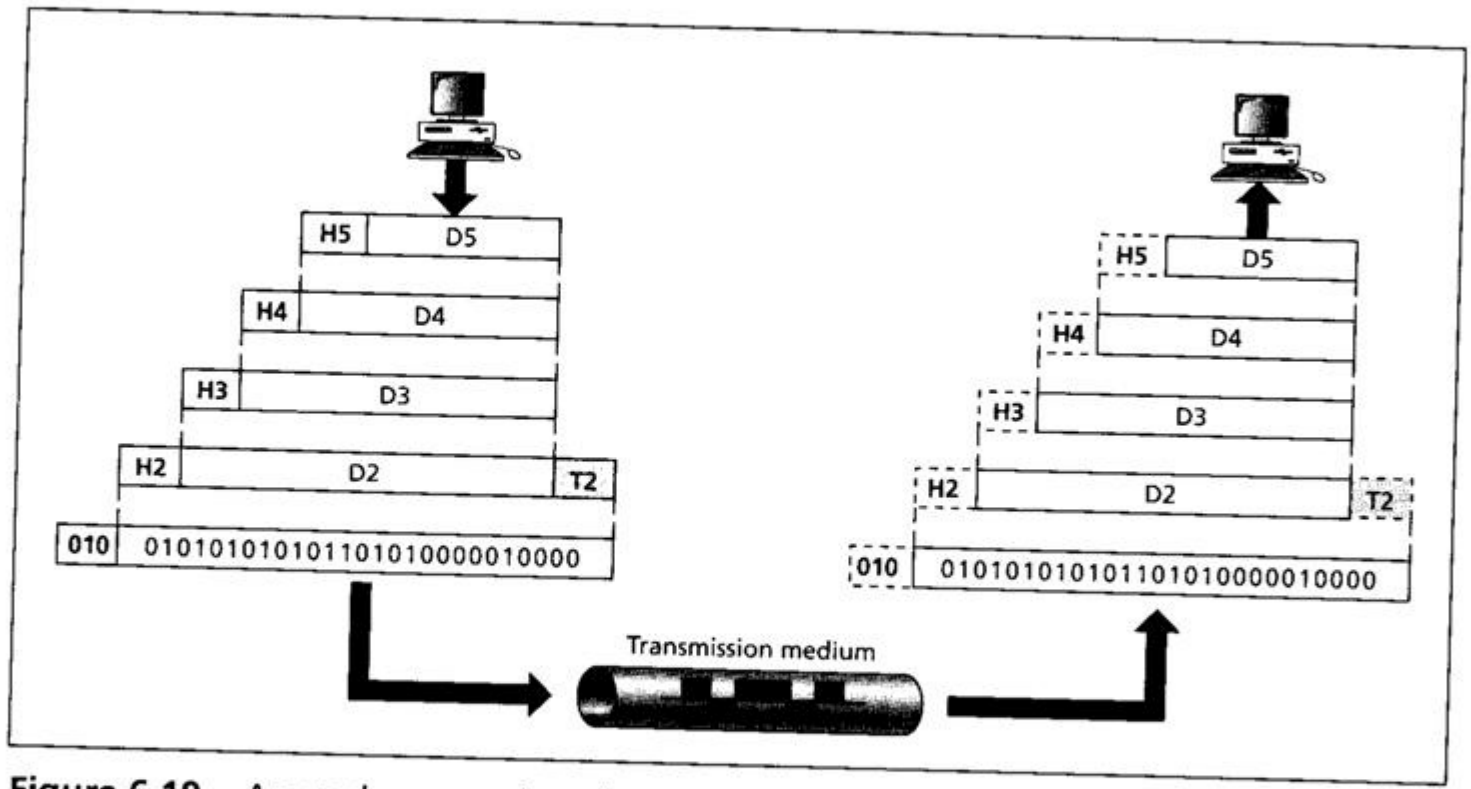


Figure 6.19 An exchange using the TCP/IP model

Upon reaching its destination, the signal passes into layer 1 and is transformed back into digital form. The data units then move back up through the layers. As each block of data reaches the next higher layer, the headers and trailers attached to it at the corresponding sending layer are removed, and actions appropriate to that layer are taken. By the time it reaches layer 5, the message is again in a form appropriate for the application and is made available to the recipient.

Note that the relationship between data units in the layers is not one-to-one. In other words, a message in layer 5 may not be necessarily encapsulated in a single segment in layer 4. In TCP, a message in layer 5 is divided into several sections and each section is encapsulated in one segment. In UDP, however, each message is encapsulated in one user datagram. This means that the messages delivered to UDP must be small enough to fit in one user datagram. The situation is the same when a segment or user datagram is encapsulated in a datagram (layer 4). A datagram in the network layer may also be split into several frames in the data link layer.

6.4 INTERNET APPLICATIONS

The main task of the Internet is to provide services for users. Among the most popular applications are electronic mail, remote login, file transfer, and accessing the World Wide Web (WWW). We briefly discuss these applications in this section.

e-mail

Let us first discuss **electronic mail (e-mail)**. Ironically, this first application that we discuss in this section cannot be supported by one client process and one server process: The reason is that e-mail is the exchange of messages between two entities. Although the sender of the e-mail can be a client program, the receiver cannot be the corresponding server, because that implies that the receiver must let their

computer run all the time, as they do not know when an e-mail will arrive. For this purpose, e-mail architecture is designed as shown in Figure 6.20.

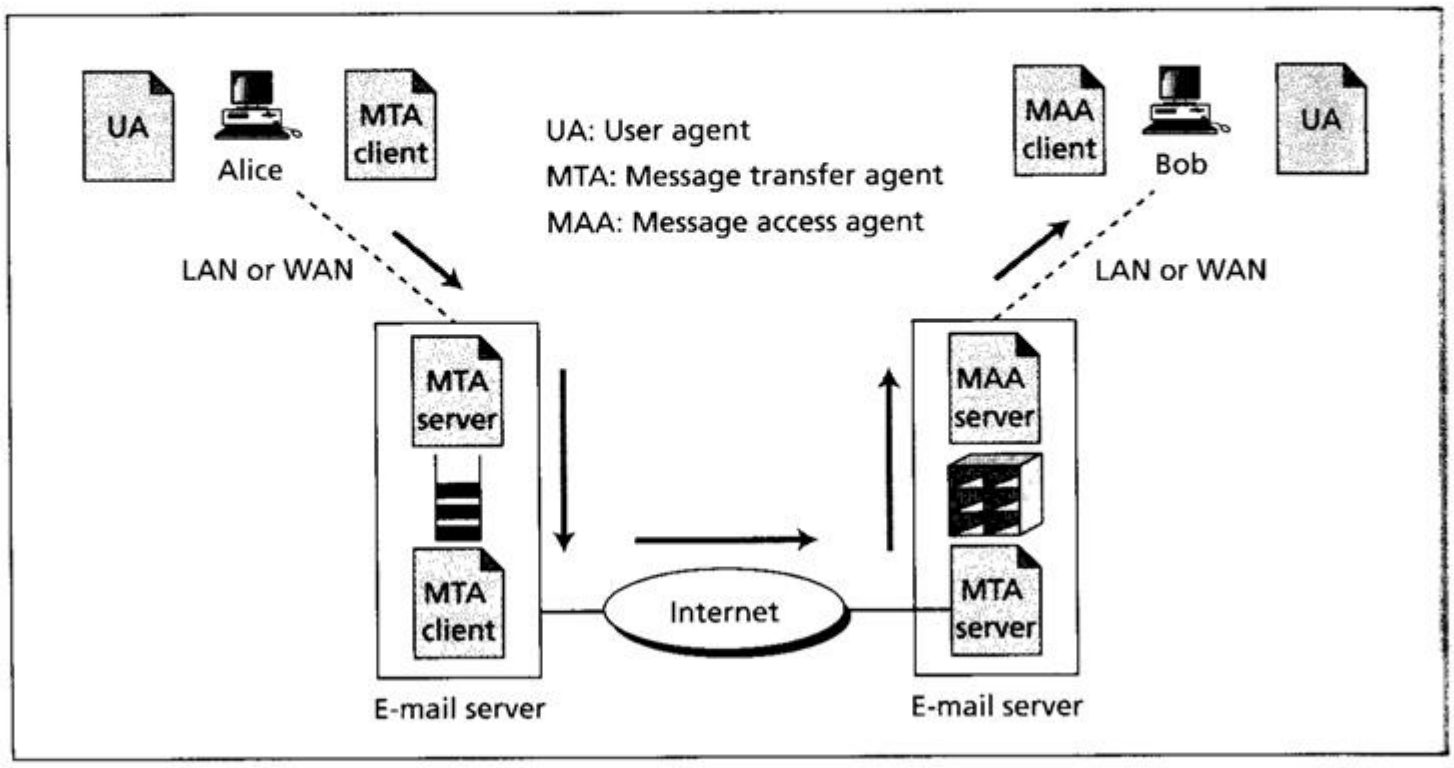


Figure 6.20 E-mail architecture

Assume that Alice is working in an organization that runs an e-mail server: every employee is connected to the e-mail server through a LAN. Alternatively, Alice could be connected to the e-mail server of an ISP through a WAN (telephone line or cable). Bob is also in one of the above two situations. The administrator of the e-mail server at Alice's site has created a queuing system that sends e-mail messages to the Internet one by one. The administrator of the e-mail server at Bob's site has created a mailbox for every user connected to the server: the mailbox holds the received messages until they are retrieved by the recipient. When Alice needs to send a message to Bob, she invokes a **user agent (UA)** program to prepare the message. She then uses another program, a **message transfer agent (MTA)**, to send the message to the mail server at her site. Note that the MTA is a client/server program with the client part installed on Alice's computer and the server part installed on the mail server. The application program that does the job of the MTA client and server is called **Simple Mail Transfer Protocol (SMTP)**. The message received at the mail server at Alice's site is queued with all other messages: each goes to its corresponding destination. In Alice's case, her message goes to the mail server at Bob's site. A client/server MTA is responsible for the e-mail transfer between the two servers. When the message arrives at the destination mail server, it is stored in Bob's *mailbox*, a special file that holds the message until it is retrieved by Bob. When Bob needs to retrieve his messages, including the one sent by Alice, he invokes another program, a **message access agent (MAA)**. The MAA is also designed as a client/server program with the client part installed on Bob's computer and the server part installed on the mail server.

There are several important points about the architecture of the e-mail system:

- Sending an e-mail from Alice to Bob is a store-and-retrieve activity. Alice can send an e-mail today: Bob, being busy, may check his e-mail three days later. During this time the e-mail is stored in Bob's mailbox on the server until it is retrieved.
- The main communication between Alice and Bob is through two application programs: the MTA client on Alice's computer and the MAA client on Bob's computer.
- The MTA client program is a *push* program: the client pushes (uploads) the message when Alice needs to send it. The MAA client program is a *pull* program: the client pulls (downloads) messages when Bob is ready to retrieve his e-mail.
- Alice and Bob cannot communicate directly using an MTA client at the sender site and an MTA server at the receiver site. This requires that the MTA server is running all the time, because Bob does not know when a message will arrive. This is not practical, because Bob probably turns off his computer when he does not need it.

access protocols

Stored e-mail remains on the mail server until it is retrieved by the recipient through an access protocol. Currently two e-mail access protocols are in common use: Post Office Protocol, Version 3 (POP3) and Internet Mail Access Protocol (IMAP).

POP

Post Office Protocol (POP) is simple but limited in functionality. The client POP software is installed on the recipient's computer and the server POP software is installed on the e-mail server.

Mail access starts with the client when the user needs to download their received e-mail from their mailbox on the mail server. The client (user agent) sends the user name and password to access the mailbox. The user can then list and retrieve the e-mail messages one by one.

POP has several deficiencies:

- It does not allow the user to organize e-mail on the server.
- The user cannot have different folders on the server, although of course the user can create folders on their own computer.
- POP does not allow the user to check the contents of their e-mail before downloading it.

IMAP

A second popular mail access protocol, **Internet Mail Access Protocol (IMAP)**, can handle these deficiencies. IMAP, although similar to POP, has more features and is more powerful and complex. IMAP provides the following extra functions. Users can:

- Check an e-mail header prior to downloading.
- Search the contents of their e-mail prior to downloading.

- Partially download e-mail. This is especially useful if the user has limited capacity and the e-mail volume is large.
- Create, delete, or rename mailboxes on the e-mail server.
- Create a hierarchy of mailboxes in a folder for e-mail storage.

Addresses

An e-mail handling system must have a unique addressing system to deliver mail. The addressing system used by SMTP consists of two parts: a *local part* and a *domain name*, separated by an @ sign (see Figure 6.21).

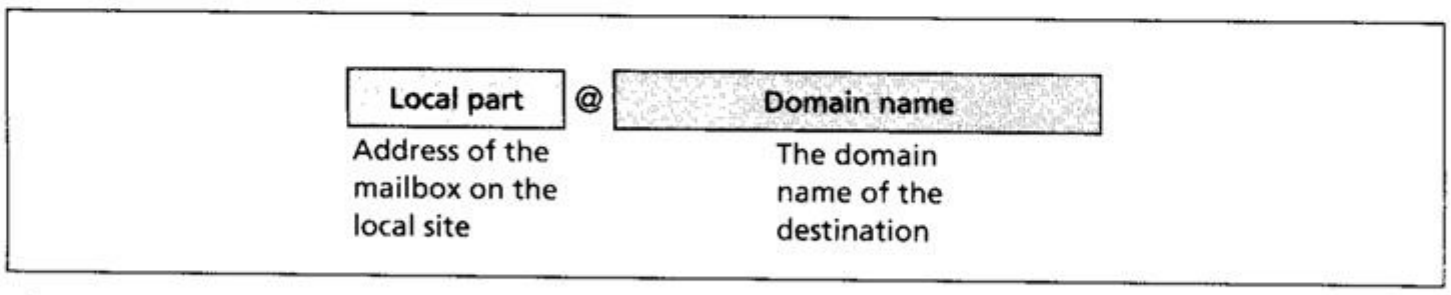


Figure 6.21 E-mail address composition

Local part

The local part defines the name of a special file, the user's *mailbox*, where all of the e-mail received for a user is stored for retrieval by the user's agent.

Domain name

The second part of the address is a name assigned to an IP address (a connection to the Internet). An organization usually selects one or more hosts to receive and send e-mail. The name assigned to each mail server, the **domain name**, comes from a universal naming system called the *domain name system* (DNS).

Multipurpose Internet Mail Extension (MIME)

As its name implies, SMTP is a simple mail transfer protocol. Its simplicity, however, comes with a price. SMTP can send messages only in NVT (*network virtual terminal*) 7-bit ASCII format. It cannot be used for languages that are not supported by 7-bit ASCII characters, such as French, German, Hebrew, Russian, Chinese, and Japanese. Also, it cannot be used to send binary files, video or audio data.

Multipurpose Internet Mail Extension (MIME) is a supplementary protocol that allows non-ASCII data to be sent through SMTP. MIME is not an e-mail protocol and cannot replace SMTP, it is only an extension to SMTP.

MIME transforms non-ASCII data at the sender's site into NVT ASCII data and delivers it to the client SMTP to be sent through the Internet. The server SMTP at the receiving side receives the NVT ASCII data and delivers it to MIME to be transformed back to the original data.

We can think of MIME as a set of software functions that transforms non-ASCII data into ASCII data and vice versa (see Figure 6.22).

File Transfer Protocol (FTP)

File Transfer Protocol (FTP) is the standard mechanism for one of the most common tasks on the Internet, copying a file from one computer to another.

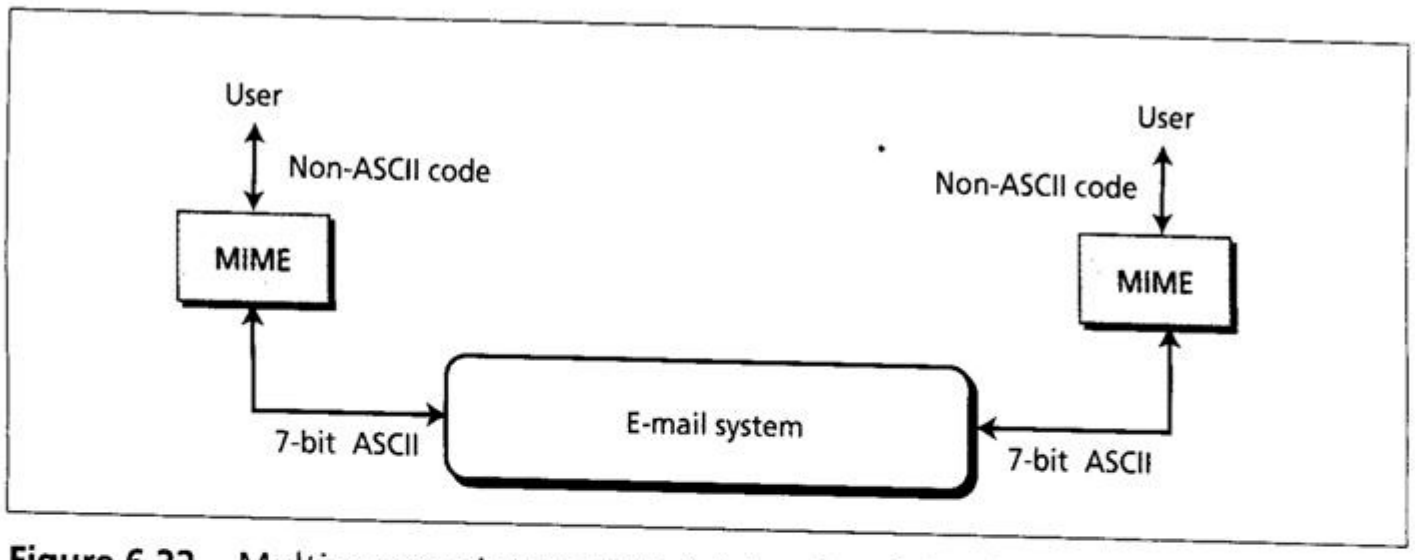


Figure 6.22 Multipurpose Internet Mail Extension (MIME)

Although file transfer from one system to another seems simple and straightforward, some problems must be dealt with first. For example, two systems may use different file naming conventions. Two systems may also have different ways to represent text and data, or different directory structures. All of these problems have been solved by FTP with a very simple and elegant approach.

FTP differs from other client-server applications in that it establishes two connections between the hosts. One connection is used for data transfer, the other for control information (commands and responses). Separation of commands and data transfer makes FTP more efficient. The control connection uses very simple rules of communication. We need to transfer only a line of command or a line of response at a time. The data connection, on the other hand, needs more complex rules due to the variety of data types transferred.

Figure 6.23 shows the basic model for FTP. The client has three components: the user interface, the client control process, and the client data transfer process. The server has two components: the server control process and the server data transfer process. The control connection is made between the control processes. The data connection is made between the data transfer processes.

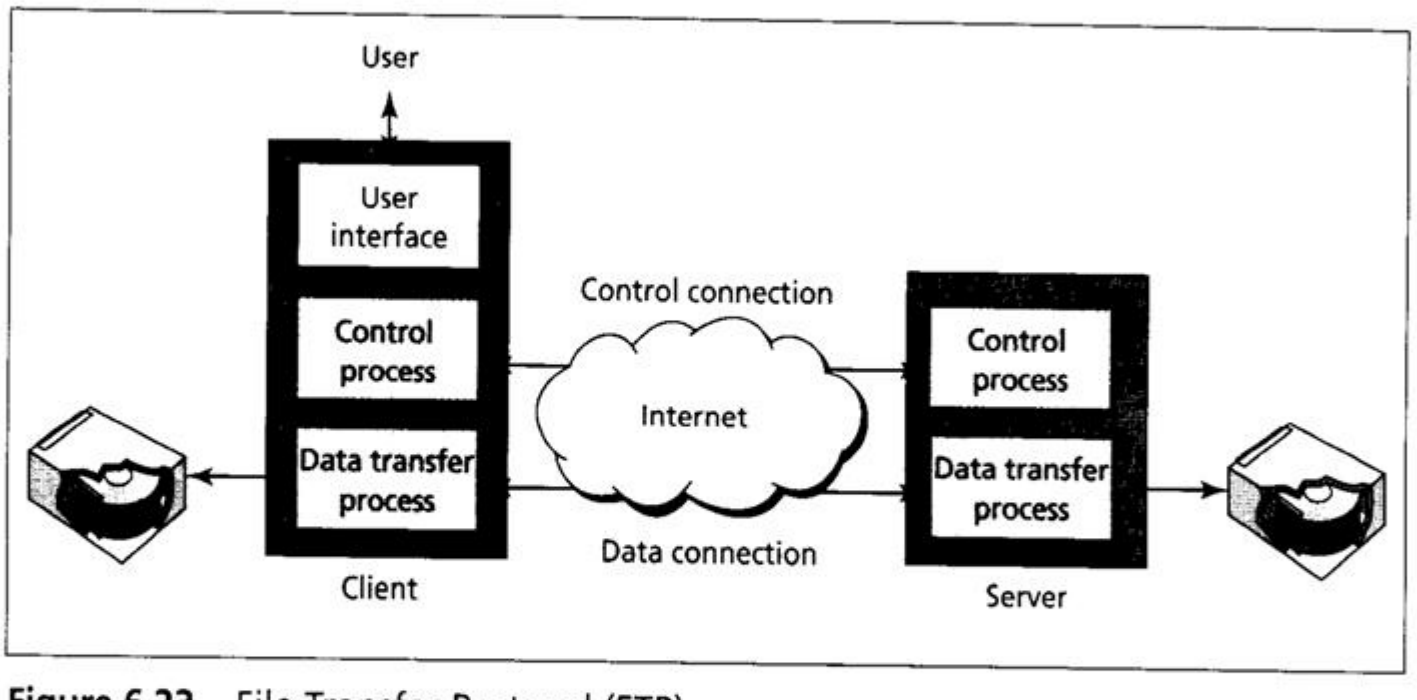


Figure 6.23 File Transfer Protocol (FTP)

The control connection remains open during the entire interactive FTP session, while the data connection is opened and then closed for each file transferred. It opens each time commands that involve transferring files are used, and it closes after the file is transferred. The two FTP connections, control and data, use different strategies and different port numbers.

Closed and open FTP

Sites that run FTP servers to allow users to access files use one of two strategies: a site is either closed or open to the public. A closed FTP site allows only specific users to access files: access is controlled by the user account and password and the general public are not allowed to access files at such sites.

An open FTP site allows anyone to access files. These sites offer **anonymous FTP**. Users can use *anonymous* as the user name and *guest* as the password. User participation in this system is very limited. Some sites allow anonymous users only a subset of commands. For example, most sites allow users to copy files, but do not allow navigation through the file system.

There are occasions when we need simply to copy a file without the need for all of the functionalities of FTP. **Trivial File Transfer Protocol (TFTP)** is designed for these types of file transfer. It makes a single connection and transfers a small file quickly. However, this application is not universally available.

Remote login – TELNET

The main task of the application layer is to provide services for users. For example, users want to be able to run different application programs at a remote site and create results that can be transferred to their local site. One way to satisfy these demands is to create different client-server application programs for each desired service. Programs such as file transfer programs (FTP and TFTP), e-mail (SMTP), and so on are already available. However, it would be impossible to write a client-server program for each specific application.

TELNET is a general-purpose client-server program that lets a user access any application program on a remote computer. In other words, it allows the user to log onto a remote computer. After logging on, a user can use the services available on the remote computer and transfer the results back to the local computer.

TELNET is an abbreviation for *terminal network*. TELNET enables the establishment of a connection to a remote system in such a way that the local terminal appears to be a terminal connected directly to the remote system.

Local login

When a user logs onto a local time-sharing system, it is called *local login*. As the user types at a terminal or a workstation running a terminal emulator, the keystrokes are accepted by the terminal driver. The terminal driver passes the characters to the operating system. The operating system, in turn, interprets the combination of characters and invokes the desired application program or utility (see Figure 6.24).

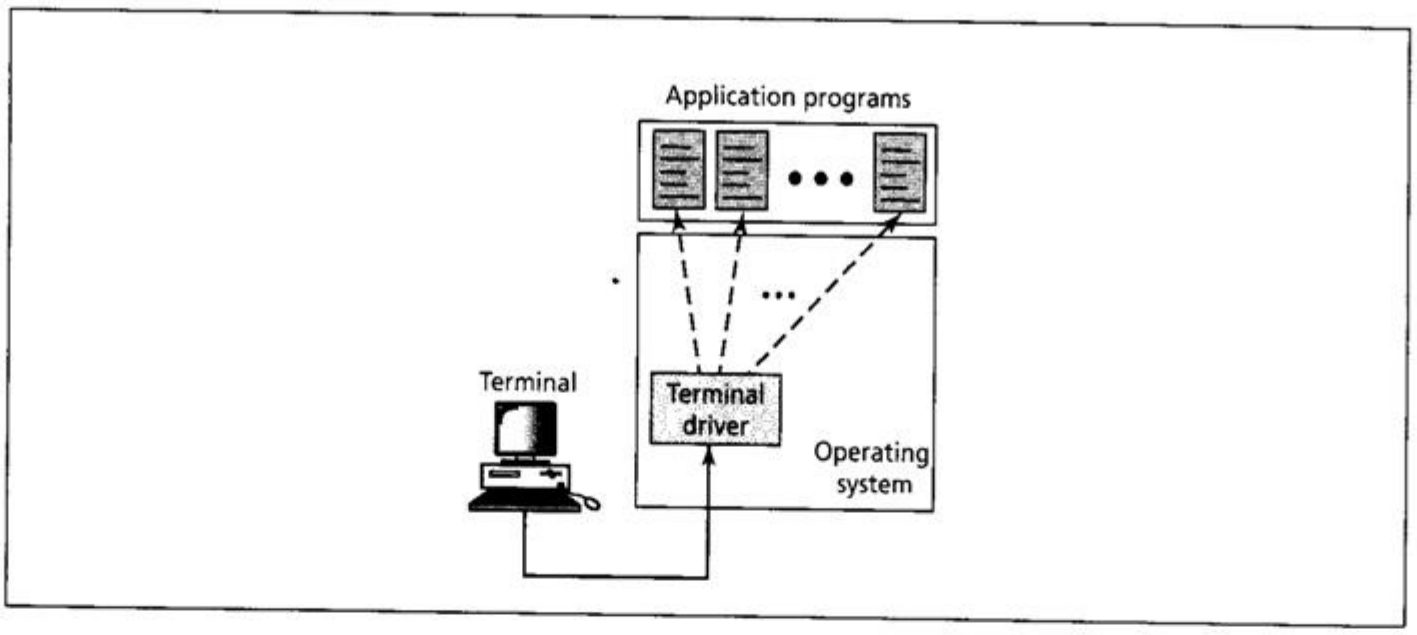


Figure 6.24 Local login

ote login The mechanism for remote login is not as simple as local login. The operating system may assign special meanings to special characters. For example, in UNIX some combinations of characters have special meanings, such as the combination of the control character with the character "z" to mean "suspend", the combination of the control character with the character "c" to mean abort, and so on. Whereas these special situations do not create any problems in local login, because the terminal emulator and the terminal driver know the exact meaning of each character or combination of characters, they may create problems in remote login. For example, which process should interpret special characters: the client or the server?

When a user wants to access an application program or utility located on a remote machine, they perform a remote login. Here the TELNET client and server programs come into use. The user sends the keystrokes to the terminal driver, where the local operating system accepts them but does not interpret them. The ASCII characters are then sent to the TELNET client, which transforms each character into a character in a universal character set called *network virtual terminal* (NVT) characters and delivers them to the local TCP/IP stack. At the server site, the TELNET server transforms each NVT character back to ASCII and then delivers it to a pseudo-terminal driver (see Figure 6.25).

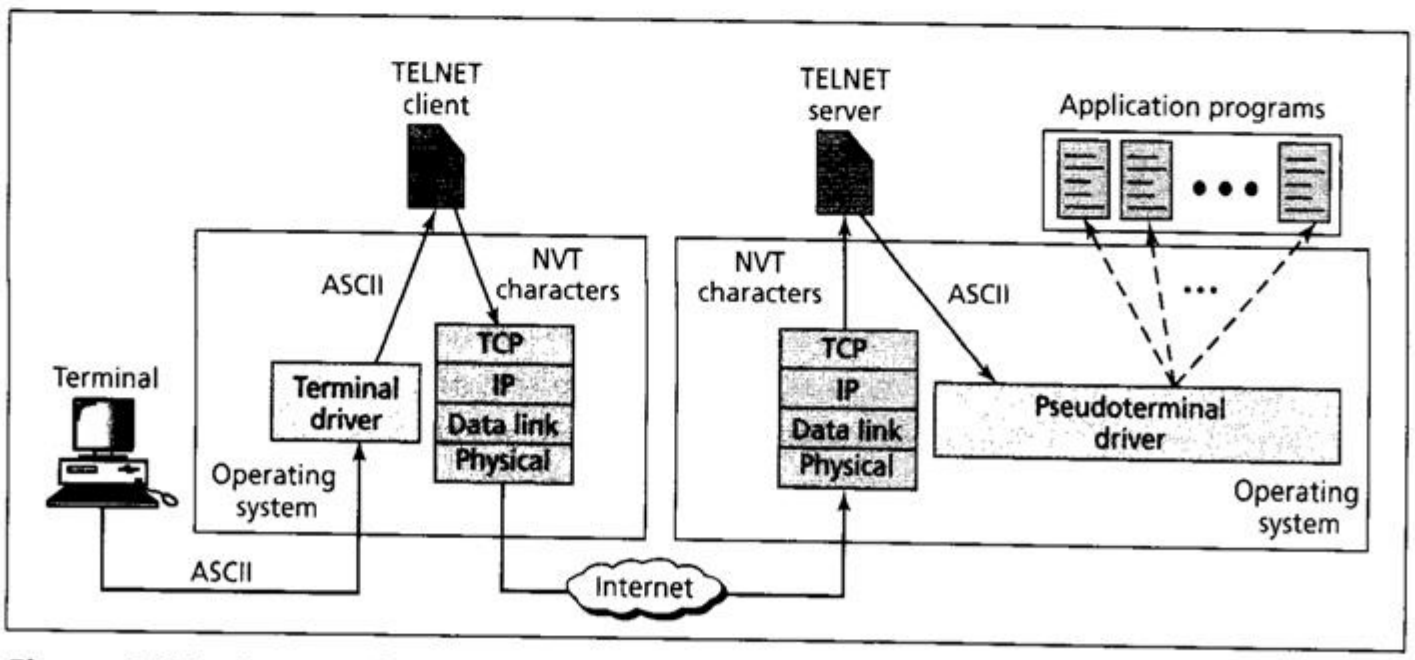


Figure 6.25 Remote login

The World Wide Web (WWW)

The **World Wide Web (WWW)**, or just "the web", is a repository of linked information spread all over the world. The WWW has a unique combination of flexibility, portability, and user-friendly features that distinguish it from other services provided by the Internet. The WWW today is a distributed client-server service in which a client using a browser can access a service using a server. However, the service provided is distributed over many locations, called *web sites*.

The idea of the World Wide Web started in 1989 at the European Particle Physics Laboratory in Geneva, Switzerland. Tim Berners-Lee needed to create a large database for physics research, which he found impossible to do using a single computer. The obvious solution was to let each piece of information be stored on an appropriate computer and let the computers be linked together through hypertext. In 1993, the University of Illinois, under the supervision of Marc Andreessen, created the first graphical browser, called Mosaic. In 1994, Andreessen and some colleagues started Netscape. Another widely used browser is Microsoft Explorer.

Hypertext and hypermedia

The WWW uses the concept of *hypertext* and *hypermedia*. In a hypertext environment, information is stored in a set of documents that are connected together using the concept of *links*. An item can be associated with another document using a link. The reader who is browsing through the document can move to other documents by choosing (clicking) on the items that are linked to other documents. Figure 6.26 shows the concept of hypertext.

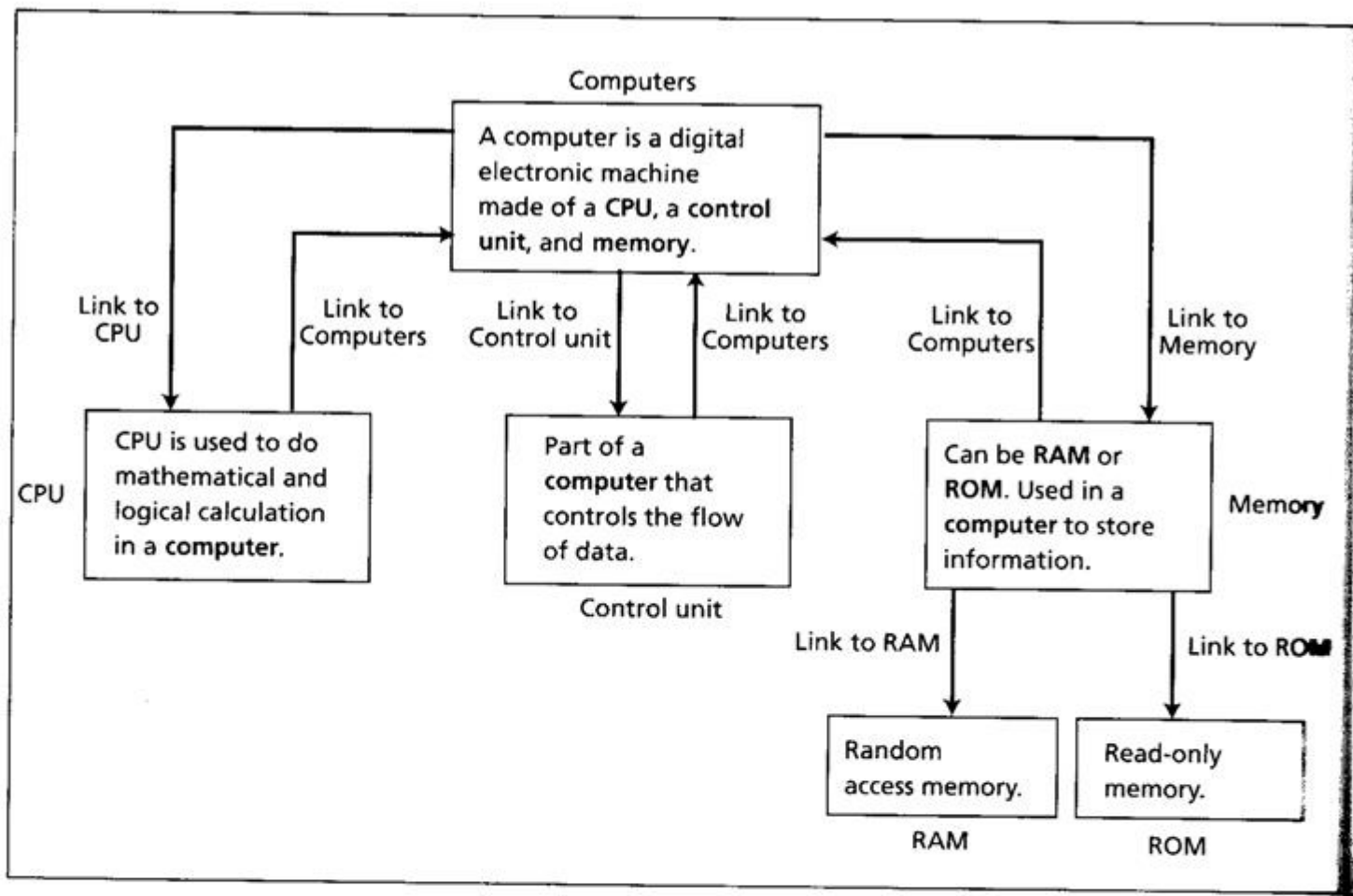


Figure 6.26 Hypertext

There are five documents in this hypothetical example: *Computers*, *CPU*, *Memory*, *ROM*, and *RAM*. A user who is browsing the Computer document can click on the

CPU link to find information about CPUs. A user who is browsing in the CPU document can click on the computer link to read more about computers. All five documents are somehow linked together.

Whereas hypertext documents contain only text, hypermedia documents can contain pictures, graphics, and sound. A unit of hypertext or hypermedia available on the web is called a *page*. The main or root page for an organization or an individual is known as a *home page*.

Components of the WWW

To use the WWW we need three components: a browser, a web server, and a protocol called Hypertext Transfer Protocol (HTTP).

Browser

A variety of vendors offer commercial browsers that interpret and display a web document, and all of them use nearly the same architecture. Each browser usually consists of three parts: a controller, client programs, and interpreters. The controller receives input from the keyboard or the mouse and uses the client programs to access the document. After the document has been accessed, the controller uses one of the interpreters to display the document on the screen. The client program can be one of the protocols described previously, such as FTP or TELNET, but it is usually HTTP. The interpreter handles the languages used today on the Internet, such as HTML or Java (see Figure 6.27).

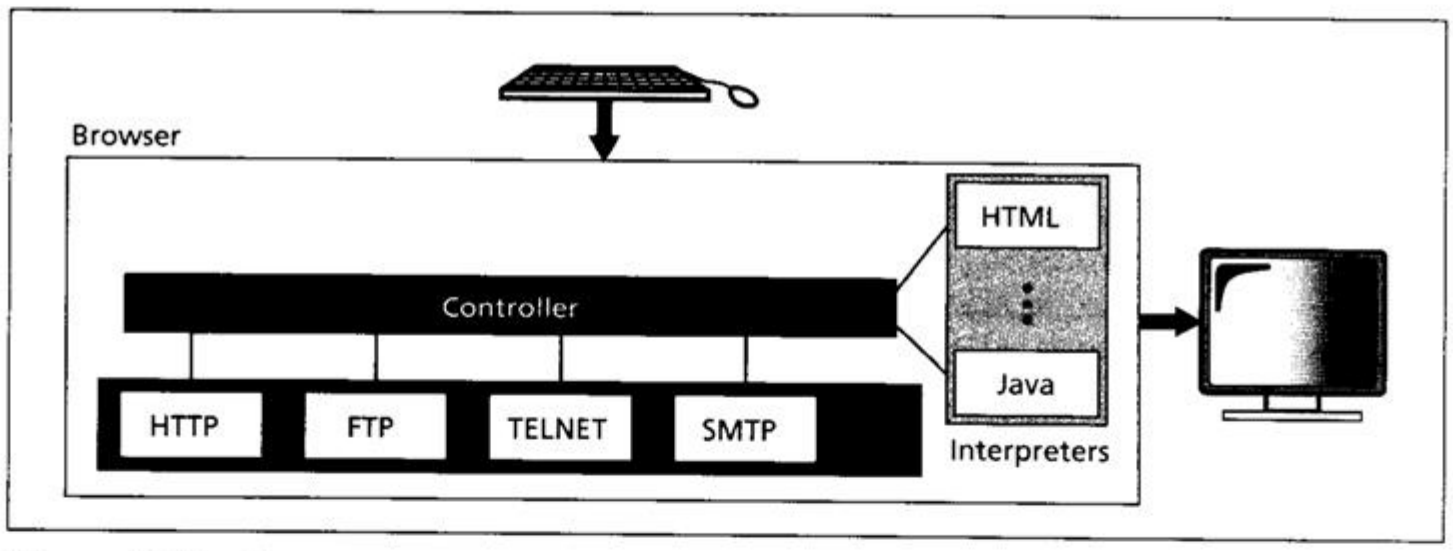


Figure 6.27 Browser structure

Server

The server stores all pages belonging to a web site.

Hypertext Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) is a protocol used mainly to access data on the World Wide Web. The protocol transfers data in the form of plain text, hypertext, audio, video, and so on. It is called Hypertext Transfer Protocol because its efficiency allows its use in a hypertext environment and enables rapid jumps from one document to another.

The idea of HTTP is very simple. A client sends a request, which looks like e-mail, to the server. The server sends the response, which looks like an e-mail

reply, to the client. The request and response messages carry data in the form of a letter with a MIME-like format.

The commands from the client to the server are embedded in a text-based request message. The contents of the requested file or other information are also embedded in a text-based response message.

Figure 6.28 illustrates an HTTP transaction between the client and server. The client initializes the transaction by sending a request message. The server replies by sending a response.

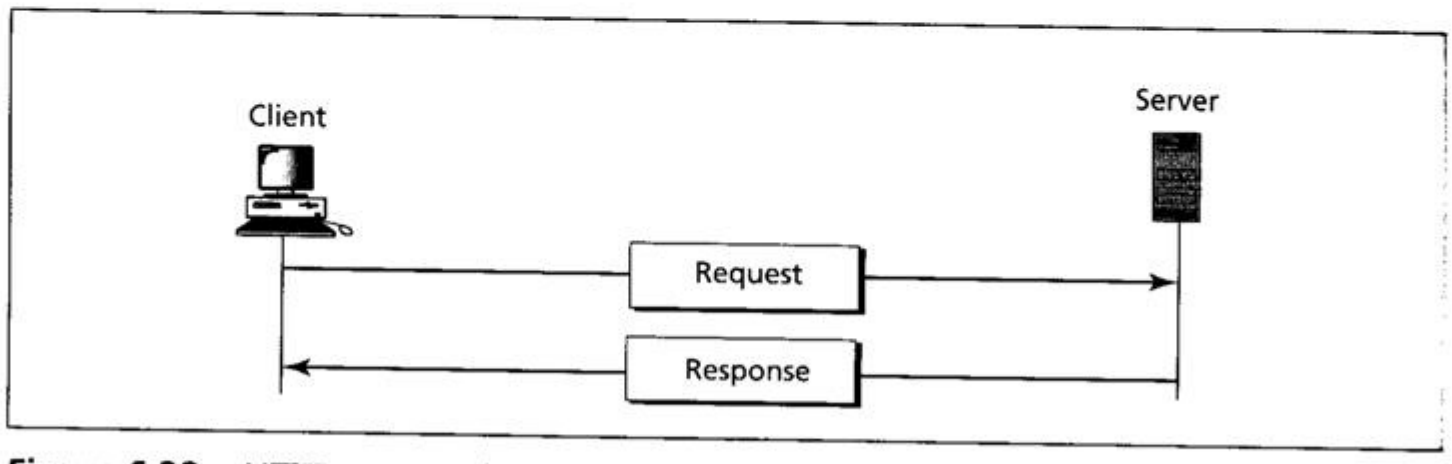


Figure 6.28 HTTP transaction

Addresses

A client that wants to access a document needs an address. To facilitate the access of documents distributed throughout the world, HTTP uses the concept of *locators*. The uniform resource locator (URL) is a standard for specifying any kind of information on the Internet. The URL defines four things: method, host computer, port, and path (see Figure 6.29).

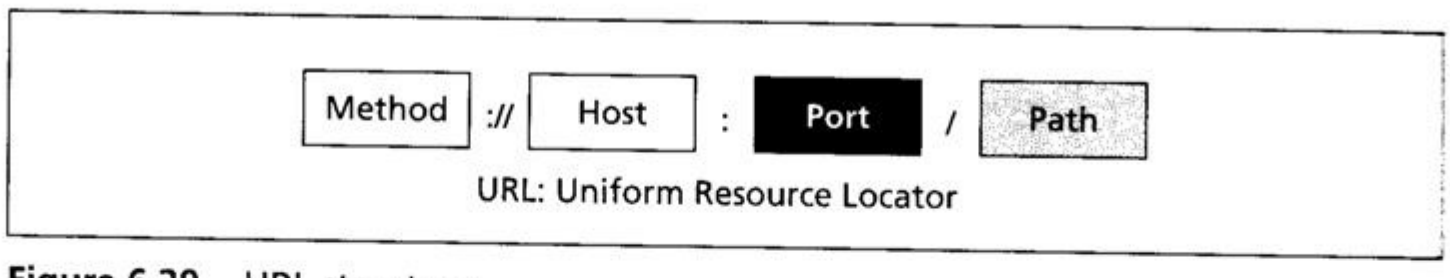


Figure 6.29 URL structure

The *method* is the protocol used to retrieve the document, for example, HTTP. The *host* is the computer on which the information is located, although the name of the computer can be an alias. Web pages are usually stored on computers, and computers are given alias names that usually begin with the characters "www." This is not mandatory, however, as any name can be given to the computer that hosts the web page.

The URL optionally can contain the port number of the server. If the port is included, it is inserted between the host and the path, and it should be separated from the host name by a colon.

The *path* is the pathname of the file where the information is located. Note that the path can itself contain slashes that, in the UNIX operating system, separate the directories from the subdirectories and files.

The documents on the WWW can be grouped into three broad categories: static, dynamic, and active. This categorization is based on the time at which the contents of the documents are determined. **Static documents** are fixed-content documents that are created and stored on a server. The client can only get a copy of the document. In other words, the contents of the file are determined when the file is created, not when it is used. Of course, the contents of the server can be changed, but the user cannot change them. When a client accesses the document, a copy of the document is sent. The user can then use a browser to display the document.

HTML

Hypertext Markup Language (HTML) is a language for creating web pages. The term *markup language* comes from the book publishing industry. Before a book is typeset and printed, a copy editor reads the manuscript and puts marks on it. Let us clarify the idea with an example. To make part of a text display in boldface with HTML, we put beginning and ending boldface tags (marks) in the text. The two tags <B> and </B> are instructions for the browser. When the browser sees these marks, it knows that the text between them must be set in boldface. A markup language such as HTML allows us to embed formatting instructions in the file itself. The instructions are included with the text. In this way, any browser can read the instructions and format the text according to the specific computer on which the browser runs.

We might ask why we do not use the formatting capabilities of word processors to create and save formatted text. The answer is that different word processors use different techniques or procedures for formatting text. For example, imagine that a user creates formatted text on a Macintosh and stores it in a web page. Another user who uses an IBM computer would not be able to receive the web page, because the two computers use different formatting procedures. HTML lets us use only ASCII characters for both the main text and the formatting instructions. In this way, every computer can receive the whole document as an ASCII document. The main text is the data, and the formatting instructions can be used by the browser to format the data.

A web page is made up of two parts: the head and the body. The head is the first part of a web page. The head contains the title of the page and other parameters that the browser will use. The actual contents of a page are in the body, which includes the text and the tags. Whereas the text is the actual information contained in a page, the tags define the appearance of the document. Figure 6.30 shows the format of an HTML document.

When a browser encounters the document shown on the left of the figure, something similar to what is shown on the right is displayed. Of course, the real format, such as title, font size, or indention depends on the browser.

Another interesting tag category is the image tag. Such nontextual information as digitized photos or graphic images is not a physical part of an HTML document. But we can use an image tag to point to the file containing the photo or image. The

```
<HTML>
  <HEAD>
      <TITLE> Small College </TITLE>
  </HEAD>
  <BODY>
   <H1> President: Eve Jones </H1>
   <H1> Vice President: Adam Brown </H1>
   <H2> Department: Math and Science </H2>
   <H3> Courses </H3>
     <LI> CS0 </LI>
     <LI> CS1 </LI>
     <LI> CS2 </LI>

   <H2> Department: Business </H2>
   <H3> Courses </H3>
     <LI> Introduction to Business </LI>
     <LI> Business Law </LI>
     <LI> Business Mathematics </LI>
  </BODY>
</HTML>
```

HTML document

Small College

President: Eve Jones
Vice President: Adam Brown
Department: Math and Science
Courses:
1. CS0
2. CS1
3. CS2
Department: Business
Courses:
1. Introduction to Business
2. Business Law
3. Business Mathematics

Result

Figure 6.30 HTML example

image tag defines the address (URL) of the image to be retrieved. It also specifies how the image should be inserted after retrieval.

A third interesting category is the hyperlink tag, which is needed to link documents together. Any item (word, phrase, paragraph, or image) can refer to another document through a mechanism called an *anchor*. An anchor is defined by <A ... > and </A> tags, and the anchored item uses the URL to refer to another document. When the document is displayed, the anchored item is underlined, blinking, or boldfaced. The user can click on the anchored item to go to another document, which may or may not be stored on the same server as the original document.

XML

HTML adds formatting capability to a document, but it does not define the type of data. For example, HTML can define that a line of text is intended as an item in a list, but it does not define the nature of the item. **Extensible Markup Language (XML)** is a language in which tags can be used to define the content (type) of the text between two tags. In HTML, the tags define the *format* and the text defines the *value*. In XML, the tags define the *type* and the text defines the *value*. Figure 6.31 compares a line of HTML with a corresponding line of XML.

The two tags <LI> and </LI> in HTML are predefined tags for HTML: the two tags <Course> and </Course> in the XML example are user-defined tags

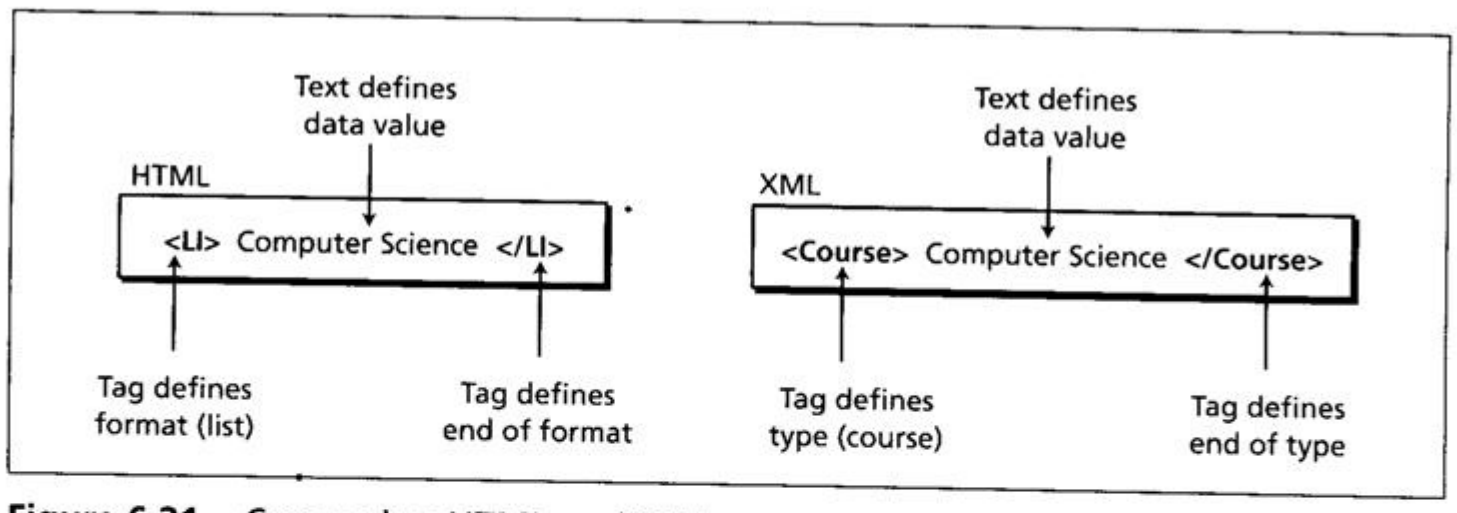


Figure 6.31 Comparing HTML and XML

created for this particular language. Another document can use different tags to denote the same type of data.

In the HTML document, the text "Computer science" can be the name of a course or the name of a computer scientist's dog: HTML cannot tell the difference. In the XML document, by contrast, the same text value is defined as the title of a course by its tags.

Why do we need to define the type of data in a web page? One reason is that it makes searching easier. For example, a search engine such as Google, which needs to find whether a particular university offers computer science courses, needs to look for the text value "Computer science" in the all corresponding HTML documents. The same search engine could limit the search only to lines tagged with <Course> in a corresponding XML document. An XML document is therefore faster to search than an HTML document.

XML also allows different organizations to define their own applications. For example, Chemistry Markup Language, CML, uses a set of tags to define the structures of materials, MathML defines the set of tags to display mathematical symbols and relations, and so on.

namic uments

A **dynamic document** is created by a web server whenever a browser requests the document. When a request arrives, the web server runs an application program that creates the dynamic document. The server returns the output of the program as a response to the browser that requested the document. Because a fresh document is created for each request, the contents of a dynamic document can vary from one request to another. A very simple example of a dynamic document is the retrieval of the time and date from a server. Time and date are dynamic information in that they change from moment to moment. The client can ask the server to run a program such as the date program in UNIX and send the result of the program to the client.

Common Gateway Interface (CGI)

The Common Gateway Interface (CGI) is a technology that creates and handles dynamic documents. CGI is a set of standards that defines how a dynamic document is written, how data is input to the program, and how the output result is used. CGI is not a new language: instead, it allows programmers to use any of

several languages such as C, C++, or the Bourne shell, Korn shell, C shell, Tcl, or Perl scripting languages.

Scripting technologies for dynamic documents

The problem with CGI technology is the inefficiency that results if part of the dynamic document that is to be created is fixed and does not change from request to request. For example, assume that we need to retrieve a list of spare parts, their availability, and prices for a specific make of car over the web. Although availability and prices vary from time to time, the name, description, and the picture of the parts are fixed. If we use CGI, the program must create an entire document each time a request is made.

The solution is to create a file containing the fixed part of the document using HTML and embed a script, a type of source code, that can be run by the server to provide the varying availability and price data. Several technologies have been employed to create dynamic documents using scripts. Among the most common are **Hypertext Preprocessor (PHP)**, which uses the Perl language, **Java Server Pages (JSP)**, which uses the Java language for scripting, **Active Server Pages (ASP)**, a Microsoft product which uses Visual Basic language for scripting, and **ColdFusion**, which embeds SQL database queries in the HTML document.

Active documents

For many applications we need to be able to run a program or a script at the client site. These are called **active documents**. For example, suppose we want to run a program that creates animated graphics on the screen, or a program that interacts with the user. The program definitely needs to be run on the client computer where the animation or interaction takes place. When a browser requests an active document, the server sends a copy of the document or a script. The document is then run under the control of the client's browser.

Java applets

One way to create an active document is to use **Java applets**. Java is a combination of a high-level programming language, a runtime environment, and a class library that allows a programmer to write an active document (an applet) and a browser to run it. An applet is a program written in Java and saved on the server, compiled and ready to be run. The document is in byte-code (binary) format. The client process (the browser) creates an instance of this applet and runs it.

A Java applet can be run by the browser in two ways. In the first method, the browser can directly request the Java applet program in the URL and receive the applet in binary form. In the second method, the browser can retrieve and run an HTML file that has embedded the address of the applet as a tag.

JavaScript

The idea of scripts in dynamic documents can also be used for active documents. If the active part of the document is small, it can be written in a scripting language, which is then interpreted and run by the client. The script is in source code (text) and not binary form. The scripting technology used in this case is usually **JavaScript**. JavaScript, which bears a small resemblance to Java, is a very high-level scripting language developed for this purpose.

Other Internet applications

During the last decades other applications have been developed on the Internet. These applications are not as popular as those discussed above, but they are gaining ground. We discuss a few of them here.

Video-conferencing

Videoconferencing can eliminate the cost of traveling, and save time and energy, by providing communication between two or more groups of participants or a set of individual participants (sometimes called *desktop* videoconferencing).

Each participant or group of participants uses a videoconferencing client program. The video and audio data is sent from the clients to the videoconferencing server program, which then redistributes data to all clients, as shown in Figure 6.32.

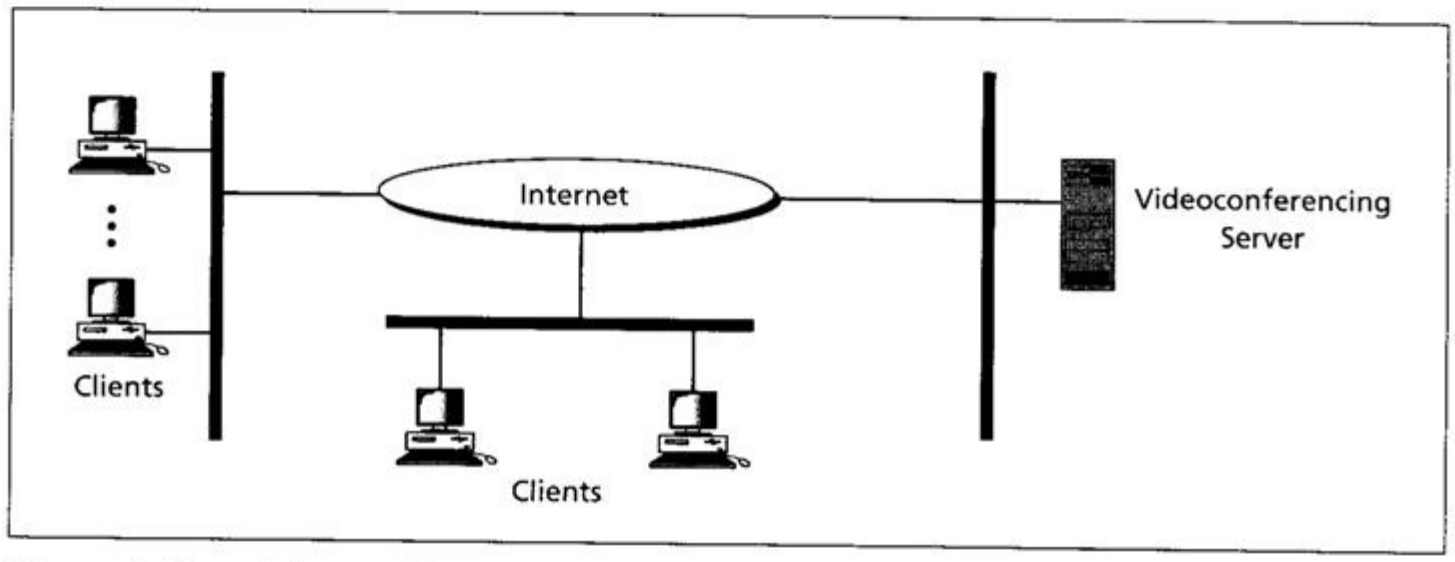


Figure 6.32 Videoconferencing

Group discussion – Listservs

Another popular class of applications is **listservs**, which allow a group of users to discuss a common topic of interest. For example, there could be a group for people interested in seventeenth century Flemish tapestries, or a group of San Francisco Seals fans. One popular use of listservs is the creation of discussion groups for students at colleges and universities, especially for those enrolled in distance learning classes. This is an excellent way for students unable to attend a traditional class to communicate with each other.

Like the others we have discussed, this is also a client-server application. However, there are two server programs running on the server: the subscriber server and the mailer server. Figure 6.33 shows the situation.

Subscribing

The subscriber server accepts membership to a group. A user who wants to be a member of the group sends an e-mail addressed to this server. The e-mail contains a SUBSCRIBE request (command) that is scanned by the subscriber server. The user is then registered if the user has permission to be part of the group. The

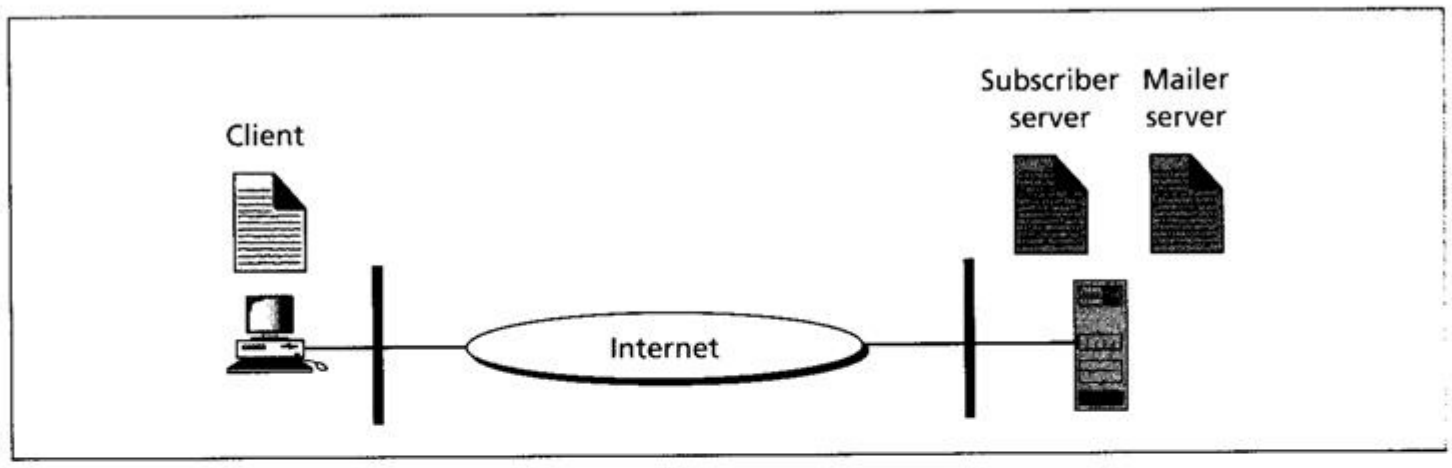


Figure 6.33 Listserver organization

subscriber server informs the user of the registration via return e-mail. A typical subscribe command is shown below:

SUBSCRIBE mailer-server-e-mail-address *user-e-mail-address*

To subscribe, the user sends the request to the subscriber server, not the mailer server.

Sending e-mail

After subscribing, a user can send e-mail to the group. This time, the user sends an e-mail addressed to the mailer server. The mailer server automatically relays the e-mail to every member of the group provided that the e-mail is from a registered user.

Unsubscribing

To cancel a subscription, a user sends an UNSUBSCRIBE command to the subscriber server. A typical unsubscribe command is shown below:

UNSUBSCRIBE **mailer-server-e-mail-address** *user-e-mail-address*

Chat

Another popular class of Internet application is **chat**. This is a real-time application like videoconferencing, in which two or more parties are involved in an exchange of text and optionally audio and video. The two parties can send text to each other, talk to each other (the same way as they might talk on the phone), and even see each other with suitable cameras.

Chatting works the same way as videoconferencing but on a smaller scale. The two parties each use a client program to send text, audio, or video information to a server. The server receives the information and relays the data with a small delay.

multiplexing 156
Multipurpose Internet Mail Extension 168
network layer 158
peer-to-peer communication 153
physical address163
physical topology 147
port number 155
process 153
process-to-process delivery 154
ring 147
router 150
security 146
star topology 147
Stream Control Transmission Protocol 158
terminal network 170
Transmission Control Protocol 157
Uniform Resource Locator 154
User Datagram Protocol 157
wide area network 149
multipoint connection 147
network 146
node 147
performance 146
physical layer 163
point-to-point connection 147
Post Office Protocol (POP) 167
process-to-process communication 153
reliability 146
ring topology 147
routing 158
Simple Mail Transfer Protocol 166
static document 175
TCP/IP protocol suite 151
trailer 164
transport layer 154
user agent 166
videoconferencing 178
World Wide Web 172

6.7 SUMMARY

- A network is a combination of hardware and software that sends data from one location to another. We discussed two types of connection in a network: point-to-point and multipoint. We discussed four types of physical topologies in a network: mesh, star, bus, and ring.
- We mentioned three types of networks: LAN, MAN, and WAN. A local area network (LAN) is usually privately owned and links the devices in a single office, building, or campus. A wide area network (WAN) provides long-distance transmission of data, image, audio, and video information over large geographic areas that may comprise a country, a continent, or even the whole world. A metropolitan area network (MAN) is a network with a size between a LAN and a WAN. When two or more networks are connected, they become an internetwork, or an internet.
- The most notable internet is called the Internet (uppercase letter "I"), a collaboration of hundreds of thousands of interconnected networks. Most end users who want Internet connection use the services of Internet service providers (ISPs). The set, or suite, of protocols that controls the Internet today is referred to as the TCP/IP protocol suite. The suite is made up of five layers.
- The application layer enables the user to access the network. It provides support for services such as electronic mail, remote file access and transfer, browsing the World Wide Web, and so on. The addresses on the application layer are specific to the application programs.
- The transport layer is responsible for process-to-process delivery of the entire message; this means that a logical communication is created between

the transport layers of the client and the server computer. During the life of the TCP/IP protocol suite, three transport layer protocols have been designed: UDP, TCP, and SCTP.

- The network layer is responsible for the source-to-destination delivery of a packet, possibly across multiple networks. The network layer ensures that each packet gets from its point of origin to its final destination. In the TCP/IP protocol suite, the main protocol at the network layer is Internet Protocol (IP).
- The data-link layer delivers a packet from one node to another. The data-link layer is also responsible for error and flow control between "hops". It uses physical or MAC address to identify the nodes.
- The physical layer coordinates the functions required to carry a bit stream over a physical medium. Although the data-link layer is responsible for moving a frame from one node to another, the physical layer is responsible for moving the individual bits that enable the frame to reach the next node.
- Electronic mail (e-mail) is the most popular application on the Internet. The main protocol used for e-mail is called Simple Mail Transfer Protocol (SMTP). Other protocols used for electronic mail are POP and IMAP.
- File Transfer Protocol (FTP) is the standard mechanism for one of the most common tasks on the Internet, copying a file from one computer to another. FTP differs from other client-server applications in that it establishes two connections: one for data transfer and one for exchanging control commands.
- TELNET is a general-purpose client-server program that lets a user access any application program on a remote computer. In other words, it allows the user to log in to a remote computer. After login, a user can use the services available on the remote computer and transfer the results back to the local computer.
- The World Wide Web (WWW), or web, is a repository of information spread all over the world and linked together. To use the WWW, we need three components: a browser, a web server, and a protocol called Hypertext Transfer Protocol (HTTP).
- Documents on the WWW can be grouped into three broad categories: static, dynamic, and active. The category is based on the time at which the contents of the document are determined. A dynamic document is created by a web server whenever a browser requests the document. An active document is a program that is run at the client site.
- Three other applications were mentioned in this chapter. Videoconferencing can provide communication between two or more groups of participants or a set of individual participants. Listservs allow a group of users to discuss a common topic of interest by using two programs running on the server: the subscriber server and the mailer server. Finally, chat is supported by a class of real-time applications similarly to videoconferencing, in which two or more parties are involved in an exchange of text, audio, and video.

6.8 PRACTICE SET

Review questions

1. What is the difference between a point-to-point and a multipoint connection?
2. List common physical network topologies.
3. Define the three types of network.
4. Distinguish between an internet and the Internet.
5. Name the layers of the TCP/IP protocol suite.
6. What is the main function of the application layer in the TCP/IP protocol suite? What type of addresses are used in this layer?
7. What is the main function of the transport layer in the TCP/IP protocol suite? What type of addresses are used in this layer?
8. What is the main function of the network layer in the TCP/IP protocol suite? What type of addresses are used in this layer?

9. What is the main function of the data-link layer in the TCP/IP protocol suite? What type of addresses are used in this layer?
10. What is the main function of the physical layer in the TCP/IP protocol suite? Why are addresses not used in the physical layer?
11. Define node-to-node delivery. In which layer does this type of delivery take place?
12. Define source-to-destination delivery. In which layer does this type of delivery take place?
13. Define process-to-process delivery. In which layer does this type of delivery take place?
14. What is the purpose of SMTP?
15. What is the purpose of FTP?
16. What is the purpose of TELNET?
17. What is the difference between local login and remote login?
18. Compare and contrast the three Internet document types: static, dynamic, and active.

Multiple-choice questions

19. The TCP/IP model has _____ layers.
    a. five
    b. six
    c. seven
    d. any of the above
20. The _____ layer of the TCP/IP protocol suite provides services for end users.
    a. data-link
    b. transport
    c. application
    d. physical
21. The _____ layer of the TCP/IP protocol suite transmits a bit stream over a physical medium.
    a. physical
    b. data-link
    c. network
    d. transport
22. The _____ layer of the TCP/IP protocol suite is responsible for node-to-node delivery of a frame between two adjacent nodes.
    a. transport
    b. network
    c. data-link
    d. session
23. The _____ layer of the TCP/IP protocol suite is responsible for source-to-destination delivery of the entire message.
    a. transport
    b. network
    c. data-link
    d. session
24. What is the domain name in the e-mail address *kayla@nasa.gov*?
    a. kayla
    b. kayla@nasa.gov
    c. nasa.gov
    d. none of the above
25. Which physical topology uses a hub or switch?
    a. bus
    b. ring
    c. star
    d. all of the above
26. IP addresses are currently _____ bits in length.
    a. 4
    b. 8
    c. 32
    d. any of the above
27. The _____ protocol is one of the protocols in the transport layer.
    a. TCP
    b. UDP
    c. SCTP
    d. all of the above
28. _____ is a protocol for file transfer.
    a. FTP
    b. SMTP
    c. TELNET
    d. HTTP
29. _____ is a protocol for e-mail services.
    a. FTP
    b. SMTP
    c. TELNET
    d. HTTP
30. _____ is a protocol for accessing and transferring documents on the WWW.
    a. FTP
    b. SMTP

c. TELNET
d. HTTP

A _____ document has fixed contents.
a. static
b. dynamic
c. active
d. all of the above

Exercises

What is the highest TCP/IP layer responsible for each of the following activities:
a. Sending a frame to the next node.
b. Sending a packet from the source to the destination.
c. Delivery of a long message from the source computer to the destination computer.
d. Logging in to a remote computer.

A small part of a bus LAN with 200 stations is damaged. How many stations are affected by this damage?

A small part of a star LAN with 200 stations is damaged. How many stations are affected by this damage?

A small part of a ring LAN with 200 stations is damaged. How many stations are affected by this damage?

If you have a square room with a computer in each corner, which topology needs least cabling? Justify your answer.
a. A bus LAN.
b. A ring LAN.
c. A star LAN with a hub in the center of the room.

37. If you have a square room with a computer in each corner, which topology is more reliable? Justify your answer.
a. A bus LAN.
b. A ring LAN.
c. A star LAN with a hub in the center of the room.

38. An engineer notices that the data received by computers at the two ends of a bus LAN contains many errors. What do you think is the problem? What can be done to solve the problem?

39. What is the advantage of having three transport protocols in TCP/IP?

40. Change the following IP addresses from dotted-decimal notation to binary notation:
a. 112.32.7.28
b. 129.4.6.8
c. 208.3.54.12
d. 38.34.2.1
e. 255.255.255.255

41. Change the following IP addresses from binary notation to dotted-decimal notation:
a. 01111110 11110001 01100111 01111111
b. 10111111 11011100 11100000 00000101
c. 00011111 11110000 00111111 11011101
d. 10001111 11110101 11000011 00011101
e. 11110111 10010011 11100111 01011101

42. Explain the client-server model on the Internet. In which layer of the TCP/IP protocol suite is the model implemented?

43. Separate the local part and the domain name in the following e-mail addresses:
a. madeline@belle.gov
b. lindsey@jasmine.com
c. wuteh@hunan.int
d. honoris@queen.org

44. Explain the difference between an e-mail address and an IP address. Is there a one-to-one relationship between the two addresses?

45. Explain the difference between FTP and TELNET. When would you use FTP and when would you use TELNET?

46. A user uses a browser to download a game program. What type of document is downloaded?

47. A user uses a browser to download a technical document. What type of document is downloaded?

48. Write a URL that uses HTTP to access a file with the path */user/general* in a computer with the alias name *www.hadb*.

A computer is a system composed of two major components: *hardware* and *software*. Computer hardware is the physical equipment. Software is the collection of programs that allows the hardware to do its job. Computer **software** is divided into two broad categories: the *operating system* and *application programs* (Figure 7.1). Application programs use the computer hardware to solve users' problems. The operating system, on the other hand, controls the access to hardware by users.

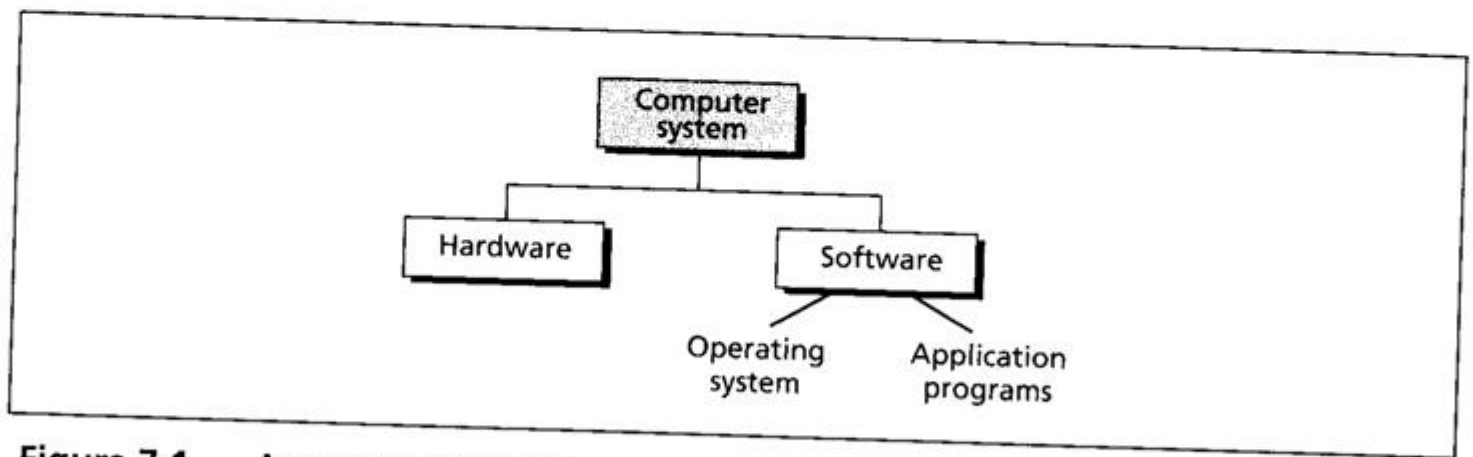


Figure 7.1 A computer system

7.1 INTRODUCTION

An **operating system** is complex, so it is difficult to give a simple universal definition. Instead, here are some common definitions:

- An operating system is an interface between the hardware of a computer and the user (programs or humans).
- An operating system is a program (or a set of programs) that facilitates the execution of other programs.
- An operating system acts as a general manager supervising the activity of each component in the computer system. As a general manager, the operating system checks that hardware and software resources are used efficiently, and when there is a conflict in using a resource, the operating system mediates to solve it.

An operating system is an interface between the hardware of a computer and the user (programs or humans) that facilitates the execution of other programs and the access to hardware and software resources.

Two major design goals of an operating system are:

- Efficient use of hardware.
- Easy of use of resources.

Bootstrap process

The operating system, based on the above definitions, provides supports for other programs. For example, it is responsible for loading other programs into memory

for execution. However, the operating system itself is a program that needs to be loaded into the memory and run. How is this dilemma solved?

The problem can be solved if the operating system is stored (by the manufacturer) in part of memory using ROM technology. The program counter of the CPU (see Chapter 5) can be set to the beginning of this ROM memory. When the computer is turned on, the CPU reads instructions from ROM and executes them. This solution, however, is not very efficient, because a significant part of the memory would need to be composed of ROM and could not therefore be used by other programs. Today's technology needs to allocate just a small part of memory to part of the operating system.

The solution adopted today is a two-stage process. A very small section of memory is made of ROM and holds a small program called the **bootstrap** program. When the computer is turned on, the CPU counter is set to the first instruction of this bootstrap program and executes the instructions in this program. This program is only responsible for loading the operating system itself, or that part of it required to start up the computer, into RAM memory. When loading is done, the program counter in the CPU is set to the first instruction of the operating system in RAM and the operating system is executed. Figure 7.2 illustrates the bootstrap process.

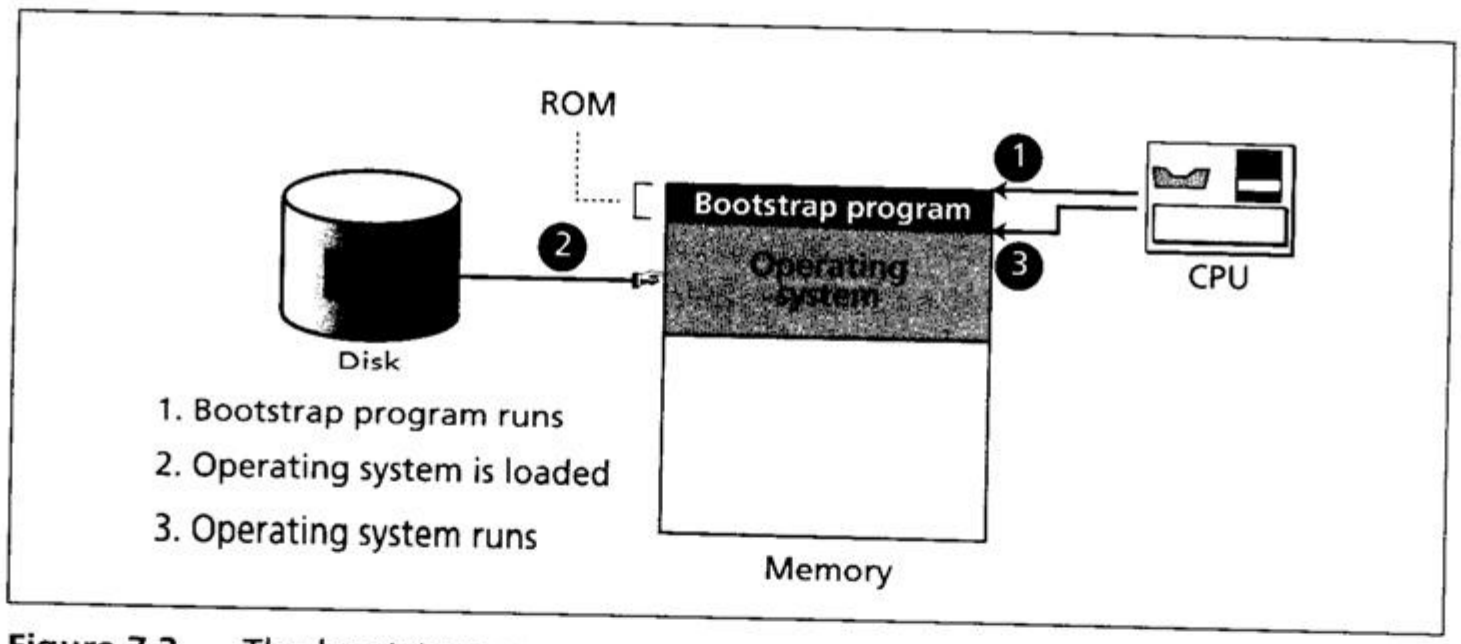


Figure 7.2 The bootstrap process

7.2 EVOLUTION

Operating systems have gone through a long history of evolution, which we summarize next.

Batch systems

Batch operating systems were designed in the 1950s to control mainframe computers. At that time, computers were large machines that used punched cards for input, line printers for output, and tape drives for secondary storage media.

Each program to be executed was called a *job*. A programmer who wished to execute a job sent a request to the operating room along with punched cards for

the program and data. The punched cards were fed into the computer by an operator. If the program was successful, a printout of the result was sent to the programmer—if not, a printout of the error was sent.

Operating systems during this era were very simple: they only ensured that all of the computer's resources were transferred from one job to the next.

Time-sharing systems

To use computer system resources efficiently, *multiprogramming* was introduced. The idea is to hold several jobs in memory at a time, and only assign a resource to a job that needs it on the condition that the resource is available. For example, when one program is using an input/output device, the CPU is free and can be used by another program. We discuss multiprogramming later in this chapter.

Multiprogramming brought the idea of **time sharing:** resources could be shared between different jobs, with each job being allocated a portion of time to use a resource. Because a computer is much faster than a human, time sharing is hidden from the user—each user has the impression that the whole system is serving them exclusively.

Multiprogramming, and eventually time sharing, improved the efficiency of computer systems tremendously. However, they required a more complex operating system. The operating system now had to do **scheduling:** allocating resources to different programs and deciding which program should use which resource, and when. During this era, the relationship between a computer and a user also changed. The user could interact with the system directly without going through an operator. A new term was also coined: **process**. A job is a program to be run, while a process is a program that is in memory and waiting for resources.

Personal systems

When personal computers were introduced, there was a need for an operating system for this new type of computer. During this era, **single-user operating systems** such as DOS (Disk Operating System) were introduced.

Parallel systems

The need for more speed and efficiency led to the design of **parallel systems:** multiple CPUs on the same machine. Each CPU can be used to serve one program or a part of a program, which means that many tasks can be accomplished in parallel instead of serially. The operating systems required for this are more complex than those that support single CPUs.

Distributed systems

Networking and internetworking, as we saw in Chapter 6, have created a new dimension in operating systems. A job that was previously done on one computer can now be shared between computers that may be thousands of miles apart. A program can be run partially on one computer and partially on another if they are connected through an internetwork such as the Internet. In addition, resources can be distributed. A program may need files located in different parts of the world

Distributed systems combine features of the previous generation with new duties such as controlling security.

Real-time systems

A **real-time system** is expected to do a task within specific time constraint. They are used with real-time applications, which monitor, respond to, or control external processes or environments. Examples can be found in traffic control, patient monitoring, or military control systems. The application program can sometimes be an embedded system as a component of a larger system, such as the control system in an automobile.

The requirements for a real-time operating system are often different than those for a general-purpose system. For this reason, we do not discuss them in this chapter.

7.3 COMPONENTS

Today's operating systems are very complex. An operating system needs to manage different resources in a computer system. It resembles an organization with several managers at the top level. Each manager is responsible for managing their department, but also needs to cooperate with others and coordinate activities. A modern operating system has at least four duties: memory manager, process manager, device manager, and file manager. Like many organizations that have a department that is not necessarily under any specific manager, an operating system also has such a component, which is usually called a user interface or a **shell**. The user interface is responsible for communication outside the operating system. Figure 7.3 shows the typical components of an operating system.

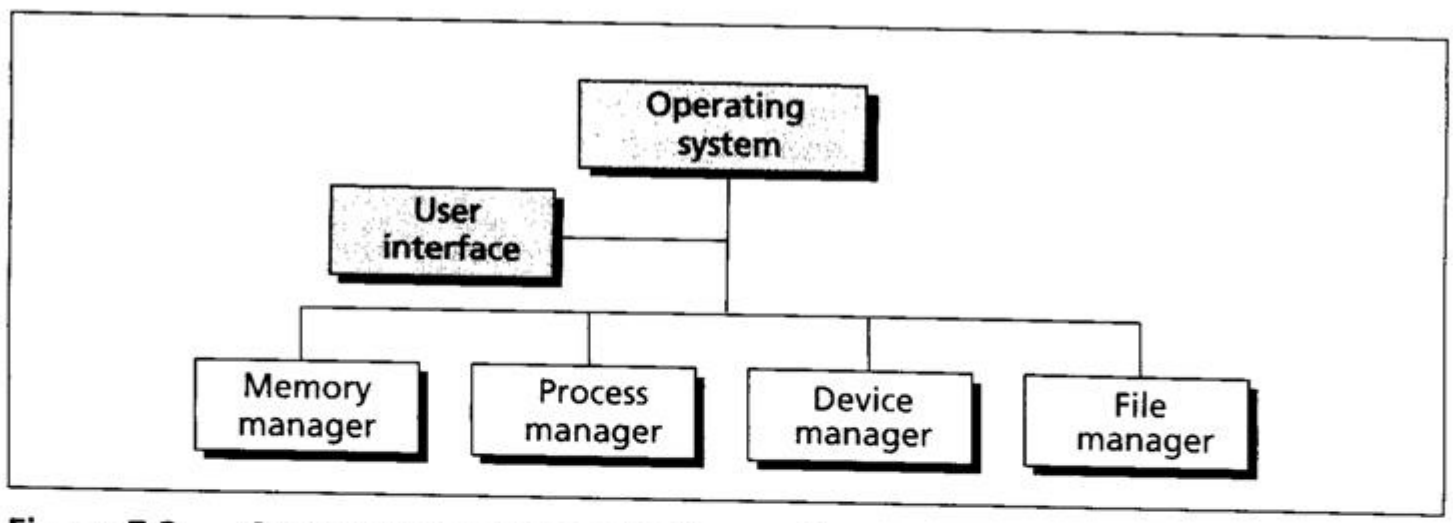


Figure 7.3 Components of an operating system

User interface

Each operating system has a **user interface**, a program that accepts requests from users (processes) and interprets them for the rest of the operating system. A user interface in some operating systems, such as UNIX, is called a **shell**. In others, it is

called a **window** to denote that it is menu-driven and has a **GUI (graphical user interface)** component.

Memory manager

One of the responsibilities of a modern computer system is **memory management**. Although the memory size of computers has increased tremendously in recent years, so has the size of the programs and data to be processed. Memory allocation must be managed to prevent applications from running out of memory. Operating systems can be divided into two broad categories of memory management: *monoprogramming* and *multiprogramming*.

Mono-programming

Monoprogramming belongs to the past, but it is worth mentioning because it helps us to understand multiprogramming. In monoprogramming, most of the memory capacity is dedicated to a single program (we consider the data to be processed by a program as part of the program): only a small part is needed to hold the operating system. In this configuration, the whole program is in memory for execution. When the program finishes running, the program area is occupied by another program (Figure 7.4).

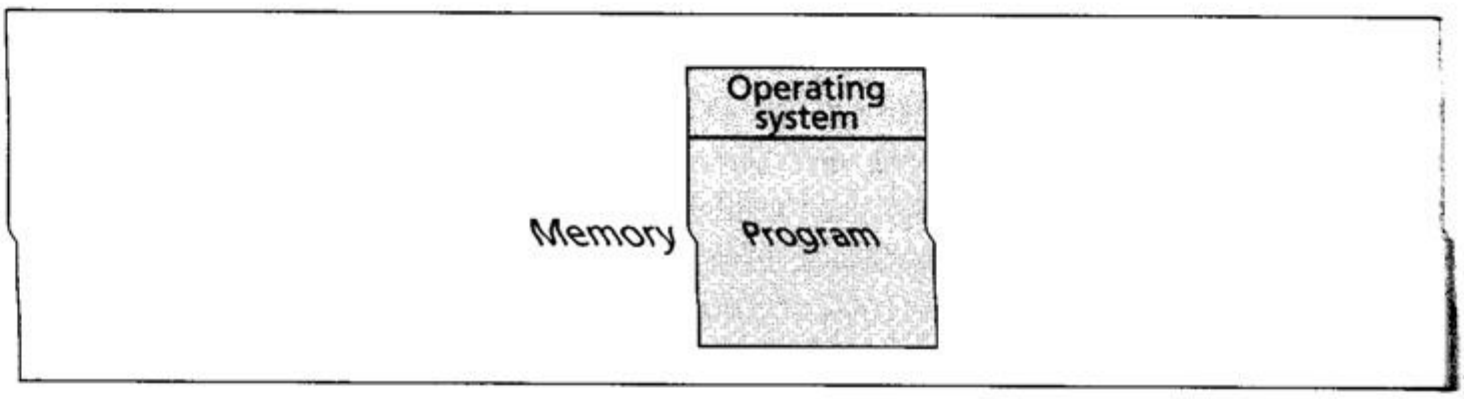


Figure 7.4 Monoprogramming

The job of the memory manager is straightforward here. It loads the program into memory, runs it, and replaces it with the next program. However, there are several problems with this technique:

- The program must fit into memory. If the size of memory is less than the size of the program, the program cannot be run.
- When one program is being run, no other program can be executed. A program, during its execution, often needs to receive data from input devices and needs to send data to output devices. Input/output devices are slow compared with the CPU, so when input/output operations are being carried out, the CPU is idle. It cannot serve another program because this program is not in memory. This is a very inefficient use of memory and CPU time.

Multi-programming

In **multiprogramming**, more than one program is in memory at the same time, and they are executed concurrently, with the CPU switching rapidly between the programs. Figure 7.5 shows memory in a multiprogramming environment.

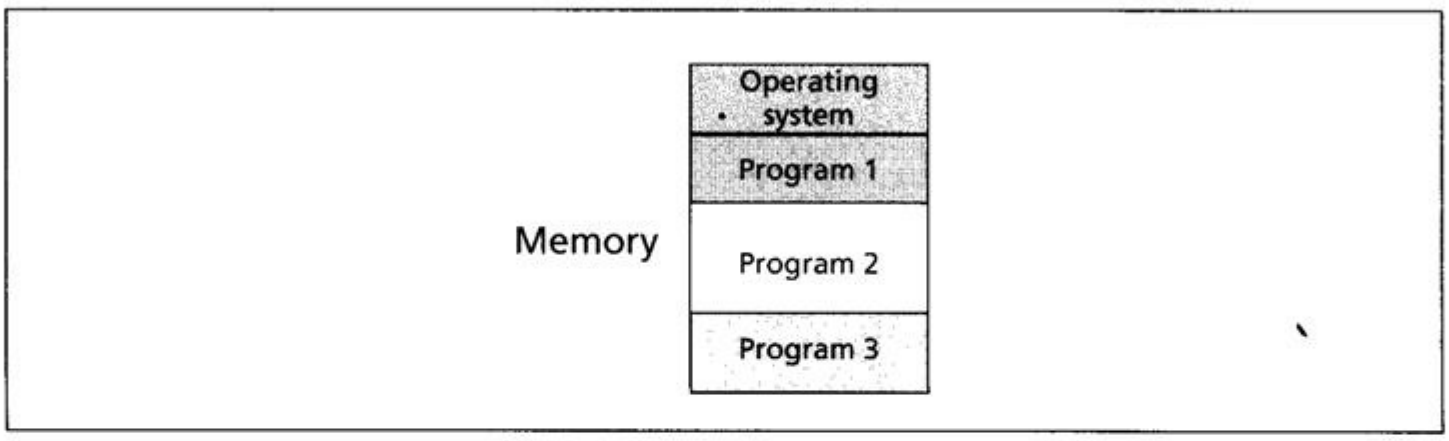


Figure 7.5 Multiprogramming

Since the 1960s, multiprogramming has gone through several improvements that can be seen in the taxonomy in Figure 7.6. We discuss each scheme very briefly in the next few sections. Two techniques belong to the *nonswapping* category, which means that the program remains in memory for the duration of execution. The other two techniques belong to the *swapping* category. This means that, during execution, the program can be swapped between memory and disk one or more times.

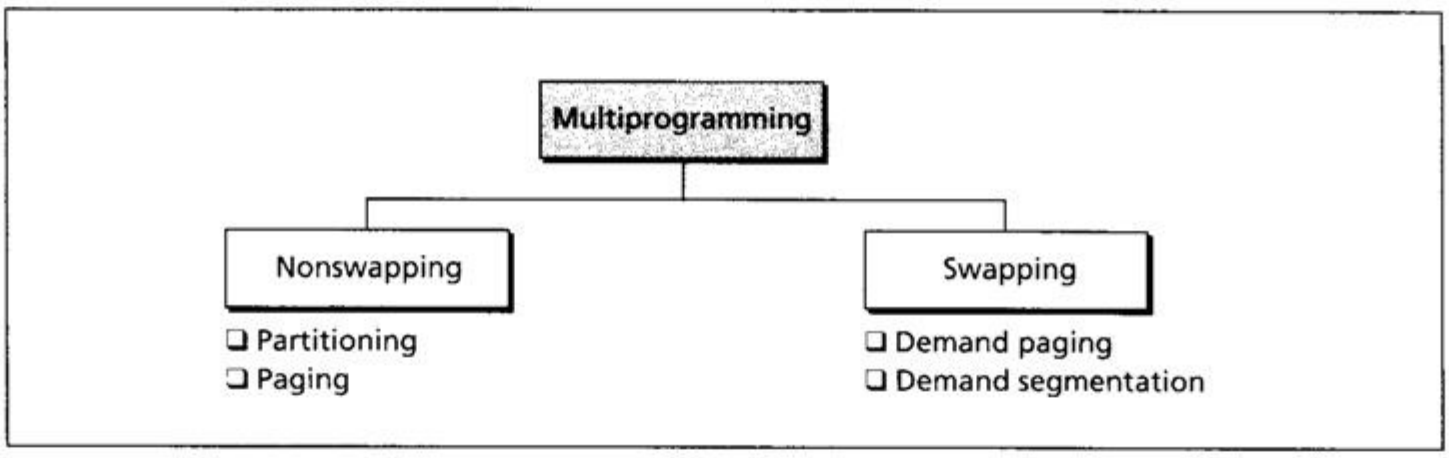


Figure 7.6 Categories of multiprogramming

Partitioning

The first technique used in multiprogramming is called **partitioning**. In this scheme, memory is divided into variable-length sections. Each section or partition holds one program. The CPU switches between programs. It starts with one program, executing some instructions until it either encounters an input/output operation or the time allocated for that program has expired. The CPU then saves the address of the memory location where the last instruction was executed and moves to the next program. The same procedure is repeated with the second program. After all the programs have been served, the CPU moves back to the first program. Priority levels can also be used to control the amount of CPU time allocated to each program (Figure 7.7).

With this technique, each program is entirely in memory and occupying contiguous locations. Partitioning improves the efficiency of the CPU, but there are still some issues:

- The size of the partitions has to be determined beforehand by the memory manager. If partition sizes are small, some programs cannot be loaded into

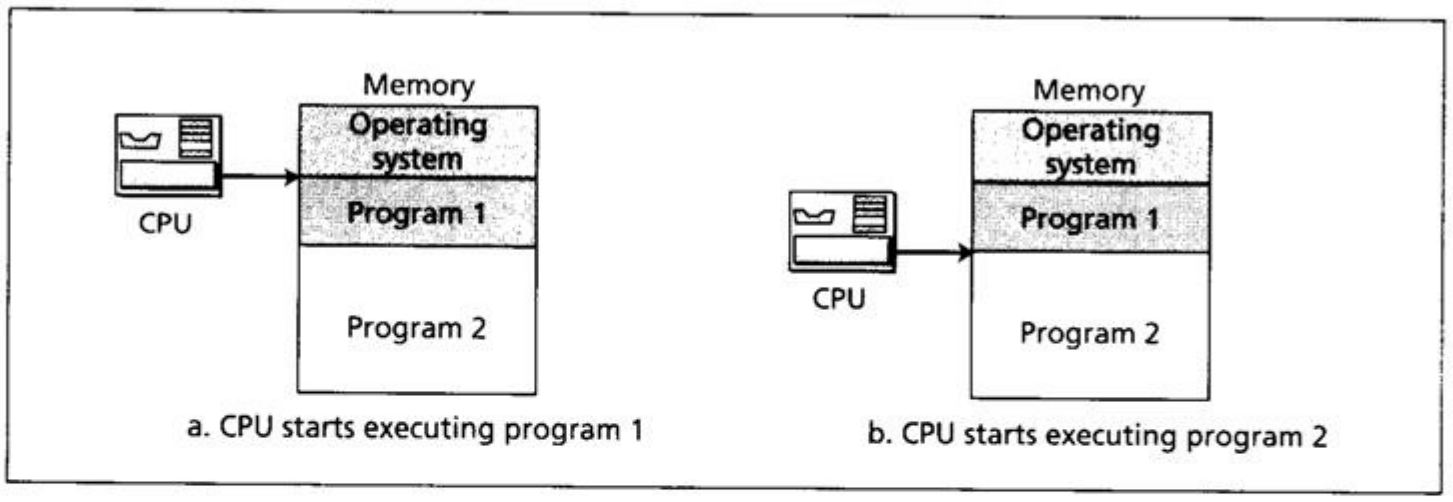


Figure 7.7 Partitioning

memory. If partition sizes are large, there might be some "holes" (unused locations) in memory.

- Even if partitioning is perfect when the computer is started, there may be some holes after completed programs are replaced by new ones.
- When there are many holes, the memory manager can compact the partitions to remove the holes and create new partitions, but this creates extra overhead on the system.

Paging

Paging improves the efficiency of partitioning. In paging, memory is divided into equally sized sections called **frames**. Programs are also divided, into equally sized sections called **pages**. The size of a page and a frame is usually the same and equal to the size of the block used by the system to retrieve information from a storage device (Figure 7.8).

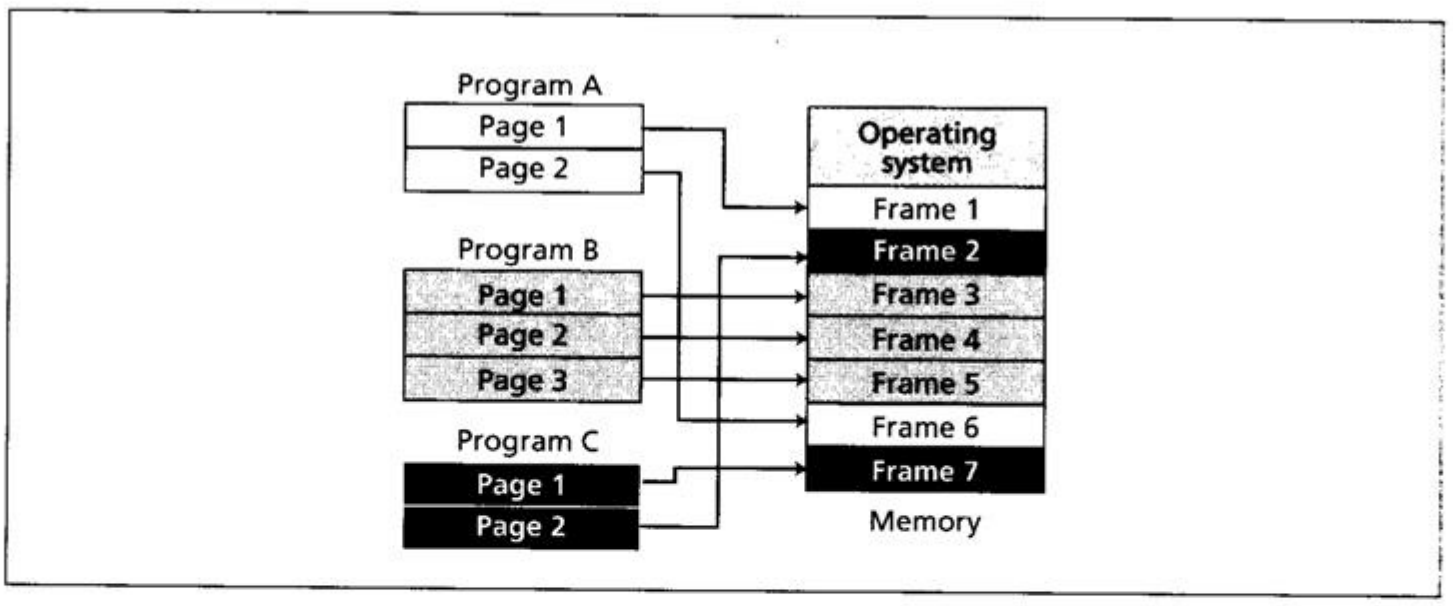


Figure 7.8 Paging

A page is loaded into a frame in memory. If a program has three pages, it occupies three frames in memory. With this technique, the program does not have to be contiguous in memory: two consecutive pages can occupy noncontiguous frames in memory. The advantage of paging over partitioning is that two programs, each using three noncontiguous frames, can be replaced by one program that needs six frames.

There is no need for the new program to wait until six contiguous frames are free before being loaded into memory.

Paging improves efficiency to some extent, but the whole program still needs to be in memory before being executed. This means that a program that needs six frames, for example, cannot be loaded into memory if there are currently only four unoccupied frames.

Demand paging

Paging does not require that the program be in contiguous memory locations, but it does require that the entire program be in memory for execution. Demand paging has removed this last restriction. In **demand paging** the program is divided into pages, but the pages can be loaded into memory one by one, executed, and replaced by another page. In other words, memory can hold pages from multiple programs at the same time. In addition, consecutive pages from the same program do not have to be loaded into the same frame—a page can be loaded into any free frame. An example of demand paging is shown in Exercise 7.9. Two pages from program A, one page from program B, and one page from program C are in the memory.

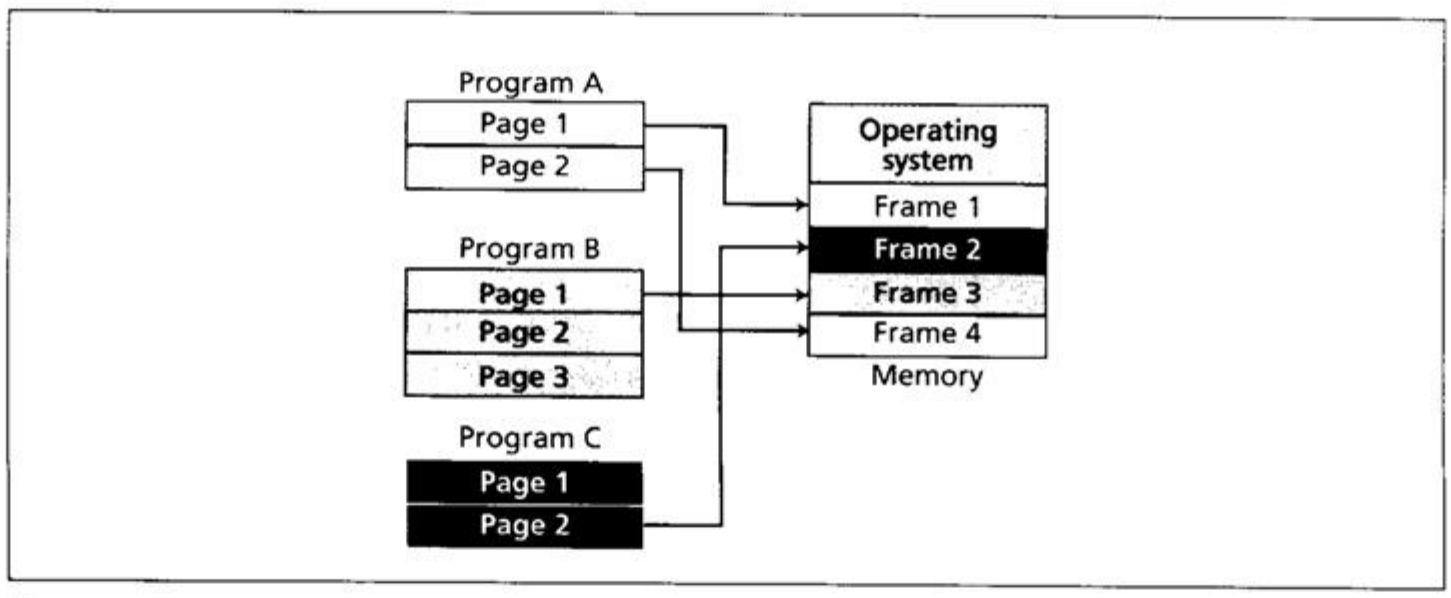


Figure 7.9 Demand paging

Demand segmentation

A technique similar to paging is *segmentation*. In paging, a program is divided into equally sized pages, which is not the way a programmer thinks—a programmer thinks in terms of modules. As we will see in later chapters, a program is usually made up of a main program and subprograms. In **demand segmentation**, the program is divided into segments that match the programmer's view. These are loaded into memory, executed, and replaced by another module from the same or a different program. An example of demand segmentation is shown in Figure 7.10. Since segments in memory are of equal size, part of a segment may remain empty.

Demand paging and segmentation

Demand paging and segmentation can be combined to further improve the efficiency of the system. A segment may be too large to fit any available free space in memory. Memory can be divided into frames, and a module can be divided into

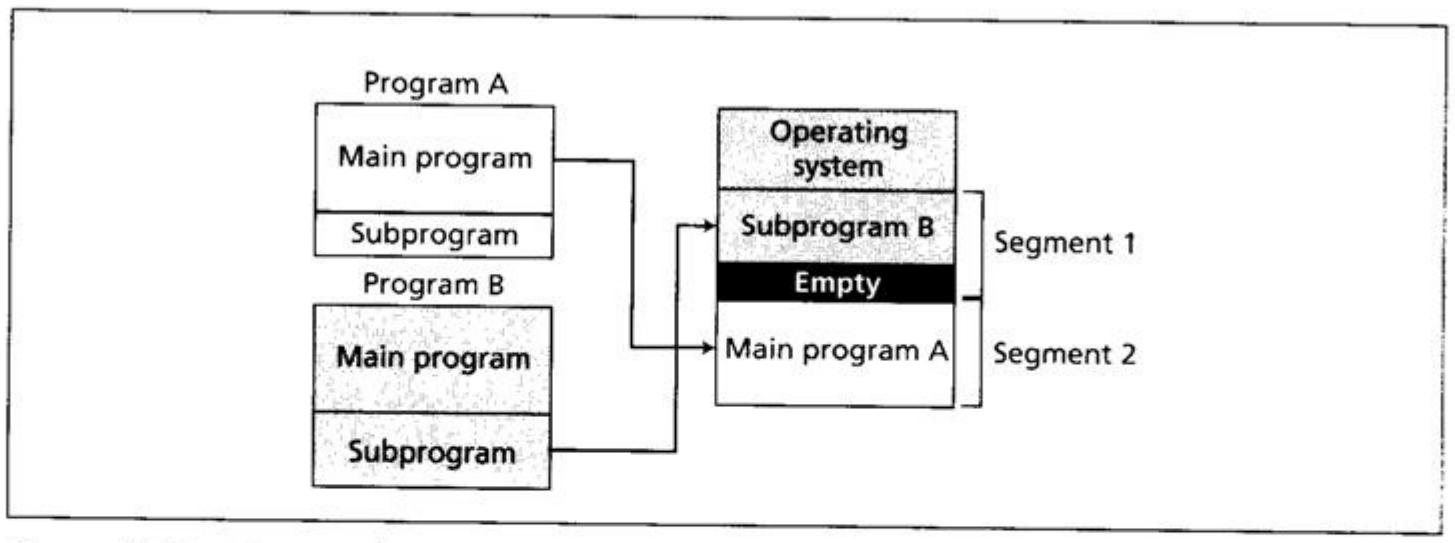


Figure 7.10 Demand segmentation

pages. The pages of a module can then be loaded into memory one by one and executed.

Virtual memory

Demand paging and demand segmentation mean that, when a program is being executed, part of the program is in memory and part is on disk. This means that, for example, a memory size of 10 MB can execute 10 programs, each of size 3 MB, for a total of 30 MB. At any moment, 10 MB of the 10 programs are in memory and 20 MB are on disk. There is therefore an actual memory size of 10 MB, but a **virtual memory** size of 30 MB. Figure 7.11 shows the concept. Virtual memory, which implies demand paging, demand segmentation, or both, is used in almost all operating systems today.

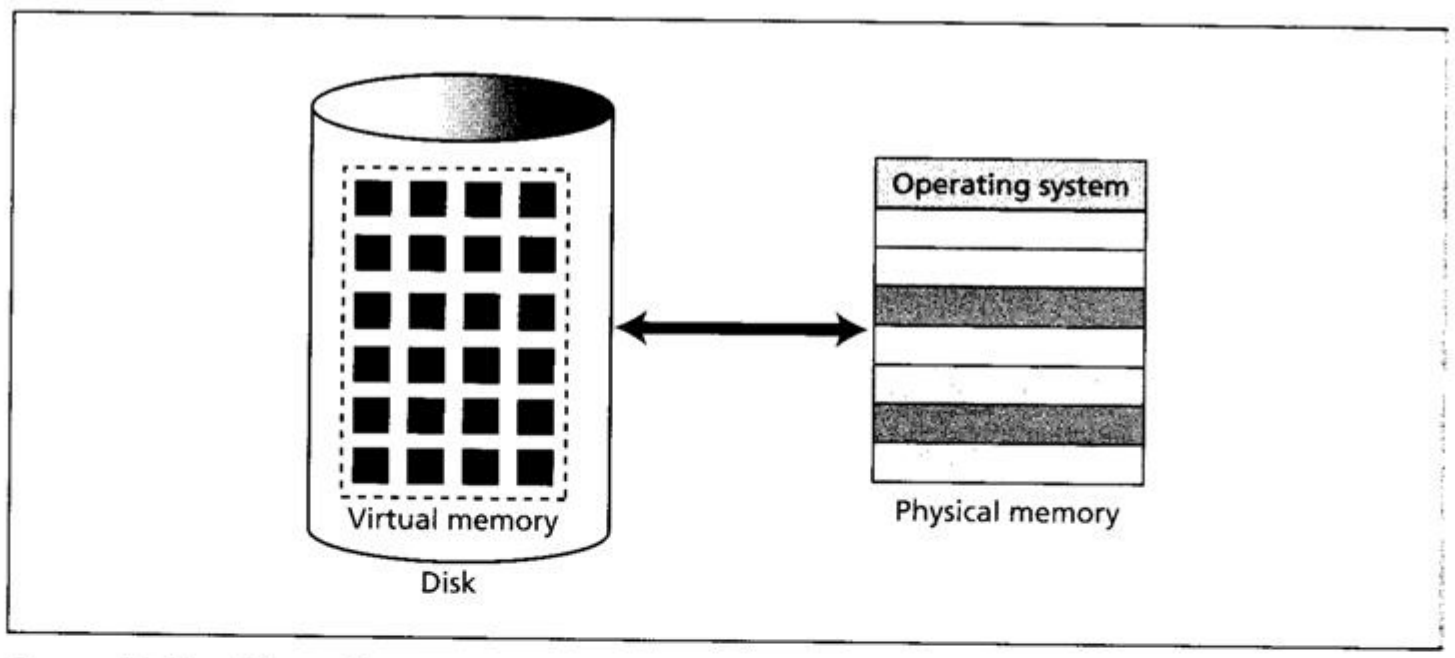


Figure 7.11 Virtual memory

Process manager

A second function of an operating system is process management, but before discussing this concept, we need to define some terms.

gram, job, d process

Modern operating systems use three terms that refer to a set of instructions: *program, job*, and *process*. Although the terminology is vague and varies from one operating system to another, we can define these terms informally.

Program

A **program** is a nonactive set of instructions stored on disk (or tape). It may or may not become a job.

Job

A program becomes a **job** from the moment it is selected for execution until it has finished running and becomes a program again. During this time a job may or may not be executed. It may be located on disk waiting to be loaded to memory, or it may be loaded into memory and waiting for execution by the CPU. It may be on disk or in memory waiting for an input/output event, or it may be in memory while being executed by the CPU. The program is a job in all of these situations. When a job has finished executing (either normally or abnormally), it becomes a program and once again resides on the disk. The operating system no longer governs the program. Note that every job is a program, but not every program is a job.

Process

A **process** is a program in execution. It is a program that has started but has not finished. In other words, a process is a job that is being run in memory. It has been selected among other waiting jobs and loaded into memory. A process may be executing or it may be waiting for CPU time. As long as the job is in memory, it is a process. Note that every process is a job, but not every job is a process.

te grams

The relationship between a program, a job, and a process becomes clearer if we consider how a program becomes a job and how a job becomes a process. This can be illustrated with a **state diagram** that shows the different states of each of these entities. Figure 7.12 is a state diagram using boundaries between a program, a job, and a process.

A program becomes a job when selected by the operating system and brought to the **hold state**. It remains in this state until it can be loaded into memory. When there is memory space available to load the program totally or partially, the job moves to the **ready state**. It now becomes a process. It remains in memory and in this state until the CPU can execute it, moving to the **running state** at this time. When in the running state, one of three things can happen:

- The process executes until it needs I/O resources
- The process exhausts its allocated time slot
- The process terminates

In the first case, the process goes into the **waiting state** and waits until I/O is complete. In the second case, it goes directly to the ready state. In the third case, it goes into the **terminated state** and is no longer a process. A process can move between the running, waiting, and ready states many times before it goes to the terminated state. Note that the diagram can be much more complex if the system uses virtual memory and swaps programs in and out of main memory.

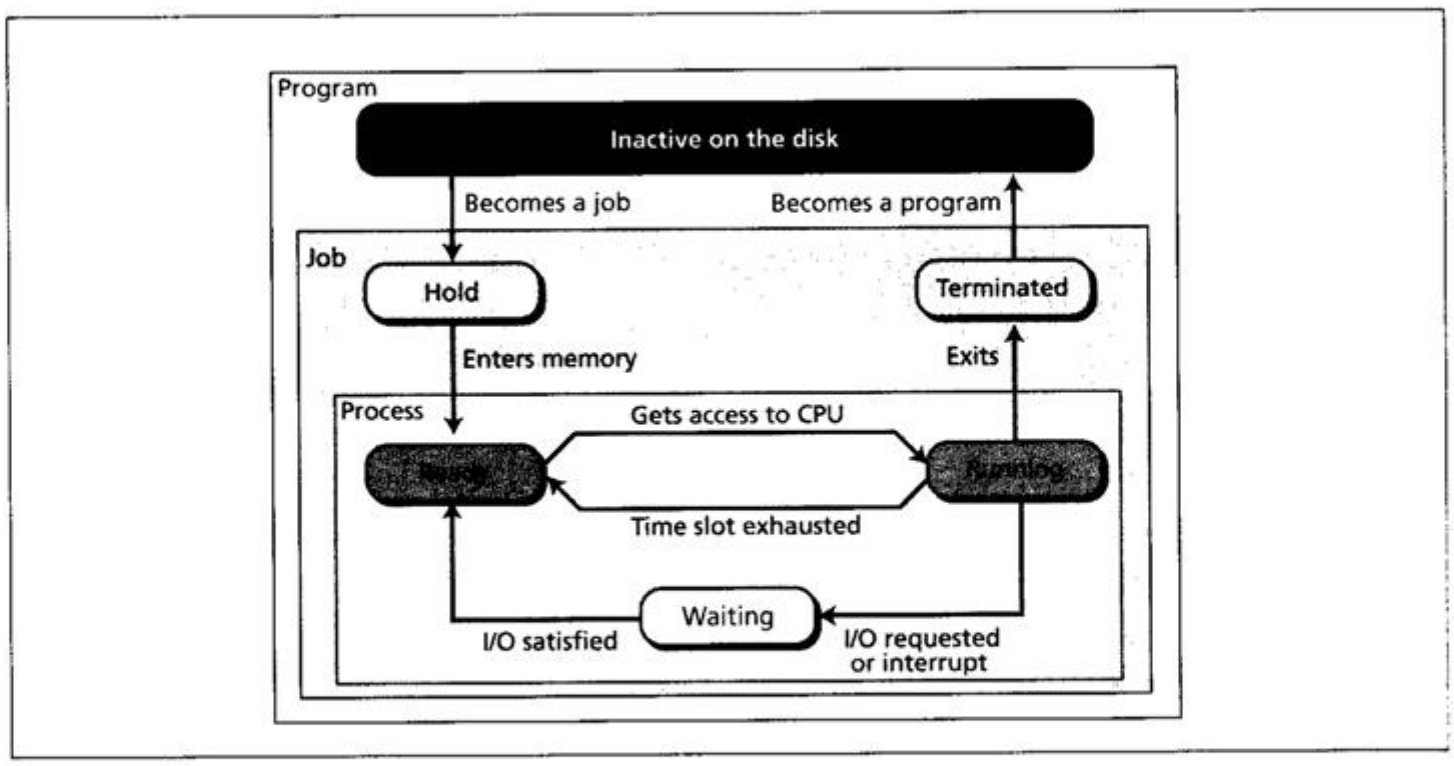


Figure 7.12 State diagram with boundaries between program, job, and process

Schedulers

To move a job or process from one state to another, the process manager uses two **schedulers**: the job scheduler and the process scheduler.

Job scheduler

The **job scheduler** moves a job from the hold state to the ready state or from the running state to the terminated state. In other words, a job scheduler is responsible for creating a process from a job and terminating a process. Figure 7.13 shows the job scheduler.

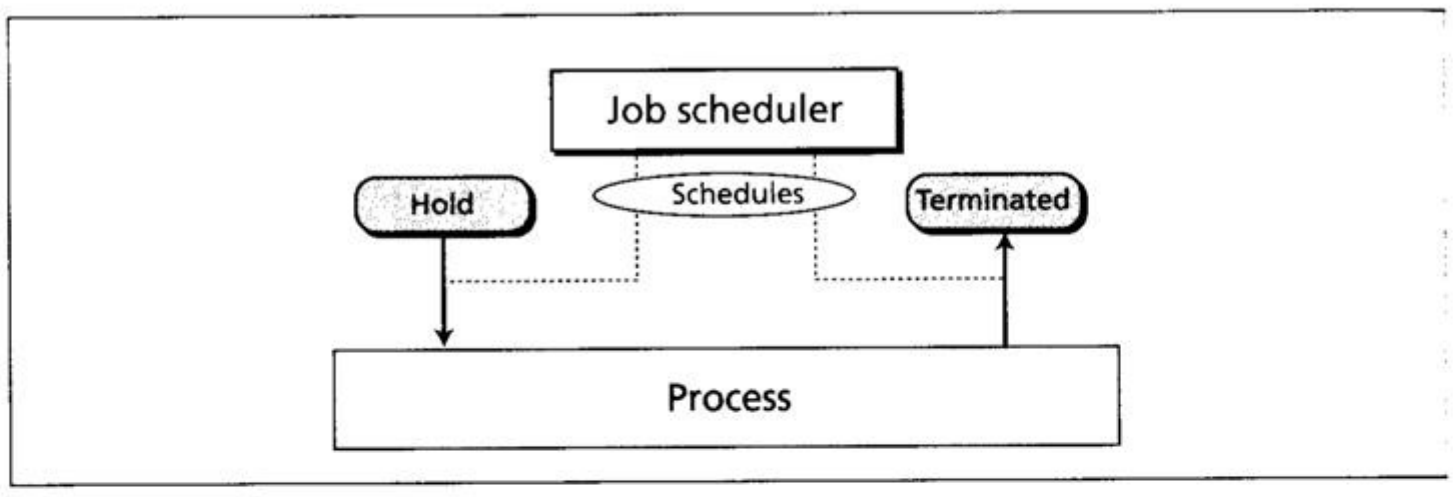


Figure 7.13 Job scheduler

Process scheduler

The **process scheduler** moves a process from one state to another. It moves a process from the running state to the waiting state when the process is waiting for some event to happen. It moves the process from the waiting state to the ready state when the event has occurred. It moves a process from the running state to the ready state if the process' time allotment has expired. When the CPU is ready to run the process, the process scheduler moves the process from the ready state to the running state. Figure 7.14 shows the process scheduler.

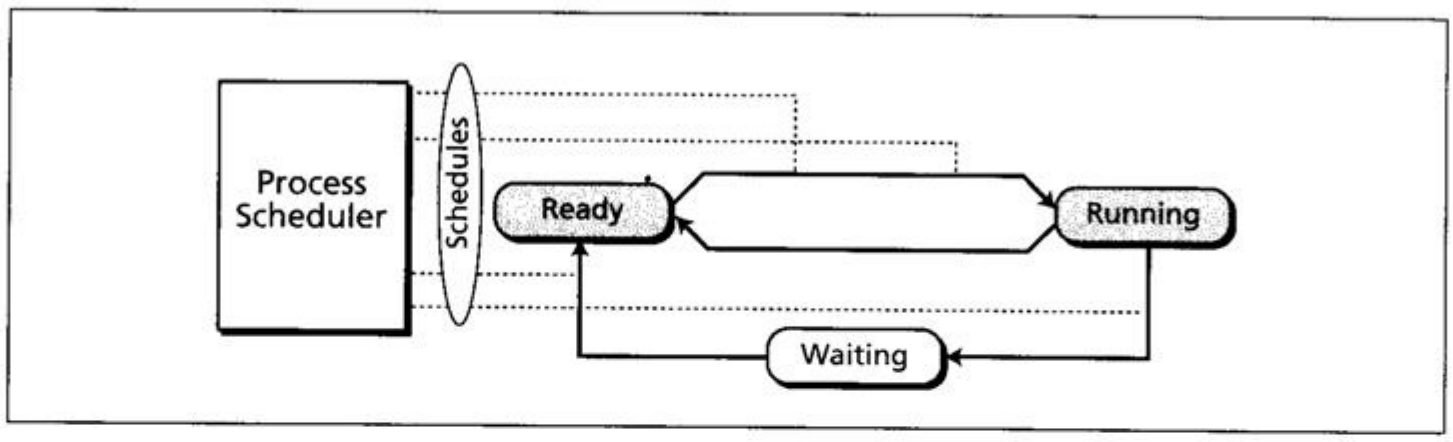


Figure 7.14 Process scheduler

Other schedulers

Some operating systems use other types of schedulers to make switching between processes more efficient.

Our state diagram shows one job or process moving from one state to another. In reality, there are many jobs and many processes competing with each other for computer resources. For example, when some jobs are in memory, others must wait until space is available. Or when a process is running using the CPU, others must wait until the CPU is free. To handle multiple processes and jobs, the process manager uses **queues** (waiting lists). A *job control block* or *process control block* is associated with each job or process. This is a block of memory that stores information about that job or process. The process manager stores the job or process control block in the queues instead of the job or process itself. The job or process itself remains in memory or disk, as it is too big to be duplicated in a queue: the job control block or process control block is the representative of the waiting job or process.

An operating system can have several queues. For example, Figure 7.15 shows the circulation of jobs and processes through three queues: the *job queue*, the *ready queue*, and the *I/O queue*. The job queue holds the jobs that are waiting for memory. The ready queue holds the processes that are in memory, ready to be run and waiting for the CPU. The I/O queue holds the processes that are waiting for an I/O device (there can be several I/O queues, one for each input/output device, but we show only one for simplicity).

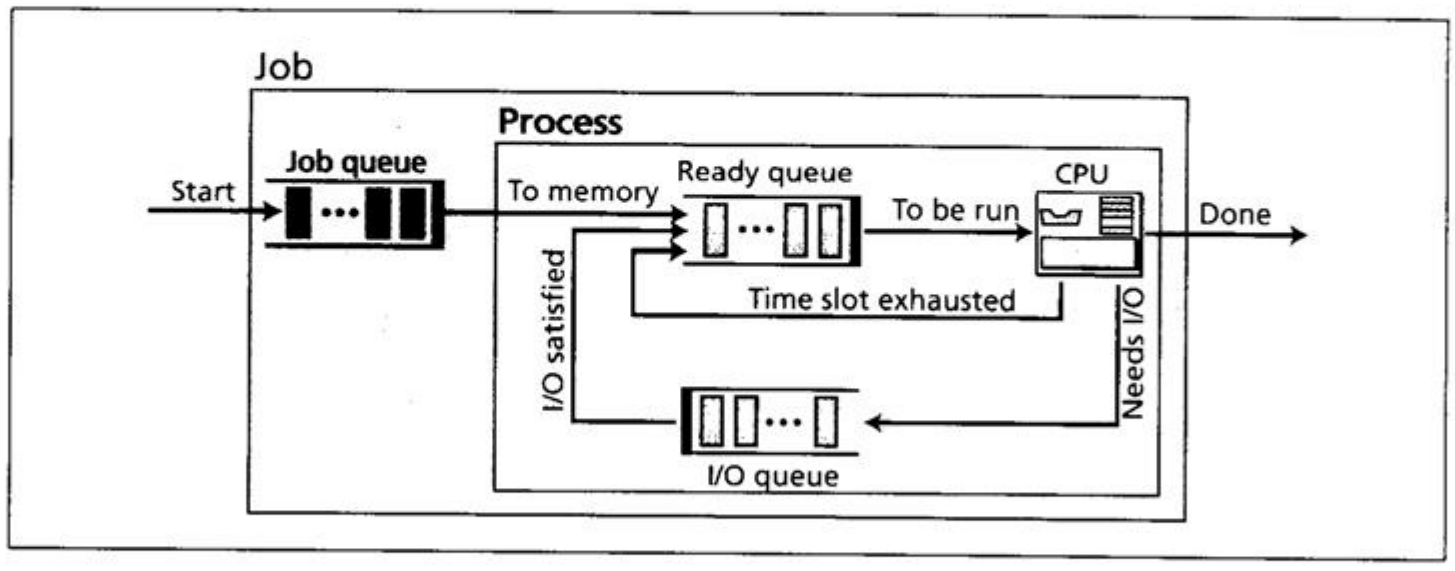


Figure 7.15 Queues for process management

The process manager can have different policies for selecting the next job or process from a queue: it could be first in, first out (FIFO), shortest length first, highest priority first, and so on.

Process synchronization

The whole idea behind process management is to synchronize different processes with different resources. Whenever resources can be used by more than one user (or process, in this case), we can have two problematic situations: *deadlock* and *starvation*. A brief discussion of these two situations follows.

Deadlock

Instead of a formal definition of **deadlock**, we give an example. Assume that there are two processes, A and B. Process A is holding a file File1 (that is, File1 is assigned to A) and cannot release it until it acquires another file, File2 (that is, A has requested File2). Process B is holding File2 (that is, File2 is assigned to B) and cannot release it until it has File1 (that is, B has requested File1). Files in most systems are not sharable—when in use by one process, a file cannot be used by another process. If there is no provision in this situation to force a process to release a file, deadlock is created (Figure 7.16).

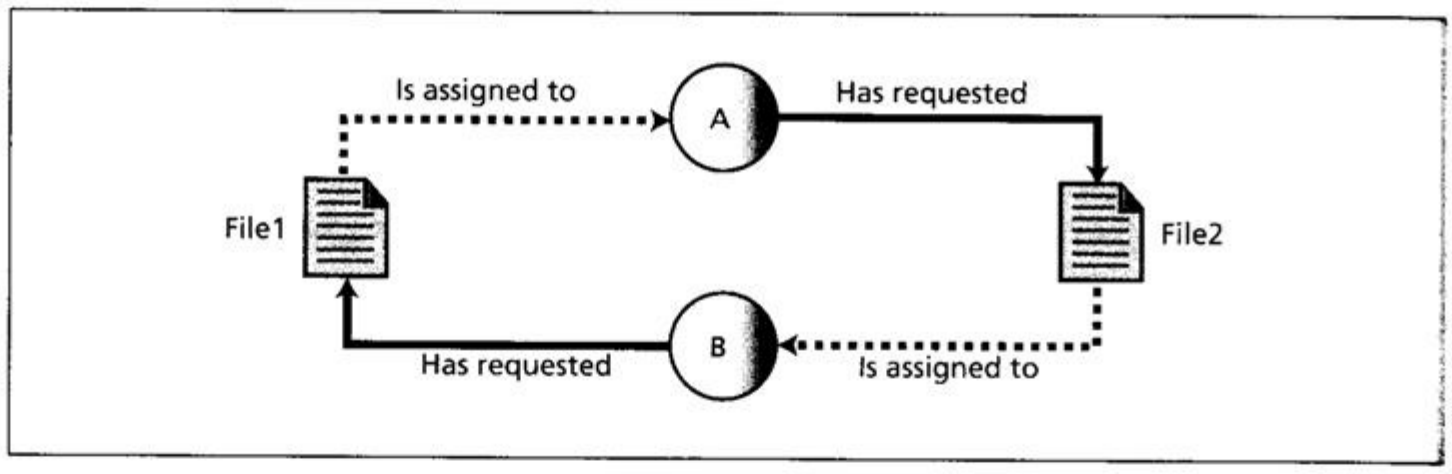


Figure 7.16 Deadlock

As an analogy, Figure 7.17 shows deadlock on a narrow bridge. The situation is similar because the resource (part of the bridge) is held by a vehicle that does not release it until it gets the other part of the bridge, which is held by the other vehicle, and vice versa.

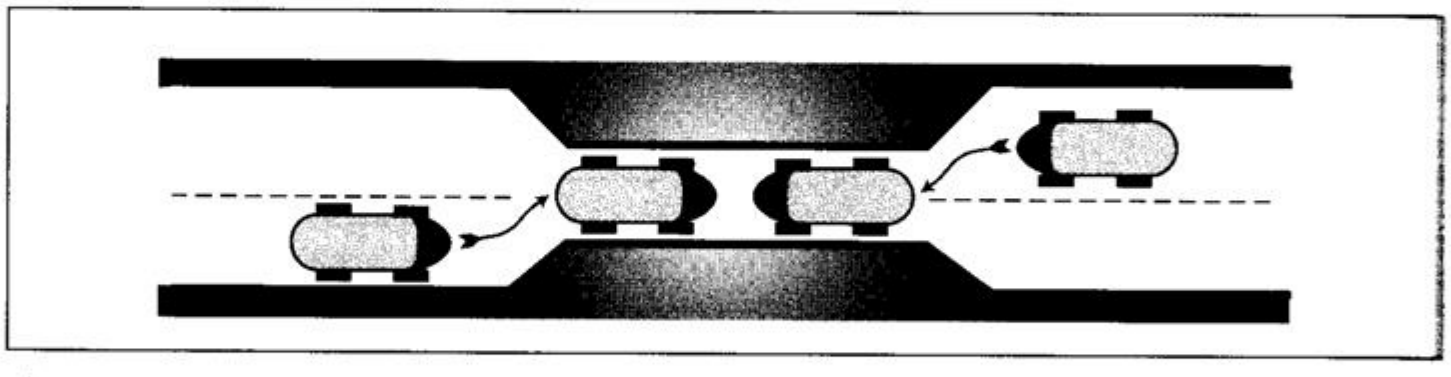


Figure 7.17 Deadlock on a bridge

Deadlock occurs if the operating system allows a process to start running without first checking to see if the required resources are ready, and allows a process to hold a resource as long as it wants. There should be some provision in the system to prevent deadlock. One solution is not to allow a process to start running until the required resources are free, but we will see later that this creates another problem. The second solution is to limit the time a process can hold a resource.

Deadlock occurs when the operating system does not put resource restrictions on processes.

Deadlock does not always occur. There are four necessary conditions for deadlock as shown below:

- **Mutual exclusion**. Only one process can hold a resource.
- **Resource holding**. A process holds a resource even though it cannot use it until other resources are available.
- **No preemption**. The operating system cannot temporarily reallocate a resource.
- **Circular waiting**. All processes and resources involved form a loop, as in Figure 7.16.

All four conditions are required for deadlock to occur. However, these conditions are only necessary preconditions, and are not sufficient to cause deadlock of themselves—they must be present for deadlock, but they might not be enough to cause it. If one of these conditions is missing, deadlock cannot occur. This gives us a method for preventing or avoiding deadlock: do not allow one of these conditions to happen.

Starvation

Starvation is the opposite of deadlock. It can happen when the operating system puts too many resource restrictions on a process. For example, imagine an operating system that specifies that a process must have possession of its required resources before it can be run.

In Figure 7.18, imagine that process A needs two files, File1 and File2. File1 is being used by process B and File2 is being used by process E. Process B terminates first and releases File1. Process A cannot be started, because File2 is still not available. At this moment, process C, which needs only File1, is allowed to run. Now process E terminates and releases File2, but process A still cannot run because File1 is unavailable.

A classic starvation problem was introduced by Edsger Dijkstra. Five philosophers are sitting at a round table (Figure 7.19). Each philosopher needs two chopsticks to eat a bowl of rice. However, one or both chopsticks could be used by a neighbor. A philosopher could starve if two chopsticks are not available at the same time.

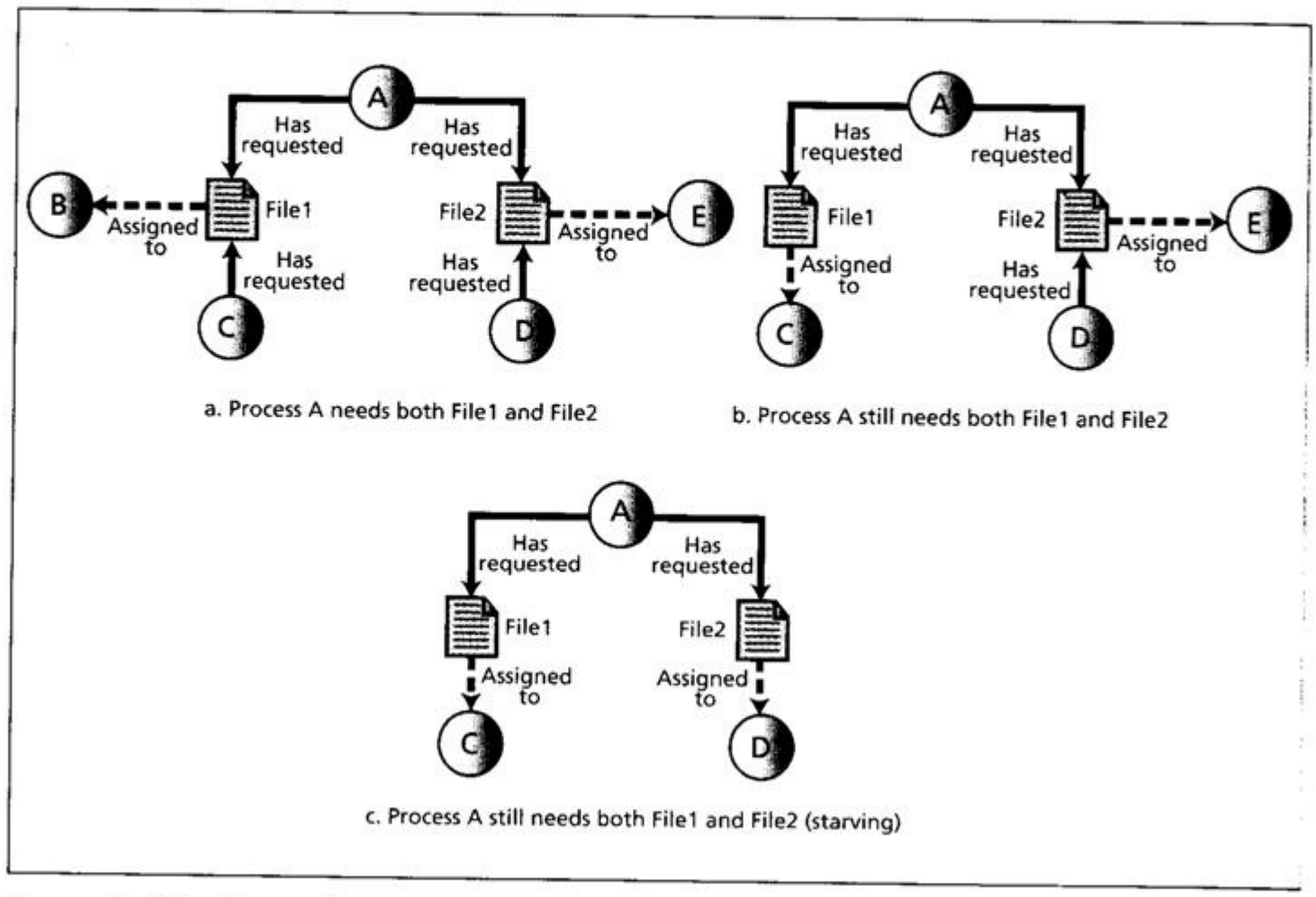


Figure 7.18 Starvation

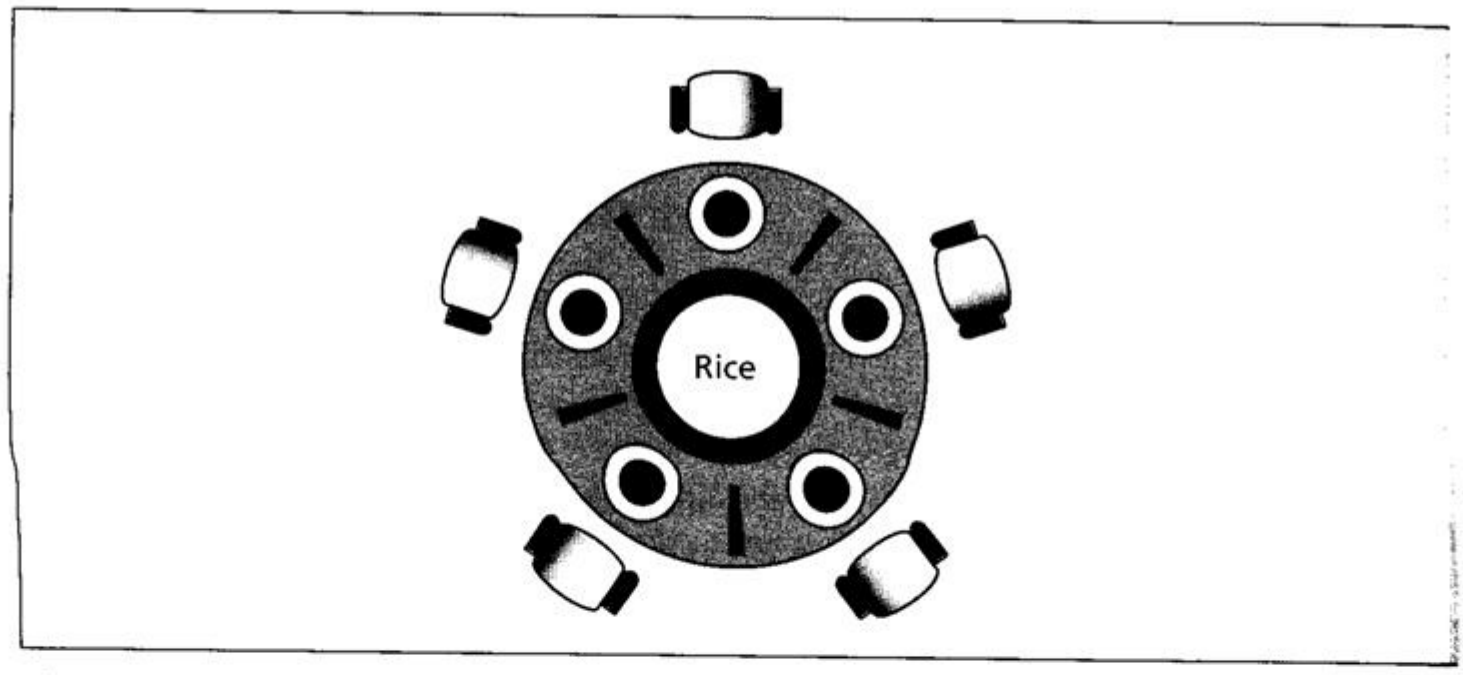


Figure 7.19 The dining philosophers problem

Device manager

The device manager, or input/output manager, is responsible for access to input/output devices. There are limitations on the number and speed of input/output devices in a computer system. Because these devices are slower in speed compared with the CPU and memory, when a process accesses an input/output device, the device is not available to other processes for a period of time. The device manager is responsible for the efficient use of input/output devices.

A detailed discussion of device managers requires advanced knowledge of operating system principles and is beyond the scope of this book. However, we can briefly list the responsibilities of a device manager:

- The device manager monitors every input/output device constantly to ensure that the device is functioning properly. The manager also needs to know when a device has finished serving one process and is ready to serve the next process in the queue.
- The device manager maintains a queue for each input/output device or one or more queues for similar input/output devices. For example, if there are two fast printers in the system, the manager can have one queue for each or one queue for both.
- The device manager controls the different policies for accessing input/output devices. For example, it may use FIFO for one device and shortest length first for another.

File manager

Operating systems today use a file manager to control access to files. A detailed discussion of the file manager also requires advanced knowledge of operating system principles and file access concepts that are beyond the scope of this book. We discuss some issues related to file access in Chapter 13, but this is not adequate to understand the actual operation of a file manager. Here is a brief list of the responsibilities of a file manager:

- The file manager controls access to files. Access is allowed only by permitted applications and/or users, and the type of access can vary. For example, a process (or a user that calls a process) may be allowed to read from a file but is not allowed to write to it (that is, change it). Another process may be allowed to execute a file and a process, but not allowed to read its contents, and so on.
- The file manager supervises the creation, deletion, and modification of files.
- The file manager can control the naming of files.
- The file manager supervises the storage of files: how they are stored, where they are stored, and so on.
- The file manager is responsible for archiving and backups.

7.4 A SURVEY OF OPERATING SYSTEMS

In this section we introduce some popular operating systems and encourage you to study them further. We have chosen three operating systems that are familiar to most computer users: UNIX, Linux, and Windows.

UNIX

UNIX was originally developed in 1969 by Thomson and Ritchie of the Computer Science Research Group at Bell Laboratories. UNIX has gone through many versions since then. It has been a popular operating system among computer programmers and computer scientists. It is a very powerful operating system with three outstanding features. First, UNIX is a portable operating system that can be moved from one platform to another without many changes. The reason is that it is

written mostly in the C language (instead of a machine language specific to a particular computer system). Second, UNIX has a powerful set of utilities (commands) that can be combined (in an executable file called a *script*) to solve many problems that require programming in other operating systems. Third, it is device-independent, because it includes device drivers in the operating system itself, which means that it can be easily configured to run any device.

UNIX is a multiuser, multiprocessing, portable operating system designed to facilitate programming, text processing, communication, and many other tasks that are expected from an operating system. It contains hundreds of simple, single-purpose functions that can be combined to do virtually every processing task imaginable. Its flexibility is demonstrated by the fact that it is used in three different computing environments: stand-alone personal environments, time-sharing systems, and client–server systems.

UNIX is a multiuser, multiprocessing, portable operating system. It is designed to facilitate programming, text processing, and communication.

UNIX structure

UNIX consists of four major components: the *kernel*, the *shell*, a standard set of *utilities*, and *application programs*. These components are shown in Figure 7.20.

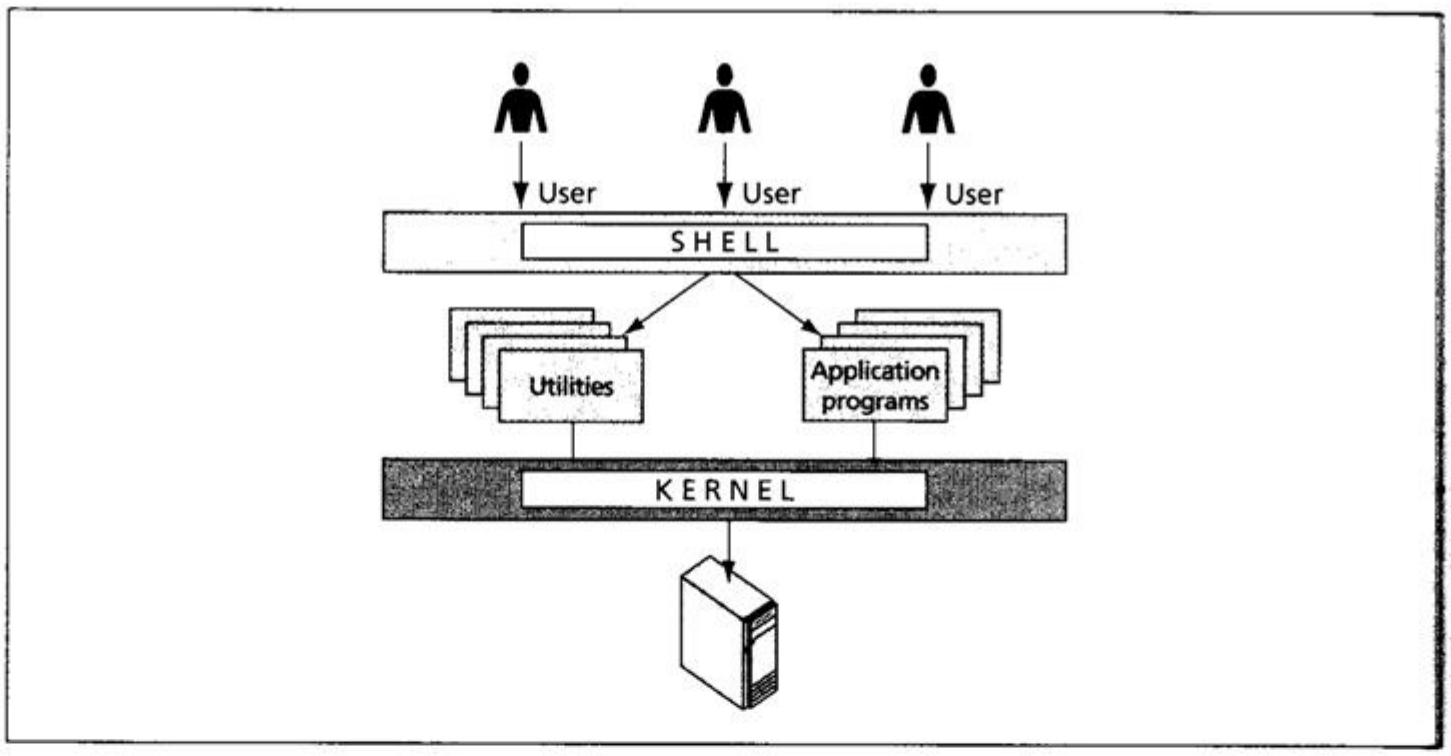


Figure 7.20 Components of the UNIX operating system

The kernel

The **kernel** is the heart of the UNIX system. It contains the most basic parts of the operating system: memory management, process management, device management, and file management. All other components of the system call on the kernel to perform these services for them.

The shell

The shell is the part of UNIX that is most visible to the user. It receives and interprets the commands entered by the user. In many respects, this makes it the most

important component of the UNIX structure. It is certainly the part that users get to know best. To do anything in the system, we must give the shell a command. If the command requires a utility, the shell requests that the kernel execute the utility. If the command requires an application program, the shell requests the kernel to run it. Some operating systems, such as UNIX, have several different shells.

Utilities

There are literally hundreds of UNIX utilities. A **utility** is a standard UNIX program that provides a support process for users. Three common utilities are text editors, search programs, and sort programs.

Many of the system utilities are actually sophisticated applications. For example, the UNIX e-mail system is considered a utility, as are the three common text editors, **vi**, **emacs**, and **pico**. All four of these utilities are large systems in themselves. Other utilities are shorter, simpler functions. For example, the list (**ls**) utility displays the files in a disk directory.

Applications

Applications in UNIX are programs that are not a standard part of the operating system distribution. Written by systems administrators, professional programmers, or users, they provide extended capabilities to the system. In fact, many of the standard utilities started out as applications years ago and proved so useful that they are now part of the system.

Linux

In 1991, Linus Torvalds, a Finnish student at the University of Helsinki at the time, developed a new operating system that is known today as **Linux**. The initial kernel, which was similar to a small subset of UNIX, has grown into a full-scale operating system today. The Linux 2.0 kernel, released in 1997, was accepted as a commercial operating system: it has all features traditionally attributed to UNIX.

mponents

Linux has the following components.

Kernel

The kernel is responsible for all duties attributed to a kernel, such as memory management, process management, device management, and file management.

System libraries

The system libraries hold a set of functions used by the application programs, including the shell, to interact with the kernel.

System utilities

The system utilities are individual programs that use the services provided by the system libraries to perform management tasks.

orking
ilities

Linux supports the standard Internet protocols discussed in Chapter 6. It supports three layers: the socket interface, protocol drivers, and network device drivers.

Security

Linux' security mechanism provides the security aspects defined traditionally for UNIX, such as authentication and access control.

Windows NT/2000/XP

In the late 1980s Microsoft, under the leadership of Dave Cutler, started development of a new single-user operating system to replace **MS-DOS** (Microsoft Disk Operating System). **Windows NT** (NT standing for New Technology) was the result. Several versions of Windows NT followed and the name was changed to **Windows 2000**. Windows XP (XP stands for eXPerience) was released in 2001. We refer to all of these versions as Windows NT or just NT.

Design goals

Design goals released by Microsoft are *extensibility, portability, reliability, compatibility,* and *performance*.

Extensibility

Windows NT is designed as a modular architecture with several layers. The purpose is to allow the higher layers to be changed with time without affecting the lower layers.

Portability

NT, like UNIX, is mostly is written in C or C++ and the code is independent of the machine language of the computer on which it is running.

Reliability

Windows NT was designed to handle error conditions including protection from malicious software. NT uses a file system called the **NT file system (NTFS)** that recovers from file-system errors.

Compatibility

NT was designed to run programs written for other operating systems and the earlier versions of Windows NT.

Performance

NT was designed to have a fast response time to applications that run on top of the operating system.

Architecture

NT uses a layered architecture, as shown in Figure 7.21.

HAL

The **hardware abstraction layer (HAL)** hides hardware differences from the upper layers.

Kernel

The kernel is the heart of the operating system. It is an object-oriented piece of software that sees any entity as an object.

Executive

The NT executive provides services for the whole operating system. It is made up of six subsystems: object manager, security reference monitor, process manager,

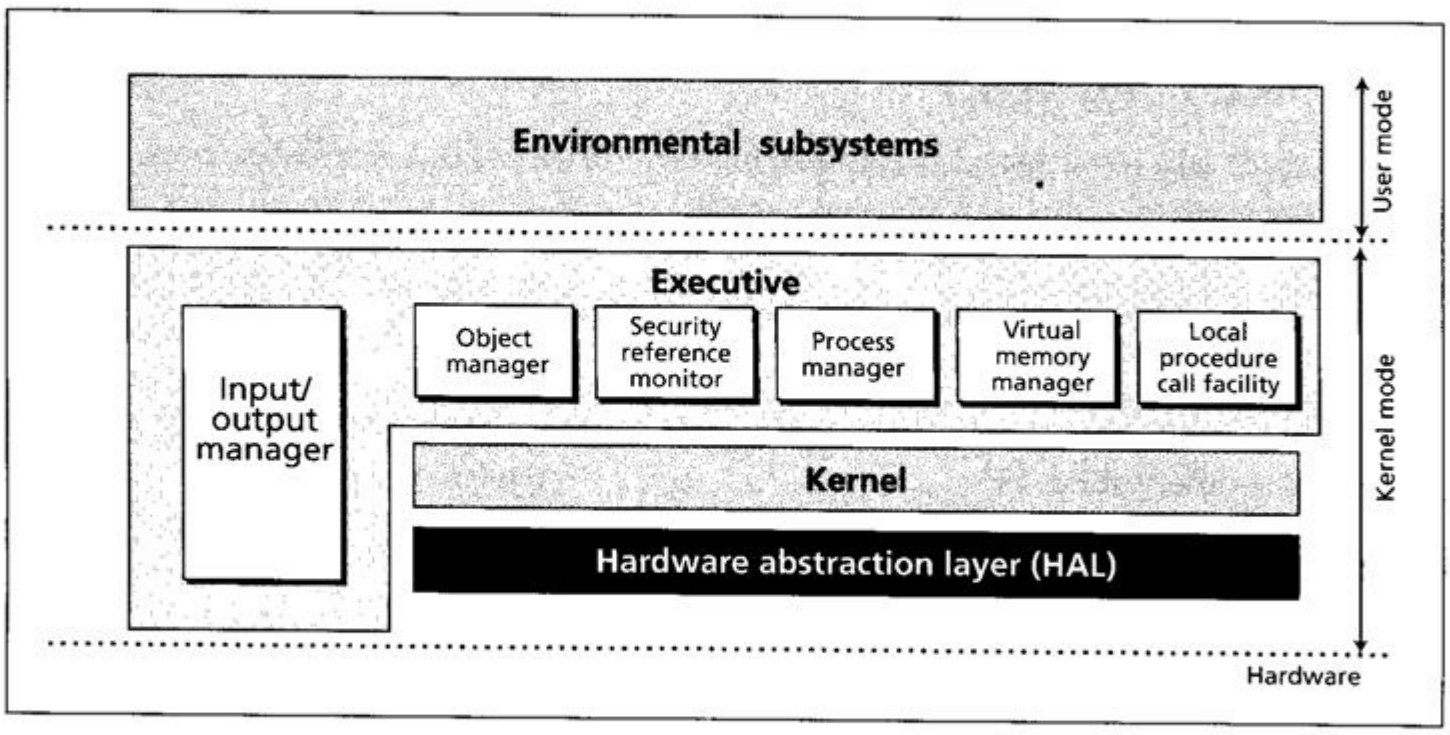


Figure 7.21 The architecture of Windows NT

virtual memory manager, local procedure call facility, and the I/O manager. Most of these subsystems are familiar from our previous discussions of operating subsystems. Some subsystems, like the object manager, are added to NT because of its object-oriented nature. The executive runs in kernel (privileged) mode.

Environmental subsystems

These are subsystems designed to allow NT to run application programs designed for NT, for other operating systems, or for earlier versions of NT. The native subsystem that runs application designed for NT is called Win32. The environment subsystems run in user mode (a non-privileged mode).

RECOMMENDED READING

[illegible] details about the subjects discussed in this chapter, the following books are recommended:

[illegible] L and Shaw A: *Operating Systems Principles*, Upper Saddle River, NJ: Prentice Hall, 2003

[illegible]Hoes A and Flynn I: *Understanding Operating Systems*, Boston, MA: Course Technology, [illegible]07

[illegible] G: *Operating Systems: A Modern Perspective*, Reading, MA: Addison Wesley, 2001

[illegible]erschatz A and Galvin P: *Operating System Concepts*, New York: Wiley, 2004

7.6 KEY TERMS

This chapter has introduced the following key terms, which are listed here with the pages on which they first occur:

batch operating system 189
circular waiting 201
demand paging 195
demand segmentation 195
distributed system 190
graphical user interface (GUI) 192
job 197
kernel 204
memory management 192
monoprogramming 192
mutual exclusion 201
no-preemption 201
page 194
parallel system 190
process 190
program 197
ready state 197
resource holding 201
scheduler 198
shell 191
software 188
state diagram 197
time sharing 190
user interface 191
virtual memory 196
window 192
Windows NT 206
bootstrap 189
deadlock 200
demand paging and segmentation 195
device manager 202
frame 194
hold state 197
job scheduler 198
Linux 205
Microsoft Disk Operating System (MS-DOS) 206
multiprogramming 190
native NT file system (NTFS) 206
operating system 188
paging 194
partitioning 193
process scheduler 198
queuing 199
real-time system 191
running state 197
scheduling 190
single-user operating system 190
starvation 201
terminal state 197
UNIX 203
utility 205
waiting state 197
Windows 2000 206
Windows XP 206

7.7 SUMMARY

- An operating system is an interface between the hardware of a computer and the user that facilitates the execution of programs and access to hardware and software resources. Two major design goals of an operating system are efficient use of hardware and ease of use of resources.
- Operating systems have gone through a long history of evolution: batch systems, time-sharing systems, personal systems, parallel systems, and distributed systems.
- A modern operating system has at least four functional areas: memory manager, process manager, device manager, and file manager. An operating system also provides a user interface.
- The first responsibility of a modern computer system is memory management. Memory allocation must be controlled by the operating system. Memory management techniques can be divided into two categories: monoprogramming and multiprogramming. In monoprogramming, most of the memory capacity is dedicated to one single program. In multiprogramming, more than one program can be in memory at the same time.
- The second responsibility of an operating system is process management. A process is a program in execution. The process manager uses schedulers and queues to manage processes. Process management involves synchronizing different processes with different resources. This may potentially create resource deadlock or starvation. Deadlock occurs when the operating system does not put resource restrictions on processes: starvation can happen when the operating system puts too many resource restrictions on a process.
- The third responsibility of an operating system is device or input/output management. There are limitations on the number and speed of input/output devices in a computer system. Because these devices are much slower compared with the CPU and memory, when a process accesses an input/output device, it is not available to other processes. The device manager is responsible for the efficient use of input/output devices.
- The fourth responsibility of an operating system is file management. An operating system uses a file manager to control access to files. Access is permitted only by processes or users that are allowed access to specific files, and the type of access can vary.
- Two common operating systems with some similarities are UNIX and Linux. UNIX is a multiuser, multiprocessing, portable operating system made up from four parts: the kernel, the shell, a standard set of utilities, and application programs. Linux has three components: a kernel, system utilities, and a system library.
- A popular family of operating systems from Microsoft is referred to as Windows NT. Windows NT is an object-oriented, multi-layer operating system. It uses several layers, including a hardware abstract layer (HAL), executive layer, and an environment subsystem layer.

7.8 PRACTICE SET

Review questions

1. What is the difference between an application program and an operating system?
2. What are the components of an operating system?
3. What is the difference between monoprogramming and multiprogramming?
4. How is paging different from partitioning?
5. How is demand paging more efficient than regular paging?
6. How is a program related to a job? How is a job related to a process? How is a program related to a process?
7. Where does a program reside? Where does a job reside? Where does a process reside?
8. What is the difference between a job scheduler and a process scheduler?
9. Why does an operating system need queues?
10. How does deadlock differ from starvation?

Multiple-choice questions

11. _____ is a program that facilitates the execution of other programs.
    a. An operating system
    b. Hardware
    c. A queue
    d. An application program
12. _____ supervises the activity of each component in a computer system.
    a. An operating system
    b. Hardware
    c. A queue
    d. An application program
13. Multiprogramming requires a _____ operating system.
    a. batch
    b. time-sharing
    c. parallel
    d. distributed
14. _____ is multiprogramming with swapping.
    a. Partitioning
    b. Paging
    c. Demand paging
    d. Queuing
15. _____ is multiprogramming without swapping.
    a. Partitioning
    b. Virtual memory
    c. Demand paging
    d. Queuing
16. In _____, only one program can reside in memory for execution.
    a. monoprogramming
    b. multiprogramming
    c. partitioning
    d. paging
17. _____ is a multiprogramming method in which multiple programs are entirely in memory with each program occupying a contiguous space.
    a. Partitioning
    b. Paging
    c. Demand paging
    d. Demand segmentation
18. In paging, a program is divided into equally sized sections called _____.
    a. pages
    b. frames
    c. segments
    d. partitions
19. In _____, the program can be divided into differently sized sections.
    a. partitioning
    b. paging
    c. demand paging
    d. demand segmentation
20. In _____, the program can be divided into equally sized sections called pages, but the pages need not be in memory at the same time for execution.
    a. partitioning
    b. paging
    c. demand paging
    d. demand segmentation
21. A process in the _____ state can go to either the ready, terminated, or waiting states.
    a. hold
    b. virtual
    c. running
    d. a and c
22. A process in the ready state goes to the running state when _____.
    a. it enters memory
    b. it requests I/O
    c. it gets access to the CPU
    d. it finishes running
23. A program becomes a _____ when it is selected by the operating system and brought to the hold state.
    a. job
    b. process
    c. deadlock
    d. partition
24. Every process is a _____.
    a. job
    b. program
    c. partition
    d. a and b

5. The _____ scheduler creates a process from a job and changes a process back to a job.
   a. job
   b. process
   c. virtual
   d. queue
6. The _____ scheduler moves a process from one process state to another.
   a. job
   b. process
   c. virtual
   d. queue
7. To prevent _____, an operating system can put resource restrictions on processes.
   a. starvation
   b. synchronization
   c. paging
   d. deadlock
8. _____ can occur if a process has too many resource restrictions.
   a. Starvation
   b. Synchronization
   c. Paging
   d. Deadlock
9. The _____ manager is responsible for archiving and backup.
   a. memory
   b. process
   c. device
   d. file
10. The _____ manager is responsible for access to I/O devices.
   a. memory
   b. process
   c. device
   d. file

Exercises

31. A computer has a monoprogramming operating system. If the size of memory is 64 MB and the memory-resident part of the operating system needs 4 MB, what is the maximum size of program that can be run by this computer?
32. Redo Exercise 31 if the operating system automatically allocates 10 MB of memory to data.
33. A monoprogramming operating system runs programs that on average need 10 microseconds access to the CPU and 70 microseconds access to the I/O devices. What percentage of time is the CPU idle?
34. A multiprogramming operating system uses an apportioning scheme and divides the 60 MB of available memory into four partitions of 10 MB, 12 MB, 18 MB, and 20 MB. The first program to be run needs 17 MB and occupies the third partition. The second program needs 8 MB and occupies the first partition. The third program needs 10.5 MB and occupies the second partition. Finally, the fourth program needs 20 MB and occupies the fourth partition. What is the total memory used? What is the total memory wasted? What percentage of memory is wasted?
35. Redo Exercise 34 if all programs need 10 MB of memory.
36. A multiprogramming operating system uses paging. The available memory is 60 MB divided into 15 frames, each of 4 MB. The first program needs 13 MB. The second program needs 12 MB. The third program needs 27 MB.
   a. How many frames are used by the first program?
   b. How many frames are used by the second program?
   c. How many frames are used by the third program?
   d. How many frames are unused?
   e. What is the total memory wasted?
   f. What percentage of memory is wasted?
37. An operating system uses virtual memory but requires the whole program to be in physical memory during execution (no paging or segmentation). The size of physical memory is 100 MB. The size of virtual memory is 1 GB. How many programs of size 10 MB can be run concurrently by this operating system? How many of them can be in memory at any time? How many of them must be on disk?

38. What is the status of a process in each of the following situations?
    a. The process is using the CPU.
    b. The process has finished printing and needs the attention of the CPU again.
    c. The process has been stopped because its time slot is over.
    d. The process is reading data from the keyboard.
    e. The process is printing data.
39. Three processes (A, B, and C) are running concurrently. Process A has acquired File1, but needs File2. Process B has acquired File3, but needs File1. Process C has acquired File2, but needs File3. Draw a diagram for these processes. Is this a deadlock situation?
40. Three processes (A, B, and C) are running concurrently. Process A has acquired File1. Process B has acquired File2, but needs File1. Process C has acquired File3, but needs File2. Draw a diagram for these processes. Is this a deadlock situation? If your answer is "no", show how the processes can eventually finish their tasks.

9.1 EVOLUTION

To write a program for a computer, we must use a computer language. A **computer language** is a set of predefined words that are combined into a program according to predefined rules **(syntax)**. Over the years, computer languages have evolved from *machine language* to *high-level languages*.

Machine languages

In the earliest days of computers, the only programming languages available were **machine languages**. Each computer had its own machine language, which was made of streams of 0s and 1s. In Chapter 5 we showed that in a primitive hypothetical computer, we need to use eleven lines of code to read two integers, add them, and print the result. These lines of code, when written in machine language, make eleven lines of binary code, each of 16 bits, as shown in Table 9.1.

Table 9.1 Code in machine language to add two integers

Hexadecimal	*Code in machine language*			
$(1FEF)_{16}$	0001	1111	1110	1111
$(240F)_{16}$	0010	0100	0000	1111
$(1FEF)_{16}$	0001	1111	1110	1111
$(241F)_{16}$	0010	0100	0001	1111
$(1040)_{16}$	0001	0000	0100	0000
$(1141)_{16}$	0001	0001	0100	0001
$(3201)_{16}$	0011	0010	0000	0001
$(2422)_{16}$	0010	0100	0010	0010
$(1F42)_{16}$	0001	1111	0100	0010
$(2FFF)_{16}$	0010	1111	1111	1111
$(0000)_{16}$	0000	0000	0000	0000

Machine language is the only language understood by the computer hardware, which is made of electronic switches with two states: off (representing 0) and on (representing (1).

The only language understood by a computer is machine language.

Although a program written in machine language truly represents how data is manipulated by the computer, it has at least two drawbacks. First, it is machine-dependent. The machine language of one computer is different than the machine language of another computer if they use different hardware. Second, it is very tedious to write programs in this language and very difficult to find errors. The co

of machine language is now is referred to as the *first generation* of programming languages.

Assembly languages

The next evolution in programming came with the idea of replacing binary code for instructions and addresses with symbols or mnemonics. Because they used symbols, these languages were first known as *symbolic languages*. These mnemonic languages were later referred to as **assembly languages**. The assembly language for our hypothetical computer to replace the machine language in Table 9.2 is shown in Program 9.1.

Table 9.2 Code in assembly language to add two integers

Code in assembly language				*Description*
LOAD	**RF**	**Keyboard**		Load from keyboard controller to register F
STORE	**Number1**	**RF**		Store register F into Number1
LOAD	**RF**	**Keyboard**		Load from keyboard controller to register F
STORE	**Number2**	**RF**		Store register F into Number2
LOAD	**R0**	**Number1**		Load Number1 into register 0
LOAD	**R1**	**Number2**		Load Number2 into register 1
ADDI	**R2**	**R0**	**R1**	Add registers 0 and 1 with result in register 2
STORE	**Result**	**R2**		Store register 2 into Result
LOAD	**RF**	**Result**		Load Result into register F
STORE	**Monitor**	**RF**		Store register F into monitor controller
HALT				Stop

A special program called an **assembler** is used to translate code in assembly language into machine language.

High-level languages

Although assembly languages greatly improved programming efficiency, they still required programmers to concentrate on the hardware they were using. Working with symbolic languages was also very tedious, because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of **high-level languages**.

High-level languages are portable to many different computers, allowing the programmer to concentrate on the application rather than the intricacies of the computer's organization. They are designed to relieve the programmer from the details of assembly language. High-level languages share one characteristic with symbolic languages: they must be converted to machine language. This process is called *interpretation* or *compilation* (described later in the chapter).

Over the years, various languages, most notably BASIC, COBOL, Pascal, Ada, C, C++, and Java, were developed. Program 9.1 shows the code for adding two integers as it would appear in the C++ language. Although the program looks longer, some of the lines are used for documentation (comments).

Program 9.1 Addition program in C++

```
/*      This program reads two integers from the keyboard and prints their
        sum.
        Written by:
        Date:
*/
#include <iostream.h>
using namespace std;
int main (void)
{
        // Local Declarations
        int number1;
        int number2;
        int result;
        // Statements
        cin >> number1;
        cin >> number2;
        result = number1 + number2;
        cout << result;
        return 0;
} // main
```

9.2 TRANSLATION

Programs today are normally written in one of the high-level languages. To run the program on a computer, the program needs to be translated into the machine language of the computer on which it will run. The program in a high-level language is called the **source program**. The translated program in machine language is called the **object program**. Two methods are used for translation: **compilation** and **interpretation**.

Compilation

A **compiler** normally translates the whole source program into the object program.

Interpretation

Some computer languages use an **interpreter** to translate the source program into the object program. Interpretation refers to the process of translating each line of the source program into the corresponding line of the object program and executing the line. However, we need to be aware of two trends in interpretation: that used by some languages before Java and the interpretation used by Java.

First approach to interpretation

Some interpreted languages prior to Java (such as BASIC and APL) used a kind of interpretation process that we refer to as the *first approach* to interpretation for the lack of any other name. In this type of interpretation, each line of the source program is translated into the machine language of the computer being used and executed immediately. If there are any errors in translation and execution, the process displays a message and the rest of the process is aborted. The program needs to be corrected and be interpreted and executed again from the beginning. This first approach was considered to be a slow process, which is why most languages use compilation instead of interpretation.

Second approach to interpretation

With the advent of Java, a new kind of interpretation process was introduced. The Java language is designed to be portable to any computer. To achieve portability, the translation of the source program to the object program is done in two steps: compilation and interpretation. A Java source program is first compiled to create Java **bytecode**, which looks like code in a machine language, but is not the object code for any specific computer: it is the object code for a virtual machine, called the Java Virtual Machine or JVM. The byte code then can be compiled or interpreted by any computer that runs a JVM emulator—that is, the computer that runs the bytecode needs only a JVM emulator, not the Java compiler.

Translation process

Compilation and interpretation differ in that the first translates the whole source code before executing it, while the second translates and executes the source code a line at a time. Both methods, however, follow the same translation process shown in Figure 9.1.

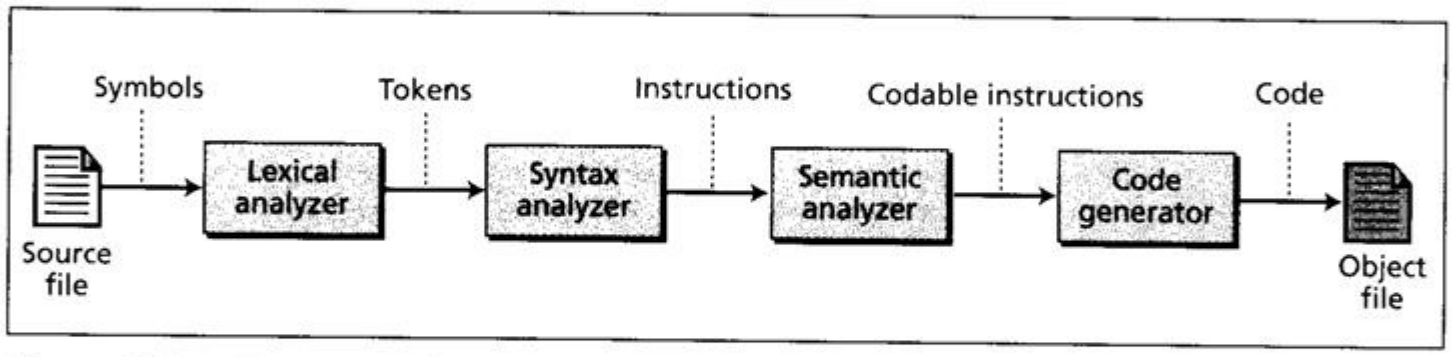


Figure 9.1 Source code translation process

Lexical analyzer

A **lexical analyzer** reads the source code, symbol by symbol, and creates a list of **tokens** in the source language. For example, the five symbols *w, h, i, l, e* are read and grouped together as the token *while* in the C, C++, or Java languages.

Syntax analyzer

The **syntax analyzer** parses a set of tokens to find instructions. For example, the token "*x*", "=", "0" are used by the syntax analyzer to create the assignment statement in the C language "$x = 0$". We will discuss the function of a parser and a syntax analyzer in more detail when we describe language recognition in artificial intelligence in Chapter 18.

Semantic analyzer

The **semantic analyzer** checks the sentences created by the syntax analyzer to be sure that they contain no ambiguity. Computer languages are normally unambiguous, which means that this stage is either omitted in a translator, or its duty is minimal. We also discuss semantic analysis in more details in Chapter 18.

Code generator

After unambiguous instructions are created by the semantic analyzer, each instruction is converted to a set of machine language instructions for the computer on which the program will run. This is done by the **code generator**.

9.3 PROGRAMMING PARADIGMS

Today computer languages are categorized according to the approach they use to solve a problem. A *paradigm*, therefore, is a way in which a computer language looks at the problem to be solved. We divide computer languages into four paradigms: *procedural* (imperative), *object-oriented*, *functional*, and *declarative*. Figure 9.2 summarizes these.

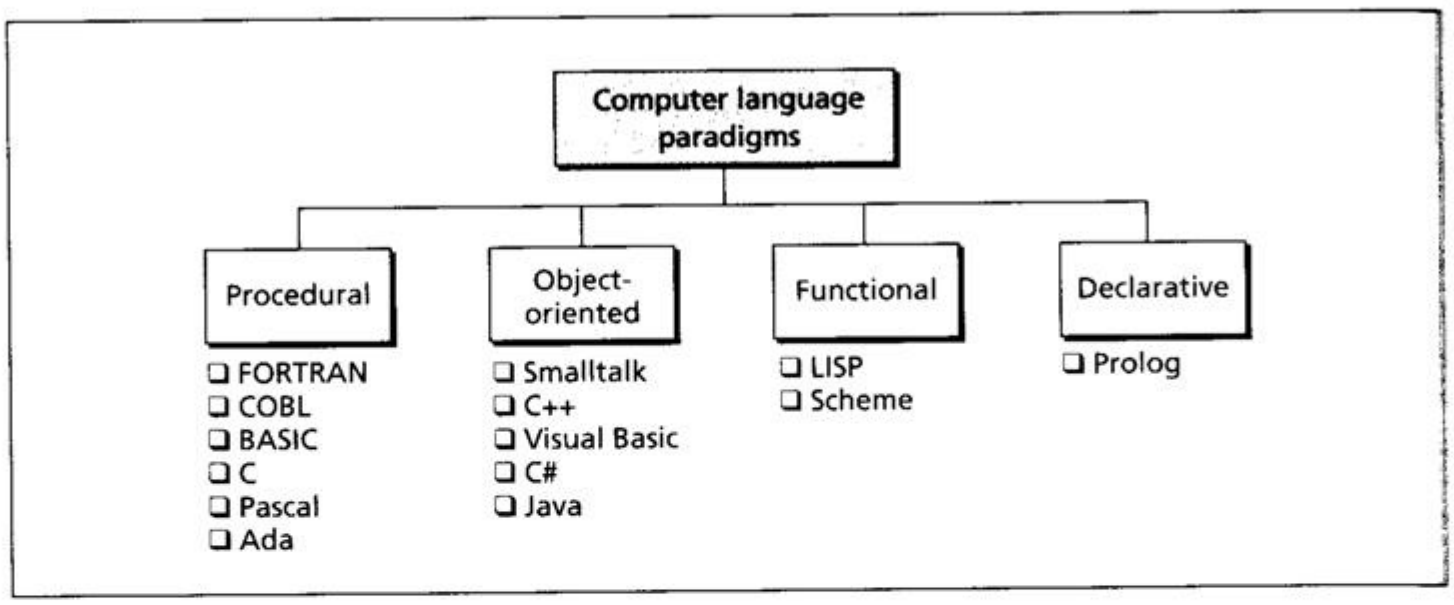


Figure 9.2 Categories of programming languages

The procedural paradigm

In the **procedural paradigm** (or **imperative paradigm**) we can think of a program as an *active agent* that manipulates *passive objects*. We encounter many passive

objects in our daily life: a stone, a book, a lamp, and so on. A passive object cannot initiate an action by itself, but it can receive actions from active agents.

A program in a procedural paradigm is an active agent that uses passive objects that we refer to as *data* or *data items*. Data items, as passive objects, are stored in the memory of the computer, and a program manipulates them. To manipulate a piece of data, the active agent (program) issues an action, referred to as a *procedure*. For example, think of a program that prints the contents of a file. To be printed, the file needs to be stored in memory (or some registers that act as memory). The file is a passive object or a collection of passive objects. To print the file, the program uses a procedure, which we call *print*. The procedure print has usually been written previously to include all the actions required to tell the computer how to print each character in the file. The program invokes or *calls* the procedure *print*. In a procedural paradigm, the object (*file*) and the procedure (*print*) are completely separate entities. The object (*file*) is an independent entity that can receive the *print* action, or some other actions, such as *delete*, *copy*, and so on. To apply any of these actions to the file, we need a procedure to act on the file. The procedure *print* (or *copy* or *delete*) is a separate entity that is written and the program only triggers it.

To avoid writing a new procedure each time we need to print a file, we can write a general procedure that can print any file. When we write this procedure, every reference to the file name is replaced by a symbol, such as F, or FILE, or something else. When the procedure is called (triggered), we pass the name of the actual file to be printed to the procedure, so that we can write a procedure called *print* but call it twice in the program to print two different files. Figure 9.3 shows how a program can call different predefined procedures to print or delete different object files.

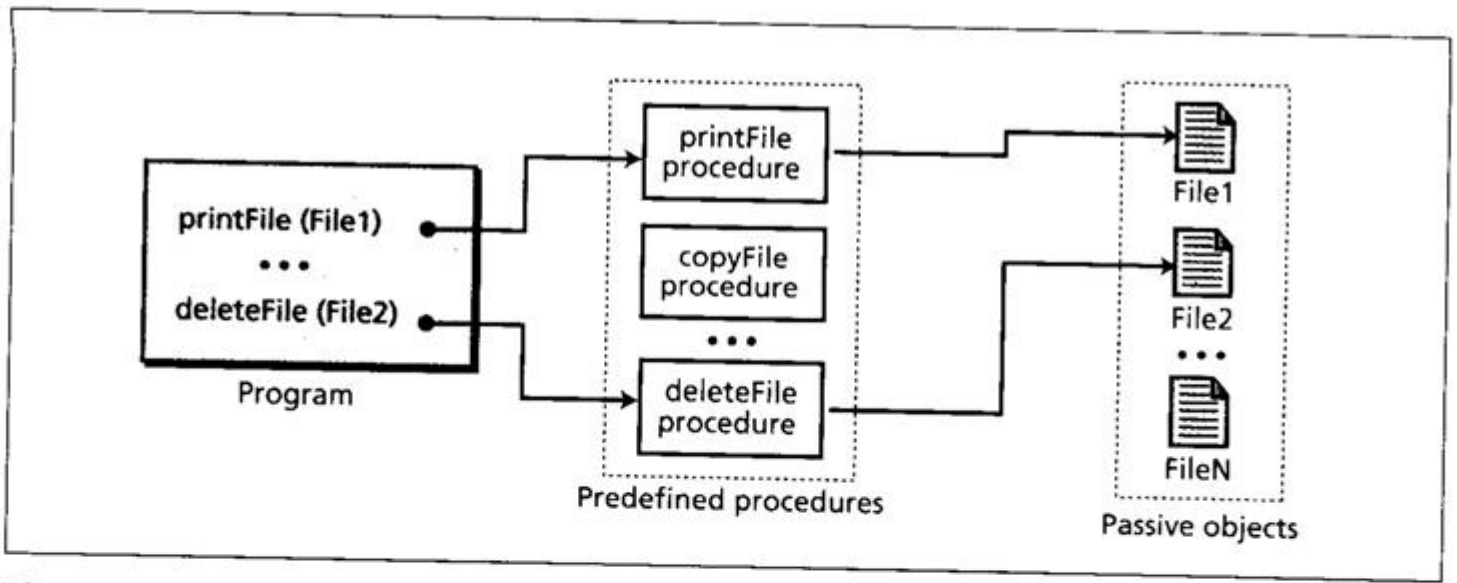


Figure 9.3 The concept of the procedural paradigm

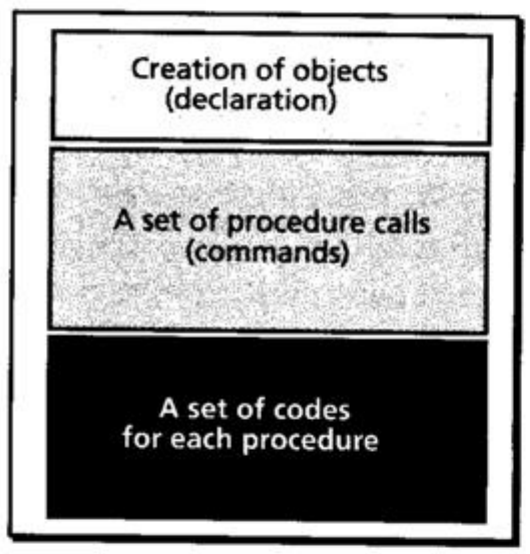
We need to separate the procedure from its triggering by the program. The program does not define the procedure (as explained later), it only triggers or calls the procedure. The procedure must already exist.

When we use a procedural high-level language, the program consists of nothing but a lot of procedure calls. Although it is not immediately obvious, even when we use a simple mathematical operator such as the addition operator (+), we are using

a procedure call to a procedure that is already written. For example, when we use the expression A + B and expect the expression to add the value of two objects A and B, we are calling the procedure *add* and passing the name of these two objects to the procedure. The procedure *add* needs two objects to act on. It adds the values of the two objects and returns the result. In other words, the expression A + B is a short cut for *add* (A, B). The designer of the language has written this procedure and we can call it.

If we think about the procedures and the objects to be acted upon, the concept of the procedural paradigm becomes simpler to understand. A program in this paradigm is made up of three parts: a part for object creation, a set of procedure calls, and a set of code for each procedure. Some procedures have already been defined in the language itself. By combining this code, the programmer can create new procedures.

Figure 9.4 shows these three components of a procedural program. There are also extra tokens in the language that are used for delimiting or organizing the calls, but these are not shown in the figure.



A procedural program

Figure 9.4 The components of a procedural program

Some procedural languages

Several high-level imperative (procedural) languages have been developed over the last few decades, such as FORTRAN, COBOL, Pascal, C, and Ada.

FORTRAN

FORTRAN (FORmula TRANslation), designed by a group of IBM engineers under the supervision of Jack Backus, became commercially available in 1957. FORTRAN was the first high-level language. During the last forty years FORTRAN has gone through several versions: FORTRAN, FORTRAN II, FORTRAN IV, FORTRAN 77, FORTRAN 99, and HPF (High Performance FORTRAN). The newest version (HPF) is used in high-speed multiprocessor computer systems. FORTRAN

has some features that, even after four decades, still make it an ideal language for scientific and engineering applications. These features can be summarized as:

- High-precision arithmetic
- Capability of handling complex numbers
- Exponentiation computation (a^b)

COBOL

COBOL (COmmon Business-Oriented Language) was designed by a group of computer scientists under the direction of Grace Hopper of the US Navy. COBOL had a specific design goal: to be used as a business programming language. The problems to be solved in a business environment are totally different from those in an engineering environment. The programming needs of the business world can be summarized as follows:

- Fast access to files and databases
- Fast updating of files and databases
- Large amounts of generated reports
- User-friendly formatted output

Pascal

Pascal was invented by Niklaus Wirth in 1971 in Zurich, Switzerland. It was named after Blaise Pascal, the 17th century French mathematician and philosopher who invented the Pascaline calculator. Pascal was designed with a specific goal in mind: to teach programming to novices by emphasizing the structured programming approach. Although Pascal became the most popular language in academia, it never attained the same popularity in industry. Today's procedural languages owe a lot to this language.

C

The **C language** was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. It was originally intended for writing operating systems and system software—most of the UNIX operating system is written in C. Later, it became popular among programmers for several reasons:

- C has all the high-level instructions a structured high-level programming language should have: it hides the hardware details from the programmer.
- C also has some low-level instructions that allow the programmer to access the hardware directly and quickly: C is closer to assembly language than any other high-level language. This makes it a good language for system programmers.
- C is a very efficient language: its instructions are short. This conciseness attracts programmers who want to write short programs.

Ada

Ada was named after Augusta Ada Byron, the daughter of Lord Byron and the assistant to Charles Babbage, the inventor of the Analytical Engine. It was created for the US Department of Defense (DoD) to be the uniform language used by all DoD

contractors. Ada has three features that make it very popular for the DoD, and industry:

- Ada has high-level instructions like other procedural languages.
- Ada has instructions to allow real-time processing. This makes it suitable for process control.
- Ada has parallel-processing capabilities. It can be run on mainframe computers with multiple processors.

The object-oriented paradigm

The **object-oriented paradigm** deals with active objects instead of passive objects. We encounter many active objects in our daily life: a vehicle, an automatic door, a dishwasher, and so on. The action to be performed on these objects are included in the object: the objects need only to receive the appropriate stimulus from outside to perform one of the actions.

Returning to our example in the procedural paradigm, a file in an object-oriented paradigm can be packed with all the procedures—called **methods** in the object-oriented paradigm—to be performed by the file: printing, copying, deleting, and so on. The program in this paradigm just sends the corresponding request to the object (print, delete, copy, and so on) and the file will be printed, copied, or deleted. Figure 9.5 illustrates the concept.

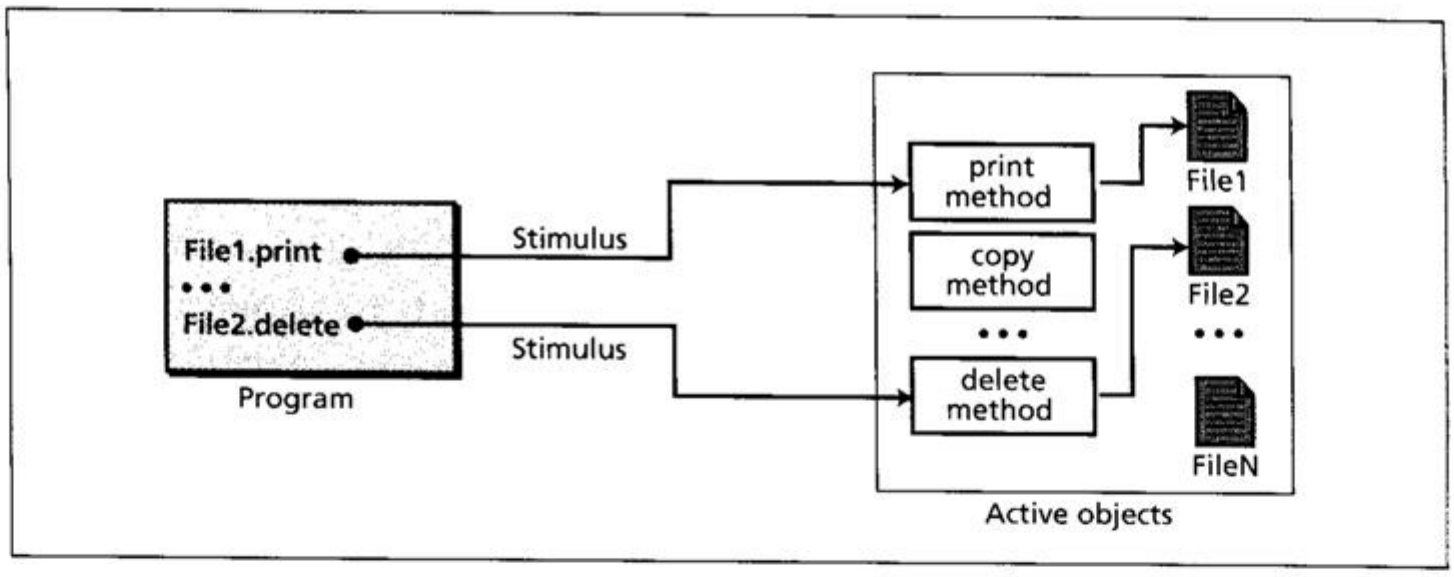


Figure 9.5 The concept of an object-oriented paradigm

The methods are shared by all objects of the same type, and also for other objects that are inherited from these objects, as we discuss later. If the program wants to print File1, it just sends the required stimulus to the active objects and File1 will be printed.

Comparing the procedural paradigm with the object-oriented paradigm (Figure 9.3 and Figure 9.5), we see that the procedures in the procedural paradigm are independent entities, but the methods in the object-oriented paradigm belong to the object's territory.

Classes

As Figure 9.5 shows, objects of the same type (files, for example) need a set of methods that show how an object of this type reacts to stimuli from outside the object's "territories". To create these methods, object-oriented languages such as C++, Java, and C# (pronounced "C sharp") use a unit called a **class**, as shown in Figure 9.6. The exact format of this program unit is different for different object-oriented languages (see Appendix F).

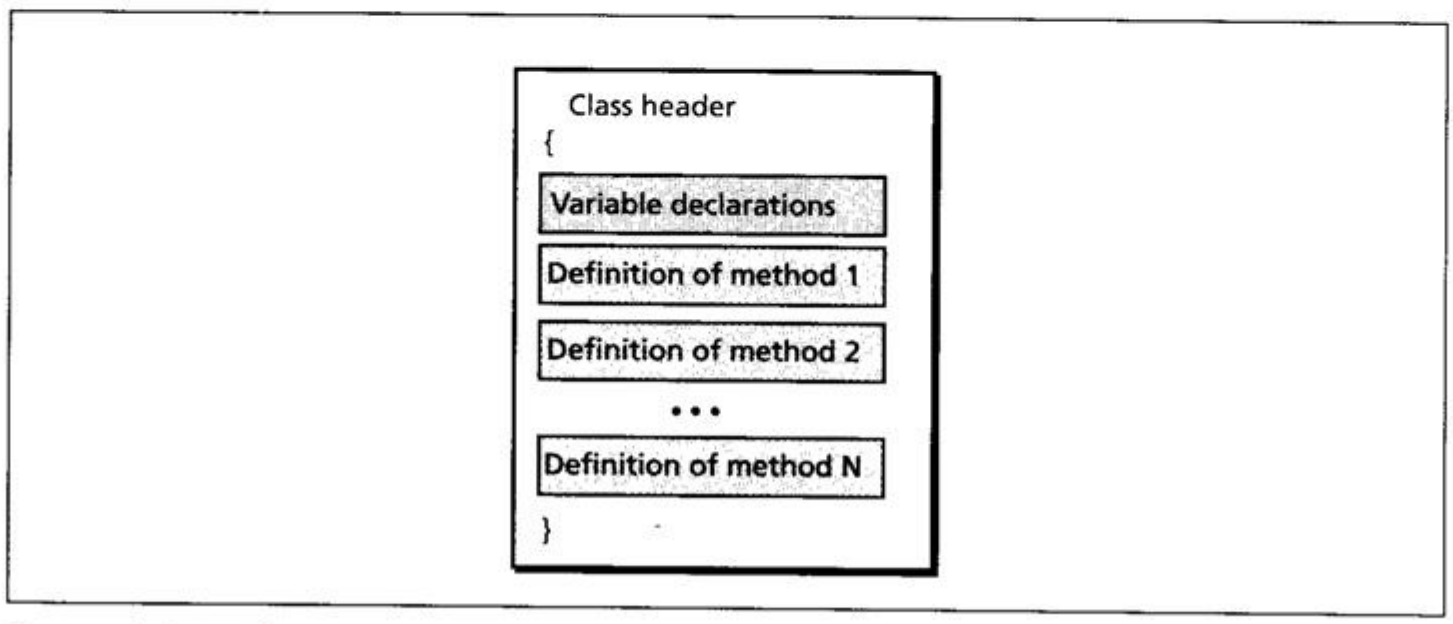


Figure 9.6 The components of a class

ıds In general, the format of methods are very similar to the functions used in some procedural languages. Each **method** has its header, its local variables, and its statement. This means that most of the features we discussed for procedural languages are also applied to methods written for an object-oriented program. In other words, we can claim that object-oriented languages are actually an extension of procedural languages with some new ideas and some new features. The C++ language, for example, is an object-oriented extension of the C language. The C++ language can even be used as a procedural language with no or minimum use of objects. The Java language is an extension of C++, but it is totally an object-oriented language.

:ance In the object-oriented paradigm, as in nature, an object can inherit from another object. This concept is called **inheritance**. When a general class is defined, we can define a more specific class that inherits some of the characteristics of the general class, but also has some new characteristics. For example, when an object of the type *GeometricalShapes* is defined, we can define a class called *Rectangles*. Rectangles are geometrical shapes with additional characteristics.

ɔrphism **Polymorphism** means "many forms". Polymorphism in the object-oriented paradigm means that we can define several operations with the same name that can do different things in related classes. For example, assume that we define two classes, Rectangles and Circles, both inherited from the class GeometricalShapes. We define two operations both named *area*, one in Rectangles and one in Circles, that calculate the area of a rectangle or a circle. The two operations have the same name

but do different things, as calculating the area of a rectangle and the area of a circle need different operands and operations.

Some object-oriented languages

Several **object-oriented languages** have been developed. We briefly discuss the characteristics of two: C++ and Java.

C++

The **C++ language** was developed by Bjarne Stroustrup at Bell Laboratory as an improvement of the C language. It uses *classes* to define the general characteristics of similar objects and the operations that can be applied to them. For example, a programmer can define a GeometricalShapes class and all the characteristics common to two-dimensional geometrical shapes, such as center, number of sides, and so on. The class can also define operations (functions or methods) that can be applied to a geometrical shape, such as calculating and printing the area, calculating and printing the perimeter, printing the coordinates of the center point, and so on. A program can then be written to create different objects of type GeometricalShapes. Each object can have a center located at a different point and a different number of sides. The program can then calculate and print the area, perimeter, and the location of the center for each object.

Three principles were used in the design of the C++ language: *encapsulation*, *inheritance*, and *polymorphism*.

Java

Java was developed at Sun Microsystems, Inc. It is based on C and C++, but some features of C++, such as multiple inheritance, are removed to make the language more robust. In addition, the language is totally class-oriented. In C++ one can solve a problem without ever defining a class, but in Java every data item belongs to a class.

A program in Java can either be an application or an **applet**. An application is a complete stand-alone program that can be run independently. An applet, on the other hand, is embedded HTML (see Chapter 6), stored on a server, and run by a browser. The browser can download the applet and run it locally.

In Java, an application program (or an applet) is a collection of classes and instances of those classes. One interesting feature of Java is the *class library*, a collection of classes. Although C++ also provides a class library, in Java the user can build new classes based on those provided by the library.

The execution of a program in Java is also unique. We create a class and pass it to the interpreter, which calls the class methods. Another interesting feature of Java is support for **multithreading**. A thread is a sequence of actions executed one after another. C++ allows only single threading—that is, the whole program is executed as a single process thread—but Java allows the concurrent execution of several lines of code.

The functional paradigm

In the **functional paradigm** a program is considered a mathematical function. In this context, a **function** is a black box that maps a list of inputs to a list of outputs (Figure 9.7).

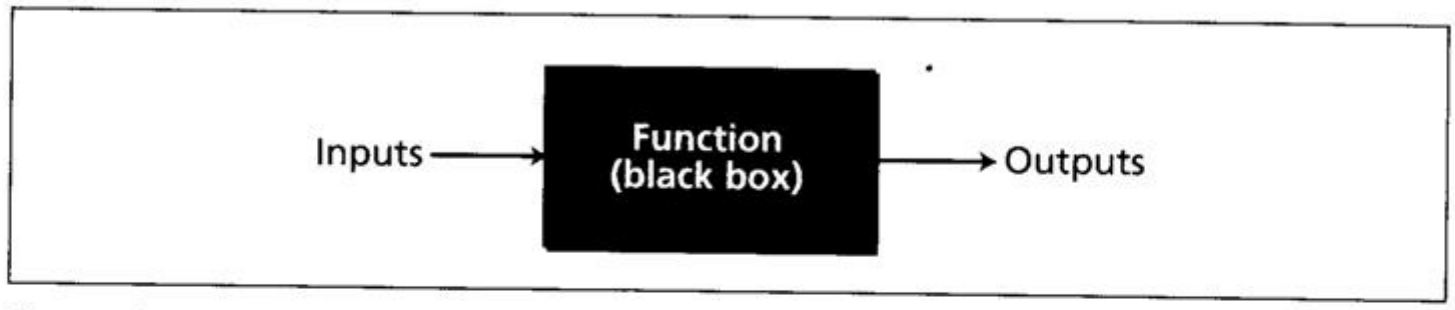


Figure 9.7 A function in a functional language

For example, *summation* can be considered as a function with *n* inputs and only one output. The function takes the *n* inputs, adds them, and creates the sum. A **functional language** does the following:

- Predefines a set of primitive (atomic) functions that can be used by any programmer.
- Allows the programmer to combine primitive functions to create new functions.

For example, we can define a primitive function called *first* that extracts the first element of a list. It may also have a function called *rest* that extracts all the elements except the first. A program can define a function that extracts the third element of a list by combining these two functions as shown in Figure 9.8.

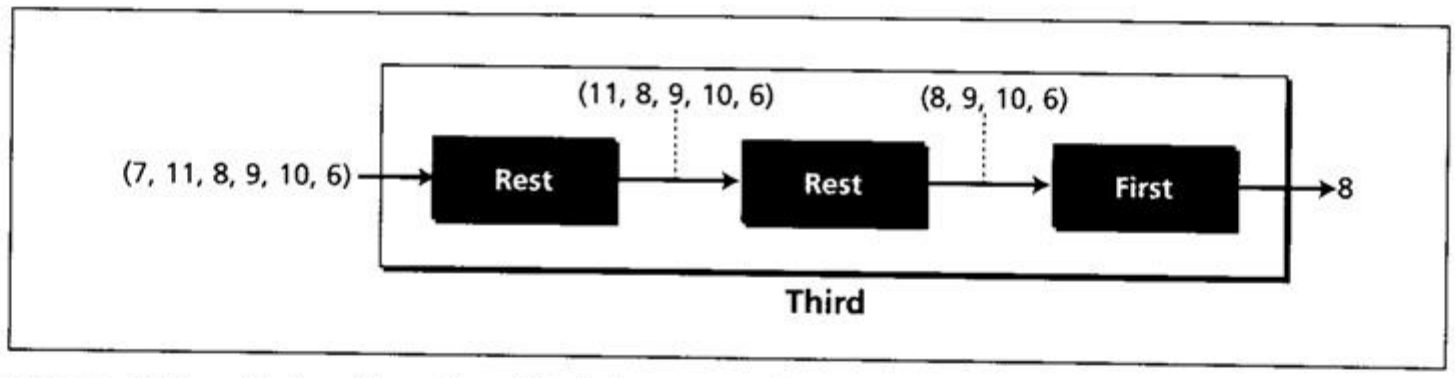


Figure 9.8 Extracting the third element of a list

A functional language has two advantages over a procedural language: it encourages modular programming and allows the programmer to make new functions out of existing ones. These two factors help a programmer create large and less error-prone programs from already-tested programs.

Some functional languages

We briefly discuss LISP and Scheme as examples of functional languages.

LISP

LISP (LISt Programming) was designed by a team of researchers at MIT in the early 1960s. It is a list-processing programming language in which everything is considered a list.

Scheme

The LISP language suffered from a lack of standardization. After a while, there were different versions of LISP everywhere. The de facto standard is the one developed by MIT in the early 1970s called **Scheme**.

The Scheme language defines a set of primitive functions that solves problems. The function name and the list of inputs to the function are enclosed in parentheses. The result is an output list, which can be used as the input list to another function. For example, there is a function, *car*, that extracts the first element of a list. There is a function, called *cdr*, that extracts the rest of the elements in a list except the first one. In other words, we have:

```
(car 2 3 7 8 11 17 20) → 2
(cdr 2 3 7 8 11 17 20) →  3 7 8 11 17 20
```

Now we can combine these two functions to extract the third element of any list.

```
(car (cdr (cdr  list)))
```

If we apply the above function to (2 3 7 8 11 17 20), it extracts 7 because the result of the innermost parentheses is 3 7 8 11 17 20. This becomes the input to the middle parentheses, with the result 7 8 11 17 20. This list now becomes the input to the car function, which takes out the first element, 7.

The declarative paradigm

A **declarative paradigm** uses the principle of logical reasoning to answer queries. It is based on formal logic defined by Greek mathematicians and later developed into *first-order predicate calculus*.

Logical reasoning is based on deduction. Some statements (facts) are given that are assumed to be true, and the logician uses solid rules of logical reasoning to deduce new statements (facts). For example, the famous rule of deduction in logic is:

```
If (A is B) and (B is C), then (A is C)
```

Using this rule and the two following facts,

```
Fact 1: Socrates is a human  →  A is B
Fact 2: A human is mortal →  B is C
```

we can deduce a new fact:

```
Fact 3: Socrates is mortal → A is C
```

Programmers study the domain of their subject—that is, know all the facts in the domain—or get the facts from experts in the field. Programmers also need to be expert in logic to carefully define the rules. The program can then deduce and create new facts.

One problem associated with declarative languages is that a program is specific to a particular domain, because collecting all the facts into one program makes it huge. This is the reason why declarative programming is limited so far to specific fields such as artificial intelligence. We discuss logic further in Chapter 18.

One of the famous declarative languages is **Prolog (PROgramming in LOGic)**, developed by A. Colmerauer in France in 1972. A program in Prolog is made up of facts and rules. For example, the previous facts about human beings can be stated as:

```
human (John)
mortal (human)
```

The user can then ask:

```
?-mortal (John)
```

and the program will respond with *yes*.

9.4 COMMON CONCEPTS

In this section we conduct a quick navigation through some procedural languages to find common concepts. Some of these concepts are also available in most object-oriented languages because, as we explain, an object-oriented paradigm uses the procedural paradigm when creating methods.

ers

One feature present in all procedural languages, as well as in other languages, is the **identifier**—that is, the name of objects. Identifiers allow us to name objects in the program. For example, each piece of data in a computer is stored at a unique address. If there were no identifiers to represent data locations symbolically, we would have to know and use data addresses to manipulate them. Instead, we simply give data names and let the **compiler** keep track of where they are physically located.

pes

A **data type** defines a set of values and a set of operations that can be applied to those values. The set of values for each type is known as the *domain* for the type. Most languages define two categories of data types: *simple types* and *composite types*.

Simple data types

A **simple type** (sometimes called an *atomic type*, *fundamental type*, *scalar type*, or *built-in type*) is a data type that cannot be broken into smaller data types. Several simple data types have been defined in imperative languages:

- An *integer* type is a whole number, that is, a number without a fractional part. The range of values an integer can take depends on the language. Some languages support several integer sizes.
- A *real* type is a number with a fractional part.
- A *character* type is a symbol in the underlying character set used by the language, for example, ASCII or Unicode.
- A *Boolean* type is type with only two values, *true* or *false*.

Composite data types

A **composite type** is a set of elements in which each element is a simple type or a composite type (that is, a recursive definition). Most languages defines the following composite types:

- An *array* is a set of elements each of the same type.
- A *record* is a set of elements in which the elements can be of different types.

Variables

Variables are names for memory locations. As discussed in Chapter 5, each memory location in a computer has an address. Although the addresses are used by the computer internally, it is very inconvenient for the programmer to use addresses for two reasons. First, the programmer does not know the relative address of the data item in memory. Second, a data item may occupy more than one location in memory. Names, as a substitute for addresses, free the programmer to think at the level at which the program is executed. A programmer can use a variable, such as *score*, to store the integer value of a score received in a test. Since a variable holds a data item, it has a type.

Variable declarations

Most procedural and object-oriented languages required that variables be declared before being used. Declaration alerts the computer that a variable with a given name and type will be used in the program. The computer reserves the required storage area and names it. Declaration is part of the object creation we discussed in the previous section. For example, in C, C++, and Java we can declare three variables of type character, integer, and real as shown below:

```
char C;
int num;
double result;
```

The first line declares a variable C to be of type character. The second declares a variable num to be of type integer. The third line declares a variable named result to be of type real.

Variable initialization

Although the value of data stored in a variable may change during the program's execution, most procedural languages allow the initialization of the variables when they are declared. Initialization stores a value in the variable. The following shows how the variables can be declared and initialized at the same time.

```
char C ='Z';
int num = 123;
double result = 256,782;
```

; A **literal** is a predetermined value used in a program. For example, if we need to calculate the area of a circle when the value of the radius is stored in the variable r, we can use the expression $3.14 \times r^2$, in which the approximate value of π (pi) is used as a literal. In most programming languages we can have integer, real, character, and Boolean literals. In most languages, we can also have string literals. To distinguish the character and string literals from the names of variables and other objects, most languages require that the character literals be enclosed in single quotes, such as 'A', and strings to be enclosed in double quotes, such as "Anne".

nts The use of literals are not considered good programming practice unless we are sure that the value of the literal will not change with time (such as the value of π in geometry). However, most literals may change value with time. For example, if a sales tax is 8 percent this year, it may not be the same next year. When we write a program to calculate the cost of items, we should not use the literal in our program.

$$\text{cost} \leftarrow \text{price} \times 1.08$$

For this reason, most programming languages define **constants**. A constant, like a variable, is a named location that can store a value, but the value cannot be changed after it has been defined at the beginning of the program. However, if next year we want to use the program again, we can change just one line at the beginning of the program, the value of the constant. For example, in a C, or C++ program, the tax rate can be defined at the beginning and used during the program.

```
const float taxMultiplier = 1.08;
...
cost = price * taxMultiplier;
```

Note that a constant, like a variable, has a type that must be defined when the constant is declared.

ınd Almost every program needs to read and/or write data. These operations can be quite complex, especially when we read and write large files. Most programming languages use a predefined function for input and output.

Input

Data is **input** by either a statement or a predefined function. The C language has several input functions. For example, the *scanf* function reads data from the keyboard, formats it, and stores it in a variable. The following is an example:

```
scanf ("%d", &num);
```

When the program encounters this instruction, it waits for the user to type an integer. It then stores the value in the variable num. The %d tells the program to expect a decimal integer.

Output

Data is **output** by either a statement or a predefined function. The C language has several output functions. For example, the *printf* function displays a string on the monitor. The programmer can include the value of a variable or variables as part of the string. The following displays the value of a variable at the end of a literal string.

```
printf ("The value of the number is: %d", num);
```

Expressions

An **expression** is a sequence of operands and operators that reduces to a single value. For example, the following is an expression with a value of 13:

```
2 * 5 + 3
```

Operator

An **operator** is a language-specific token that requires an action to be taken. The most familiar operators are drawn from mathematics. For example, multiply (*) is an operator—it indicates that two numbers are to be multiplied together. Every language has operators, and their use is rigorously specified in the syntax, or rules, of the language.

- **Arithmetic operators** are used in most languages. Table 9.3 shows some arithmetic operators used in C, C++, and Java.

Table 9.3 Arithmetic operators

Operator	*Definition*	*Example*
+	Addition	3 + 5
–	Subtraction	2 – 4
*	Multiplication	Num * 5
/	Division (the result is the quotient)	Sum / Count
%	Division (the result is the remainder)	Count % 4
++	Increment (add 1 to the value of the variable)	Count++
– –	Decrement (subtract 1 from the value of the variable)	Count– –

- **Relational operators** compare data to see if a value is greater than, less than, or equal to another value. The result of applying relational operators is a Boolean value (true or false). C, C++, and Java use six relational operators, as shown in Table 9.4:

Table 9.4 Relational operators

Operator	*Definition*	*Example*
<	Less than	Num1 < 5
<=	Less than or equal to	Num1 <= 5
>	Greater than	Num2 > 3
>=	Greater than or equal to	Num2 >= 3
==	Equal to	Num1 == Num2
!=	Not equal to	Num1 != Num2

- **Logical operators** combine Boolean values (true or false) to get a new value. The C language uses three logical operators, as shown in Table 9.5:

Table 9.5 Logical operators

Operator	*Definition*	*Example*
!	Not	! (Num1 < Num2)
&&	And	(Num1 < 5) && (Num2 > 10)
||	Or	(Num1 < 5) || (Num2 > 10)

Operand

An **operand** receives an operator's action. For any given operator, there may be one, two, or more operands. In our arithmetic example, the operands of division are the dividend and the divisor.

Statements

A **statement** causes an action to be performed by the program. It translates directly into one or more executable computer instructions. For example, C, C++, and Java define many types of statements. We discuss some of these in this section.

Assignment statements

An **assignment statement** assigns a value to a variable. In other words, it stores the value in the variable, which has already been created in the declaration section. We use the symbol ← in our algorithm to define assignment. Most languages (like C, C++, and Java) use the symbol = for assignment. Other languages such as Ada or Pascal use := for assignment.

Compound statements

A **compound statement** is a unit of code consisting of zero or more statements. It is also known as a *block*. A compound statement allows a group of statements to be treated as a single entity. A compound statement consists of an opening brace, an

optional statement section, followed by a closing brace. The following shows the makeup of a compound statement.

```
{
   x = 1;
   y = 20;
}
```

Control statements

A program in a procedural language is a set of statements. The statements are normally executed one after another. However, sometimes it is necessary to change this sequential order, for example to repeat a statement or a set of statements, or two different sets of statements to be executed based on a Boolean value. The instruction provided in the machine languages of computers for this type of deviation from sequential execution is the *jump* instruction we discussed briefly in Chapter 5. Early imperative languages used the *go to* statement to simulate the *jump* instruction. Although the *go to* statement can be found in some imperative languages today, the principle of structured programming discourages its use. Instead, structured programming strongly recommends the use of the three constructs of *sequence*, *selection*, and *repetition*, as we discussed in Chapter 8. Control statements in imperative languages are related to selection and repetition.

- Most imperative languages have two-way and multi-way selection statements. Two-way selection is achieved through the *if-else* statement, multi-way selection through the *switch* (or *case)* statement. The UML diagram and the code for the *if-else* statement is shown in Figure 9.9.

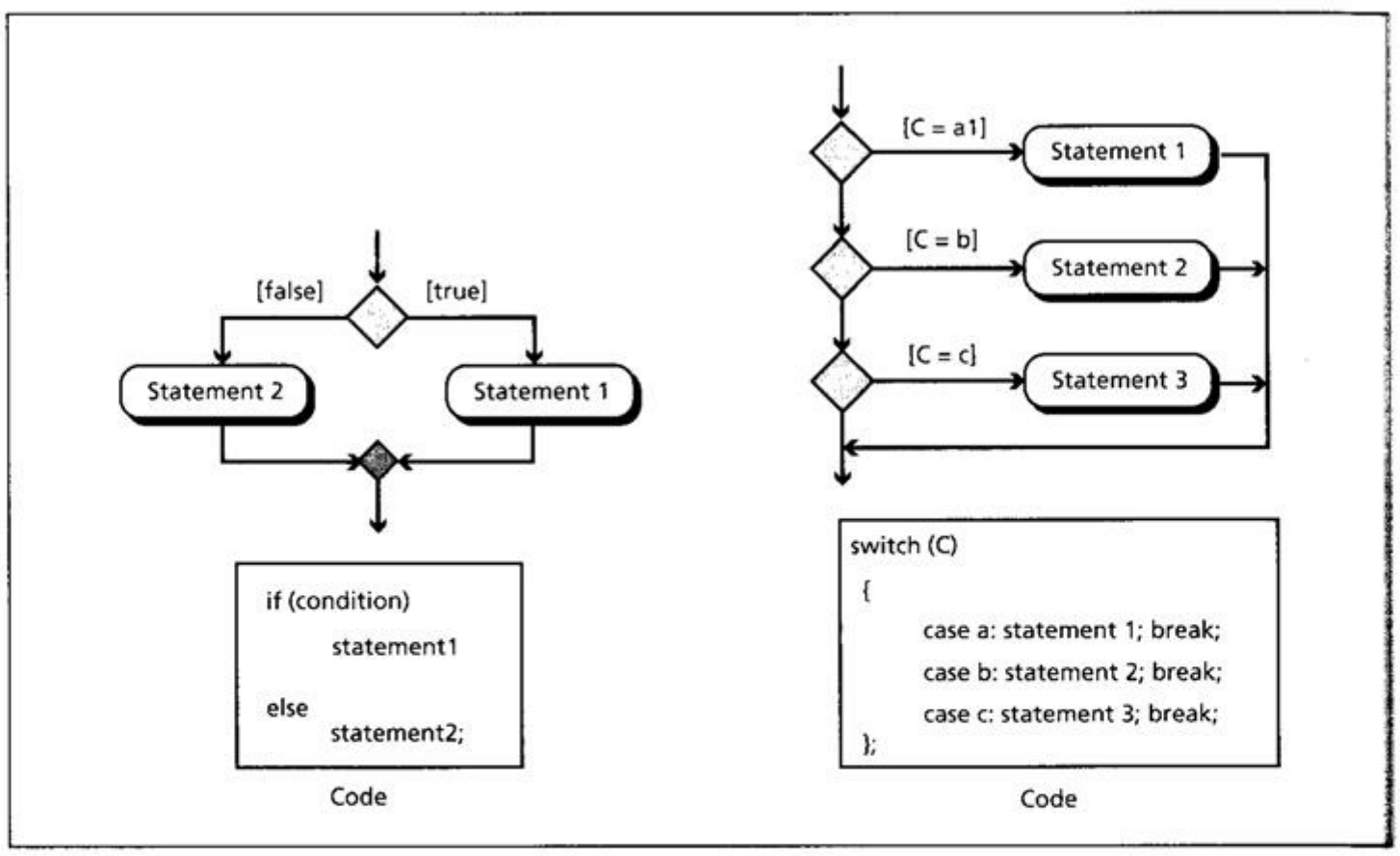


Figure 9.9 Two-way and multi-way decisions

In an *if-else* statement, if the condition is true, statement 1 is executed, while if the condition is false, statement 2 is executed. Both statement 1 and 2 can be any type of statement, including a null statement or a compound statement. Figure 9.9 also shows the code for the *switch* (or *case*) statement. The value of C (a, b, or c) decides which of statement1, statement2, or statement 3 is executed.

- We discussed the repetition construct in Chapter 8. Most of the imperative languages define between one and three loop statements that can achieve repetition. C, C++ and Java define three loop statements, but all of them can be simulated using the *while* loop (Figure 9.10).

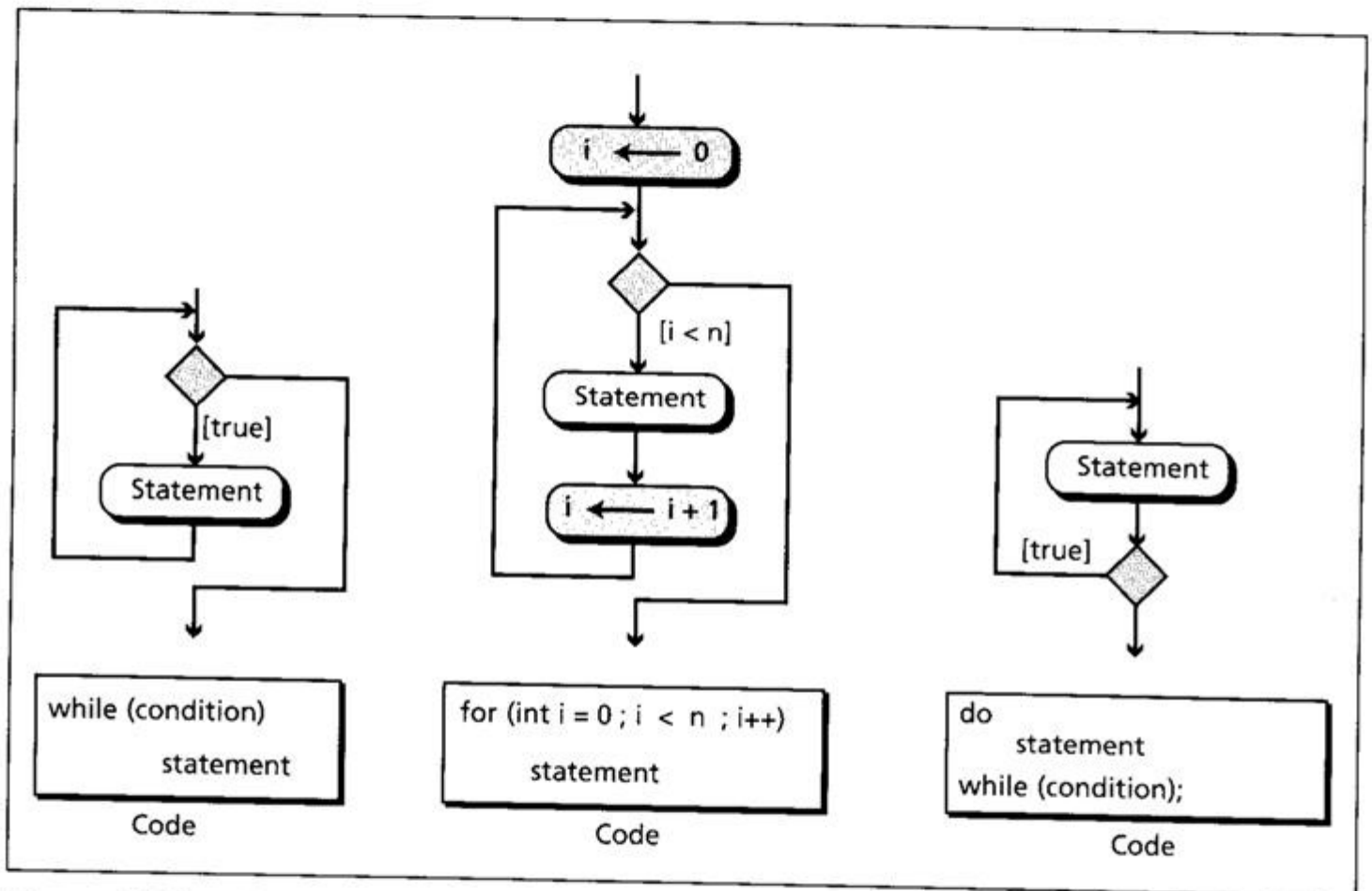


Figure 9.10 Three types of repetition

The main repetition construct in the C language is the *while* loop. A *while* loop is a *pretest* loop: it checks the value of a testing expression. If the value is true, the program goes through one iteration of the loop and tests the value again. The *while* loop is considered an event-controlled loop: the loop continues in iteration until an event happens that changes the value of the test expression from true to false.

The *for* loop is also a pretest loop. However, in contrast to the *while* loop, it is a counter-controlled loop. A counter is set to an initial value and is incremented or decremented in each iteration. The loop is terminated when the value of the counter matches a predetermined value.

The *do* loop is also an event-controlled loop. However, in contrast to the *while* loop, it is a post-test loop. The loop does one iteration and tests the value of an expression. If it is false, it terminates, if true, it does one more iteration and tests again.

Subprograms

In Chapter 8 we showed that a selection sort algorithm can be written as a main program and a **subprogram**. All procedures that are needed to find the smallest item among an unsorted list can be grouped into a subprogram. The idea of subprograms is crucial in procedural languages and to a lesser extent in object-oriented languages. We explained that a program written in a procedural language is a set of procedures that are normally predefined, such as addition, multiplication, and so on. However, sometimes a subset of these procedures to accomplish a single task can be collected and be placed in their own program unit, a subprogram. This is useful because the subprogram makes programming more structural: a subprogram to accomplish a specific task can be written once but called many times, just like predefined procedures in the programming language.

Subprograms also make programming easier: in incremental program development the programmer can test the program step by step by adding a subprogram at each step. This helps to detect errors before the next subprogram is written. Figure 9.11 illustrates the idea of a subprogram.

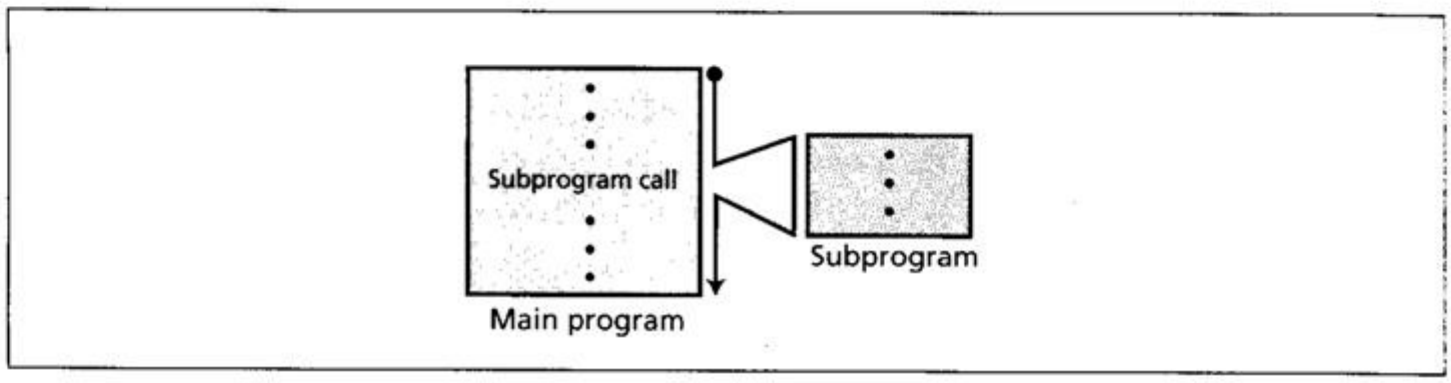


Figure 9.11 The concept of a subprogram

Local variables

In a procedural language, a subprogram, like the main program, can call predefined procedures to operate on local objects. These local objects or **local variables** are created each time the subprogram is called and destroyed when control returns from the subprogram. The local objects *belong* to the subprograms.

Parameters

It is rare for a subprogram to act only upon local objects. Most of the time the main program requires a subprogram to act on an object or set of objects created by the main program. In this case, the program and subprogram use *parameters*. These are referred to as **actual parameters** in the main program and **formal parameters** in the subprogram.

A program can normally pass parameters to a subprogram in either of two ways:

- By *value*
- By *reference*

These are described below.

Pass by value

In parameter **pass by value**, the main program and the subprogram create two different objects (variables). The object created in the program belongs to the program and the object created in the subprogram belongs to the subprogram. Since the territory is different, the corresponding objects can have the same or different names. Communication between the main program and the subprogram is one-way, from the main program to the subprogram. The main program sends the value of the actual parameter to be stored in the corresponding formal parameter in the subprogram: there is no communication of parameter value from the subprogram to the main program.

Example 9.1

Assume that a subprogram is responsible for carrying out printing for the main program. Each time the main program wants to print a value, it sends it to the subprogram to be printed. The main program has its own variable X, the subprogram has its own variable A. What is sent from the main program to the subprogram is the *value* of variable X. This value is stored in the variable A in the subprogram and the subprogram will then print it (Figure 9.12).

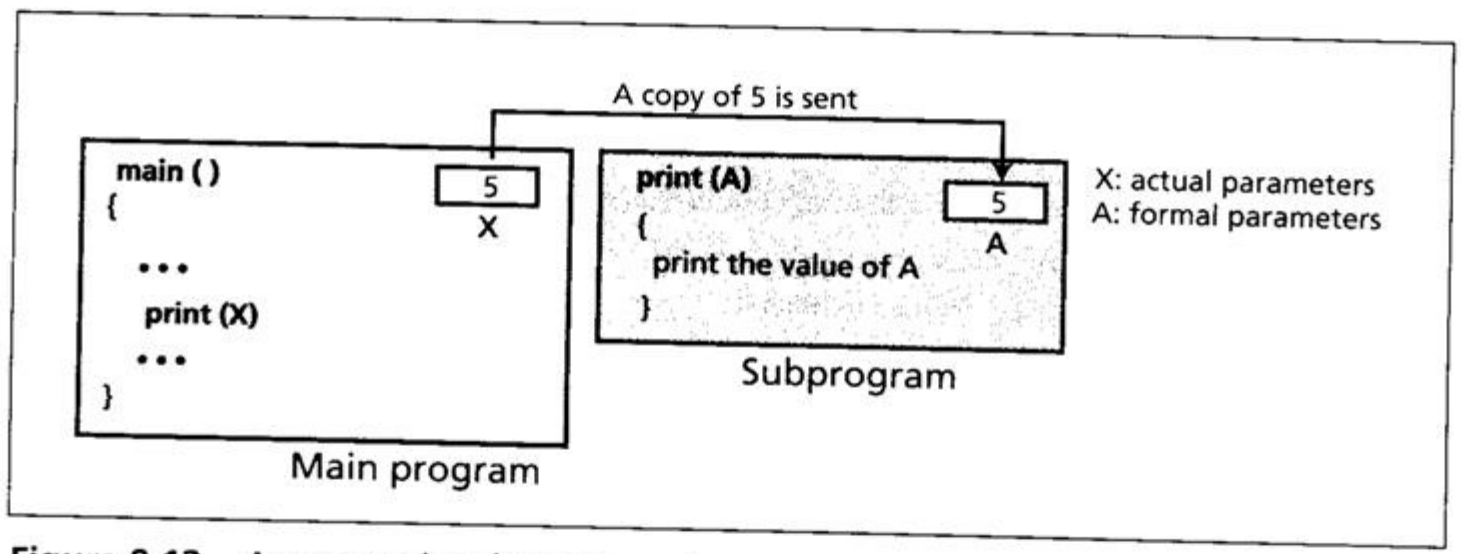


Figure 9.12 An example of pass by value

Example 9.2

In Example 9.1, since the main program sends only a value to the subprogram, it does not need to have a variable for this purpose: the main program can just send a literal value to the subprogram. In other words, the main program can call the subprogram as print (X) or print (5).

Example 9.3

An analogy of pass by value in real life is when a friend wants to borrow and read a valued book that you wrote. Since the book is precious, possibly out of print, you make a copy of the book and pass it to your friend. Any harm to the copy therefore does not affect your book.

Pass by value has an advantage: the subprogram receives only a value. It cannot change, either intentionally or accidentally, the value of the variable in the main program. However, the inability of a subprogram to change the value of the variable

in the main program is a disadvantage when the program actually needs the subprogram to do so.

Example 9.4

Assume that the main program has two variables X and Y that needs to swap their values. The main program calls a subprogram called *swap* to do so. It passes the value of X and Y to the subprogram, which are stored in two variables A and B. The *swap* subprogram uses a local variable T (temporary) and swaps the two values in A and B, but the original values in X and Y remain the same: they are not swapped. This is illustrated in Figure 9.13.

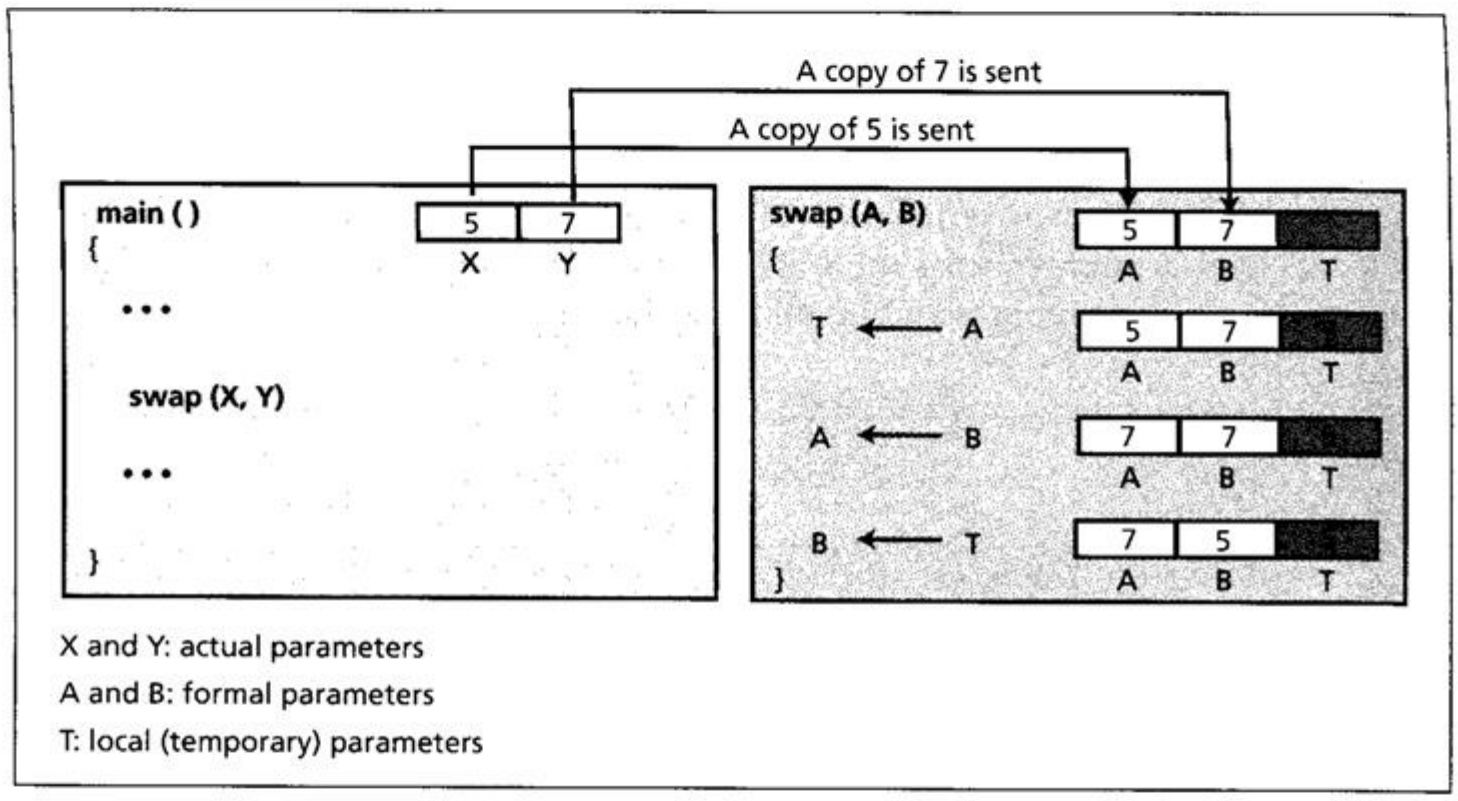


Figure 9.13 An example in which pass by value does not work

Pass by reference

Pass by reference was devised to allow a subprogram to change the value of a variable in the main program. In pass by reference, the variable, which in reality is a location in memory, is shared by the main program and the subprogram. The same variable may have different names in the main program and the subprogram, but both names refer to the same variable. Metaphorically, we can think of pass by reference as a box with two doors: one opens in the main program, the other opens in the subprogram. The main program can leave a value in this box for the subprogram, the subprogram can change the original value and leave a new value for the program in it.

Example 9.5

If we use the same *swap* subprogram but let the variables be passed by reference, the two values in X and Y are actually exchanged, as Figure 9.14 shows.

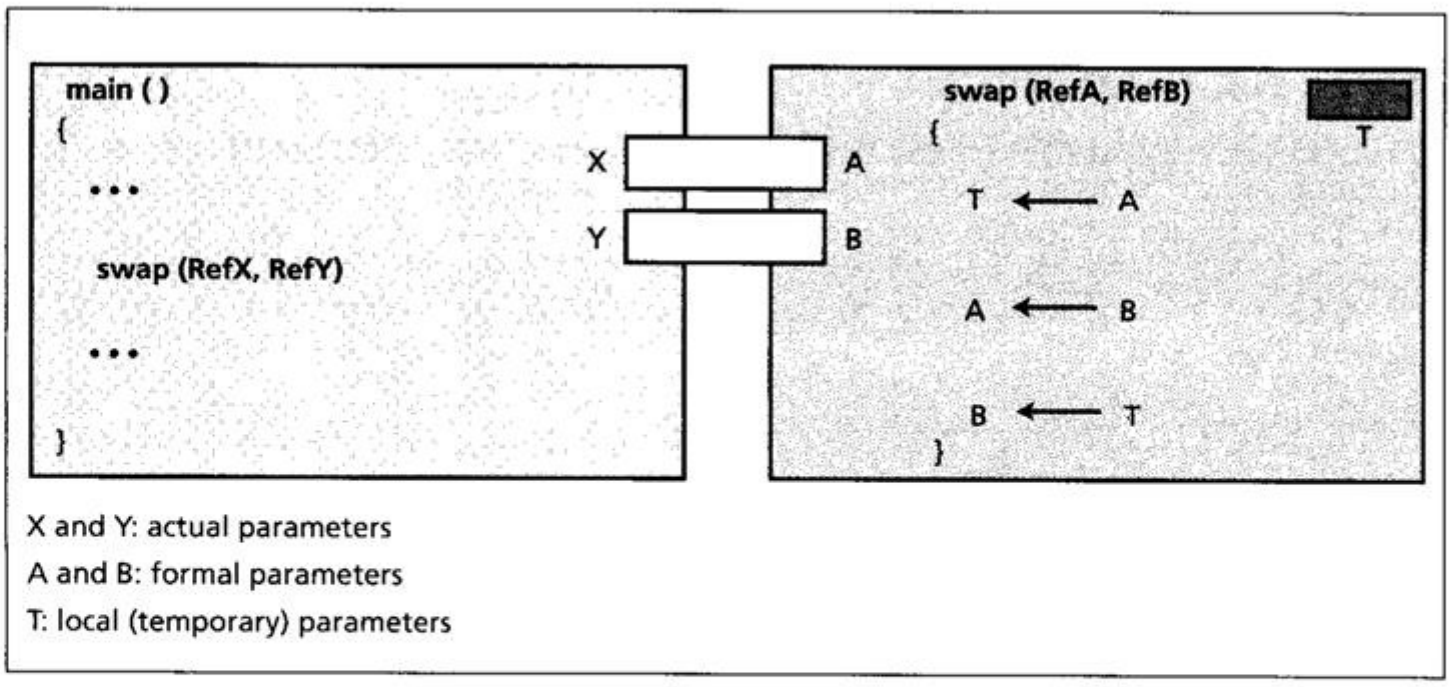


Figure 9.14 An example of pass by reference

Returning values

A subprogram can be designed to return a value or values. This is the way that predefined procedures are designed. When we use the expression C ← A + B, we actually call a procedure *add* (A, B) that return a value to be stored in the variable C.

Implementation

The concept of subprogram is implemented differently in different languages. In C and C++, the subprogram is implemented as a **function**.

9.5 RECOMMENDED READING

For more details about the subjects discussed in this chapter, the following books are recommended:

- Cooke D A: *Concise Introduction to Computer Languages*, Pacific Grove, CA: Brooks/Cole, 2003
- Tucker A and Noonan R: *Programming Languages: Principles and Paradigms*, Burr Ridge, IL: McGraw-Hill, 2002
- Pratt T and Zelkowitz M: *Programming Languages, Design and Implementation*, Englewood Cliffs, NJ: Prentice Hall, 2001
- Sebesta R: *Concepts of Programming Languages*, Boston, MA: Addison Wesley, 2006

9.6 KEY TERMS

This chapter has introduced the following key terms, which are listed here with the pages on wl they first occur:

actual parameter 262
applet 252
assembler 243
assignment statement 259
C language 249
class 251
COmmon Business-Oriented Language (COBOL) 248
compiler 244
compound statement 259
constant 257
data type 255
expression 258
FORmula TRANslation (FORTRAN) 248
functional paradigm 252
identifier 255
inheritance 251
interpretation 245
Java 252
LISt Programming Language (LISP) 253
local variable 262
machine language 242
multithreading 252
object-oriented language 252
operand 259
output 258
pass by reference 263
polymorphism 251
Programming in LOGic (PROLOG)254
semantic analyzer 246
source program 244
subprogram 262
syntax analyzer 246
variable 256

Ada 249
arithmetic operator 258
assembly language 243
bytecode 245
C++ language 252
code generator 246
compilation 244
composite type 256
computer language 242
control statement 260
declarative paradigm 254
formal parameter 262
functional language 253
high-level language 243
imperative paradigm 246
input 258
interpreter 245
lexical analyzer 246
literal 257
logical operator 259
method 250
object program 244
object-oriented paradigm 250
operator 258
Pascal 249
pass by value 263
procedural paradigm 246
relational operator 259
simple type 256
statement 259
syntax 242
token 246

9.7 SUMMARY

- A computer language is a set of predefined words that are combined into a program according to predefined rules, the language's syntax. Over the years, computer languages have evolved from machine language to high-level languages. The only language understood by a computer is machine language.
- High-level languages are portable to many different computers, allowing the programmer to concentrate on the application rather than the intricacies of the computer's organization.
- To run a program on a computer, the program needs to be translated into the computer's native machine language. The program in the high-level language is called the *source program*. The translated program in machine language is called the *object program*. Two methods are used for translation: *compilation* and *interpretation*. A compiler translates the whole source program into the object program. Interpretation refers to the process of translating each line of the source program into the corresponding object program line by line and executing them.
- The translation process uses a lexical analyzer, a syntax analyzer, a semantic analyzer, and a code generator to create a list of tokens.
- A paradigm describes a way in which a computer language can be used to approach a problem to be solved. We divide computer languages into four paradigms: *procedural*, *object-oriented*, *functional*, and *declarative*.
- The procedural paradigm considers a program as an active agent that manipulates passive objects. FORTRAN, COBOL, Pascal, C, and Ada are examples of procedural languages.
- The object-oriented paradigm deals with active objects instead of passive objects. C++ and Java are common object-oriented languages.
- In the functional paradigm, a program is considered as a mathematical function. In this context, a function is a black box that maps a list of inputs to a list of outputs. LISP and Scheme are common functional languages.
- A declarative paradigm uses the principle of logical reasoning to answer queries. One of the best-known declarative languages is PROLOG.
- Some common concepts in procedural and object-oriented languages are *identifiers*, *data types*, *variables*, *literals*, *constants*, *inputs* and *outputs*, *expressions*, and *statements*. Most languages use two categories of control statements: *decision* and *repetition*. *Subprogramming* is a common concept among procedural languages.

9.8 PRACTICE SET

Review questions

1. Distinguish between machine language and assembly language.
2. Distinguish between assembly language and a high-level language.
3. Which computer language is directly related to and understood by a computer?
4. Distinguish between compilation and interpretation.
5. List four steps in programming language translation.
6. List four common computer language paradigms.
7. Compare and contrast a procedural paradigm with an object-oriented paradigm
8. Define a class and a method in an object-oriented language. What is the relation between these two concepts and the concept of an object?
9. Define a functional paradigm.
10. Define a declarative paradigm.

Multiple-choice questions

11. The only language understood by computer hardware is a _____ language.
    a. machine
    b. symbolic
    c. high-level
    d. none of the above
12. C, C++, and Java can be classified as _____ languages.
    a. machine
    b. symbolic
    c. high-level
    d. natural
13. _____ is a program's code in machine language.
    a. A procedure
    b. An object program
    c. A source program
    d. none of the above
14. FORTRAN is a(n) _____ language.
    a. procedural
    b. functional
    c. declarative
    d. object-oriented
15. Pascal is a(n) _____ language.
    a. procedural
    b. functional
    c. declarative
    d. object-oriented
16. C++ is a(n) _____ language.
    a. procedural
    b. functional
    c. declarative
    d. object-oriented
17. LISP is a(n) _____ language.
    a. procedural
    b. functional
    c. declarative
    d. object-oriented
18. _____ is a common language in the business environment.
    a. FORTRAN
    b. C++
    c. C
    d. COBOL
19. _____ is a popular object-oriented language.
    a. FORTRAN
    b. COBOL
    c. C++
    d. LISP
20. A _____ program can be either an application or an applet.
    a. FORTRAN
    b. C++
    c. C
    d. Java
21. LISP and Scheme are both _____ languages.
    a. procedural
    b. functional
    c. declarative
    d. object-oriented
22. Prolog is an example of a(n) _____ language.
    a. procedural
    b. functional
    c. declarative
    d. object-oriented

Exercises

23. Declare three variables of type integer in the C language.
24. Declare three variables of type real in C and initialize them to three values.
25. Declare three constants in C of type character, integer, and real respectively.
26. Explain why a constant must be initialized when it is declared.
27. Find how many times the *statement* in the following code segment in C is executed:

```
A = 5
while (A < 8)
{
        statement;
        A = A + 2;
}
```

28. Find how many times the *statement* in the following code segment in C is executed:

```
A = 5
while (A < 8)
{
        statement;
        A = A – 2;
}
```

29. Find how many times the *statement* in the following code segment in C is executed:

```
for (int i = 5; i < 20, i++)
{
        statement;
        i = i + 1;
}
```

30. Find how many times the *statement* in the following code segment in C is executed:

```
A = 5
do
{
        statement;
        A = A + 1;
} while (A < 10) ;
```

31. Write the code in Exercise 28 using a *do-while* loop.
32. Write the code in Exercise 29 using a *do-while* loop.
33. Write the code in Exercise 29 using a *while* loop.
34. Write the code in Exercise 30 using a *for* loop.
35. Write the code in Exercise 28 using a *for* loop.
36. Write a code fragment using a *while* loop that never executes its body.
37. Write a code fragment using a *do* loop that never executes its body.
38. Write a code fragment using a *for* loop that never executes its body.
39. Write a code fragment using a *while* loop that never stops.
40. Write a code fragment using a *do* loop that never stops.
41. Write a code fragment using a *for* loop that never stops.
42. In the following code, find all the literal values:

```
C = 12 * A + 4 * (B - 5)
```

43. In the following code, find the variables and literals:

```
Hello = "Hello";
```

44. Change the following segment of code to use a *switch* statement:

```
if (A = = 4) statement1;
else
        if (A == 6) statement 2
        else if (A == 8) statement 3
```

45. If the subprogram *calculate (A, B, S, P)* accepts the value of A and B and calculates their sum S and product P, which variable do you pass by value and which one by reference?
46. If the subprogram *smaller(A, B, S)* accepts the value of A and B and finds the smaller of the two, which variable do you pass by value and which one by reference?
47. If the subprogram *cube (A)* accepts the value of A and calculates its cube (A^3), should you pass A to the subprogram by value or by reference?
48. If the subprogram needs to get a value for A from the keyboard and return it to the main program, should you pass A to the subprogram by value or by reference?
49. If the subprogram needs to display the value of A on the monitor, should you pass A to the subprogram by value or by reference?

Software engineering is the establishment and use of sound engineering methods and principles to obtain reliable software. This definition, taken from the first international conference on software engineering in 1969, was proposed thirty years after the first computer was built.

10.1 THE SOFTWARE LIFECYCLE

A fundamental concept in **software engineering** is the **software lifecycle**. Software, like many other products, goes through a cycle of repeating phases (Figure 10.1).

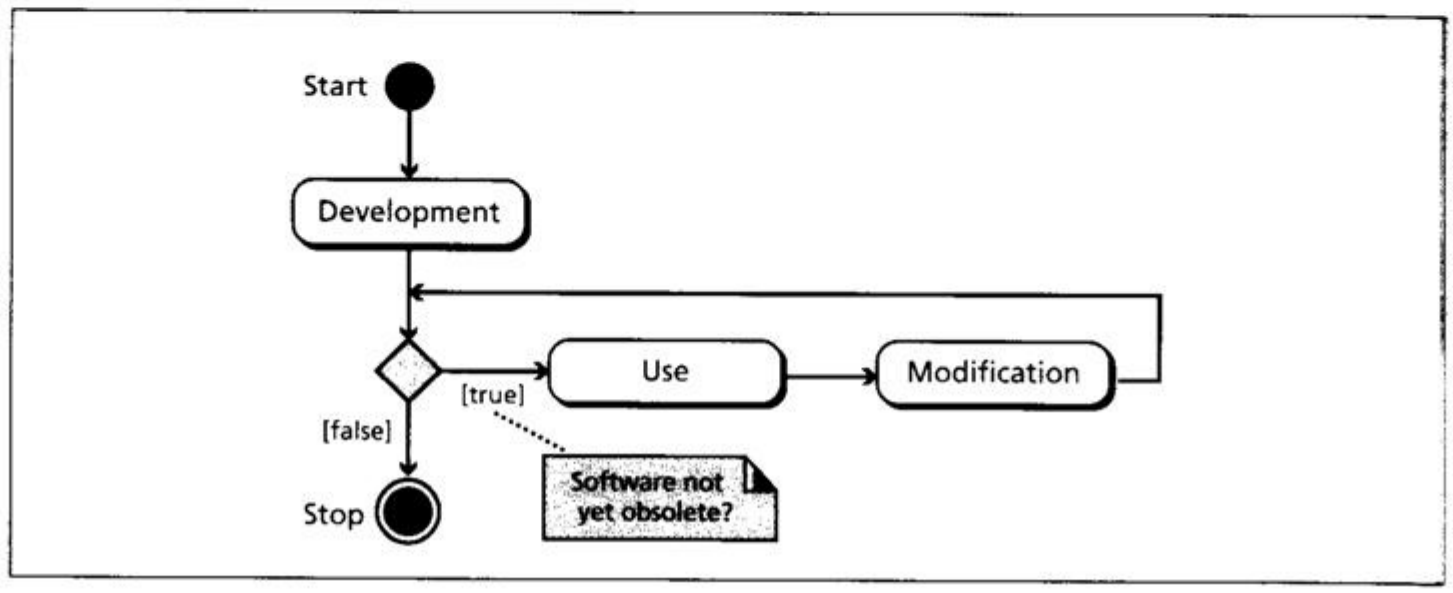


Figure 10.1 The software lifecycle

Software is first developed by a group of developers. Usually it is in use for a while before modifications are necessary. Modification is often needed due to errors found in the software, changes in the rules or laws governing its design, or changes in the company itself. The software therefore needs to be modified before further use. These two steps, *use* and *modify*, continue until the software becomes obsolete. By "obsolete", we mean that the software loses its validity because of inefficiency, obsolescence of the language, major changes in user requirements, or other factors.

Development process models

Although software engineering involves all three processes in Figure 10.1, in this chapter we discuss only the **development process**, which is shown outside the cycle in Figure 10.1. The development process in the software lifecycle involves four phases: analysis, design, implementation, and testing. There are several models for the development process. We discuss the two most common here: the waterfall model and the incremental model.

The waterfall model

One very popular model for the software development process is known as **the waterfall model** (Figure 10.2). In this model, the development process flows in only one direction. This means that a phase cannot be started until the previous phase is

completed. For example, the analysis phase of the whole project should be completed before its design phase is started. The entire design phase should be finished before the implementation phase can be started.

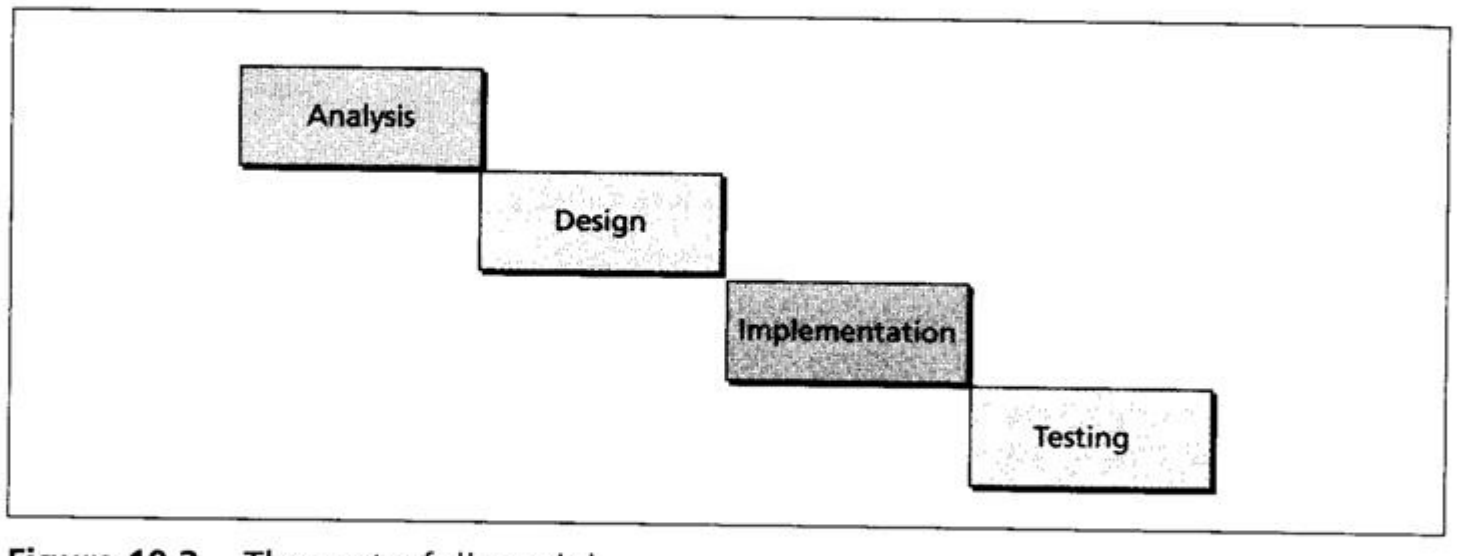


Figure 10.2 The waterfall model

There are advantages and disadvantages to the waterfall model. One advantage is that each phase is completed before the next phase starts. The group that works on the design phase, for example, knows exactly what to do because they have the complete results of the analysis phase. The testing phase can test the whole system because the entire system under development is ready. However, a disadvantage of the waterfall model is the difficulty in locating a problem: if there is a problem in part of the process, the entire process must be checked.

ne cremental odel

In the **incremental model**, software is developed in a series of steps. The developers first complete a simplified version of the whole system. This version represents the entire system but does not include the details. Figure 10.3 shows the incremental model concept.

In the second version, more details are added, while some are left unfinished, and the system is tested again. If there is a problem, the developers know that the problem is with the new functionality, they do not add more functionality until the existing system works properly. This process continues until all required functionality has been added.

10.2 ANALYSIS PHASE

The development process starts with the **analysis phase**. This phase results in a specification document that shows *what* the software will do without specifying *how* it will be done. The analysis phase can use two separate approaches, depending on whether the implementation phase is done using a procedural programming language or an object-oriented language. We briefly discuss both in this section.

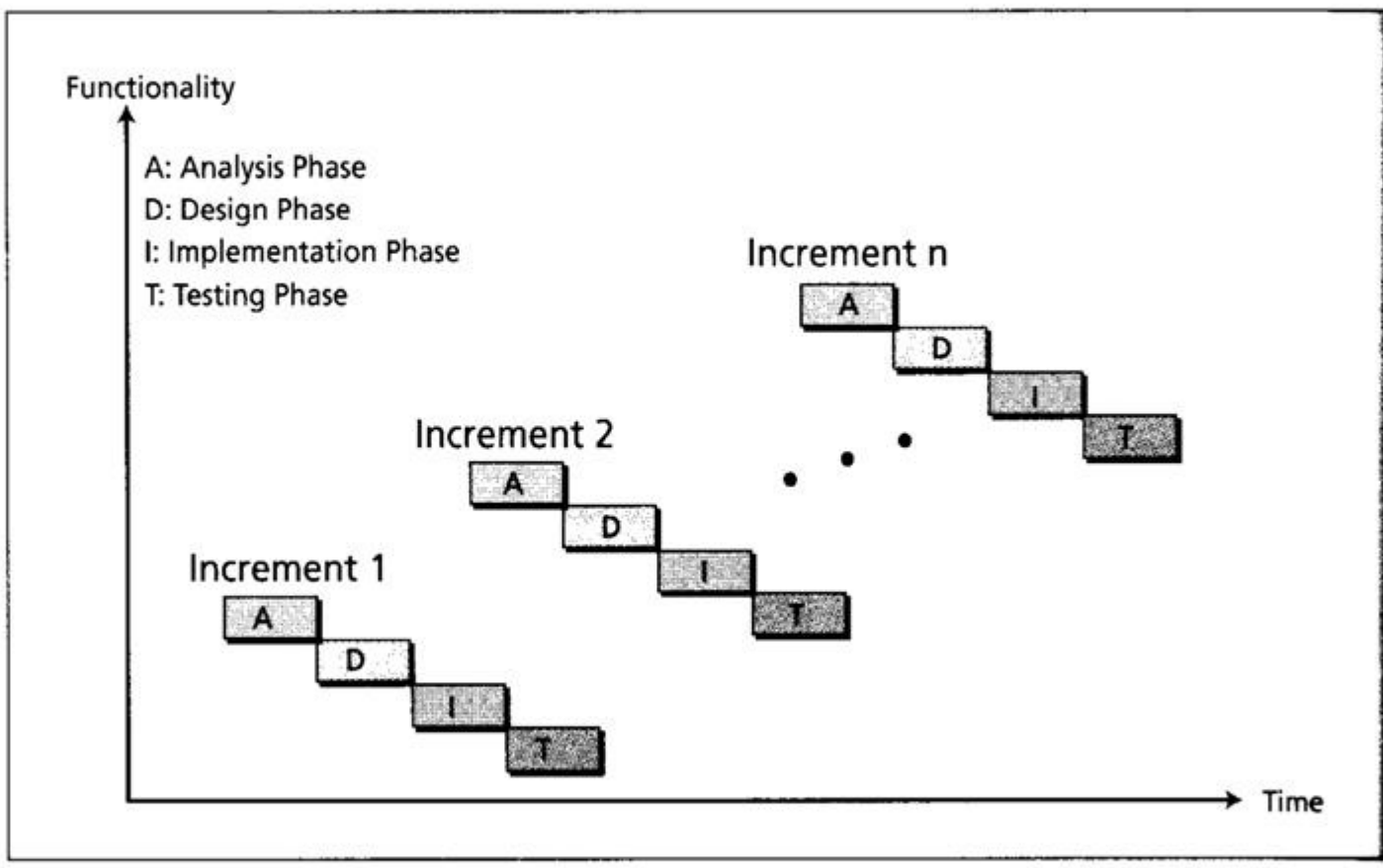


Figure 10.3 The incremental model

Procedure-oriented analysis

Procedure-oriented analysis—also called *structured analysis* or *classical analysis*—is the analysis process used if the system implementation phase will use a procedural language. The specification in this case may use several modeling tools, but we discuss only a few of them here.

Data flow diagrams

Data flow diagrams show the movement of data in the system. They use four symbols: a square box shows the source or destination of data, a rectangle with rounded corners shows the process (the action to be performed on the data), an open-ended rectangle shows where data is stored, and arrows show the flow of data.

Figure 10.4 shows a simplified version of a booking system in a small hotel that accepts reservations from potential guests through the Internet and confirms or denies the reservation based on available vacancies.

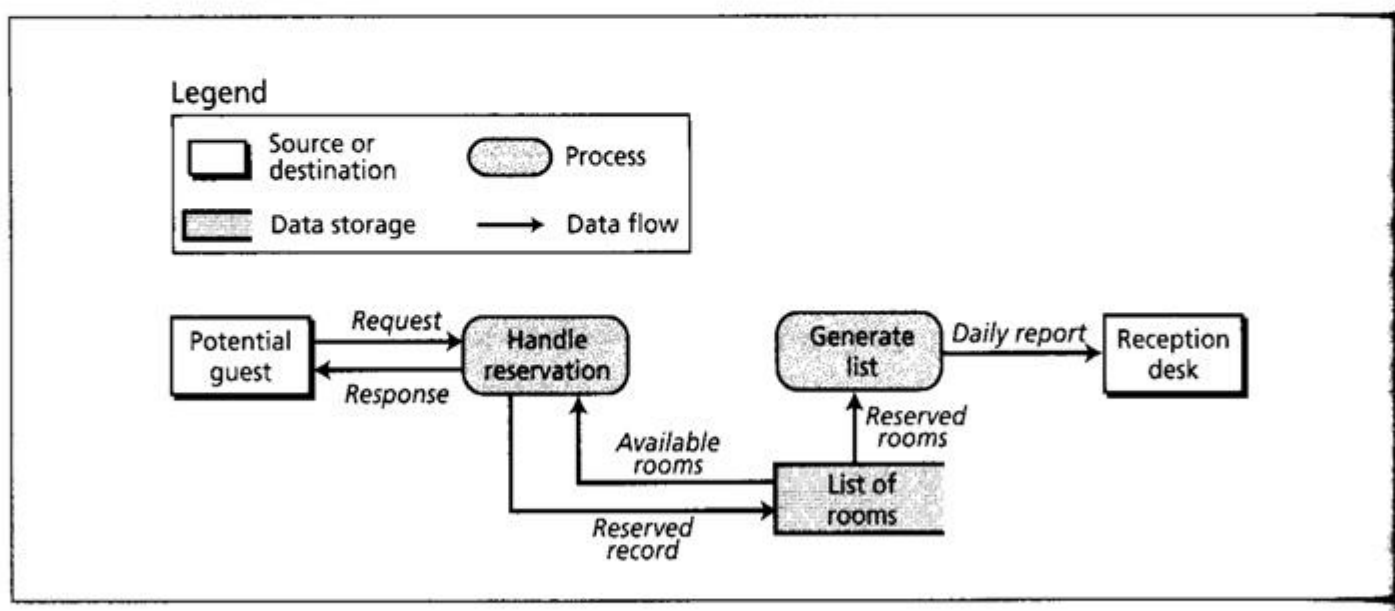


Figure 10.4 An example of a data flow diagram

The only process in this diagram (handle reservation) checks the availability using the reservation file and accepts or rejects a reservation. If the reservation is accepted, it will be recorded in the reservation file.

ity-tionship grams

Another modeling tool used during the analysis phase is the **entity-relationship diagram**. Since this diagram is also used in database design, we discuss it in Chapter 12.

te grams

State diagrams (see Appendix B) provide another useful tool that is normally used when the state of the entities in the system will change in response to events. As an example of a state diagram, we show the operation of a one-passenger elevator. When a floor button is pushed, the elevator moves in the requested direction. It does not respond to any other request until it reaches its destination.

Figure 10.5 shows a state diagram for this old-style elevator. The elevator can be in one of three states: moving up, moving down, or parked. Each of these states is represented by a rounded rectangle in the state diagram. When the elevator is in the parked state, it accepts a request. If the requested floor is the same as the current floor, the request is ignored—the elevator remains in the parked state. If the requested floor is above the current floor, the elevator starts moving up. If the requested floor is lower than the requested floor, the elevator starts moving down. Once moving, the elevator remains in one of the moving states until it reaches the requested floor.

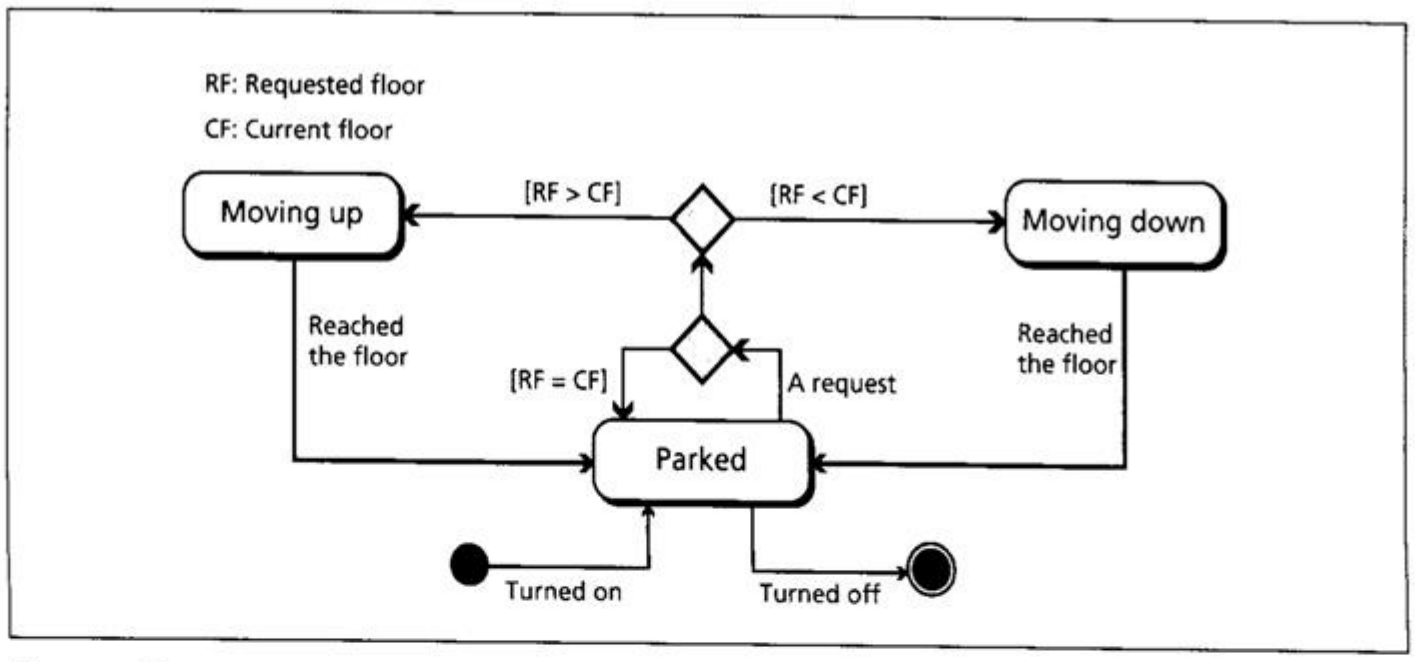


Figure 10.5 An example of a state diagram

Object-oriented analysis

Object-oriented analysis is the analysis process used if the implementation uses an object-oriented language. The specification document in this case may use several tools, but we discuss only a few of them here.

se case iagrams

A **use-case diagram** gives the user's view of a system: it shows how users communicate with the system. A use-case diagram uses four components: system, use cases, actors, and relationships. A system, shown by a rectangle, performs a function. The

actions in the system are shown by use cases, which are denoted by rounded rectangles. An actor is someone or something that uses the system. Although actors are represented by stick figures, they do not necessarily represent human beings.

Figure 10.6 shows the use case diagram for the old-style elevator for which we gave a state diagram in Figure 10.5. The system in this figure is the elevator. The only actor is the user of the elevator. There are two uses cases: pressing the elevator button (in the hall of each floor) and pressing the floor button inside the elevator. The elevator has only one button on each floor that gives the signal to the elevator to move to that floor.

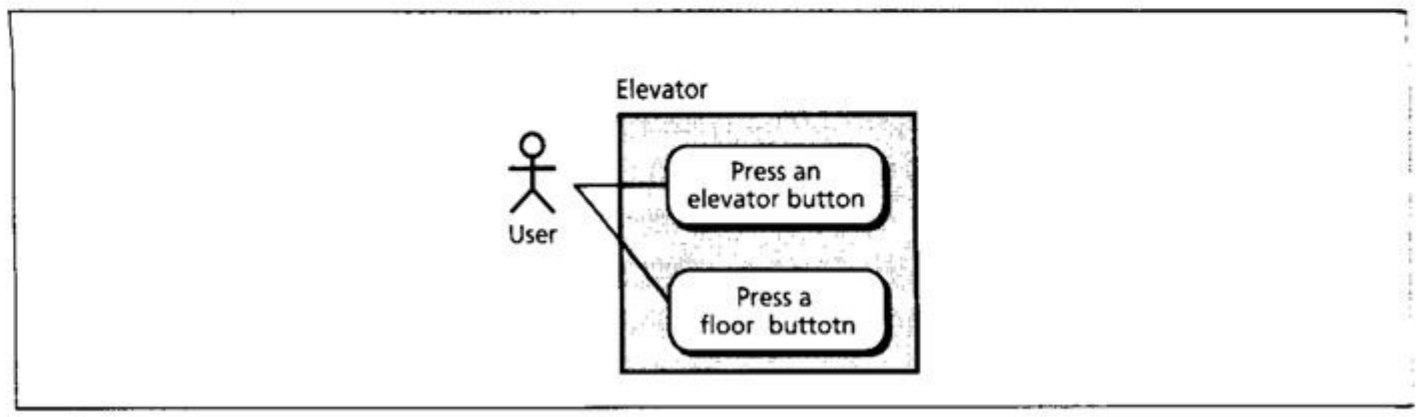


Figure 10.6 An example of use case diagram

Class diagrams

The next step in analysis is to create a **class diagram** for the system. For example, we can create a class diagram for our old-style elevator. To do so, we need to think about the entities involved in the system. In the elevator system we have two classes of entities: the buttons and the elevator itself. At first glance, therefore, it looks as if we have two classes: a button class and an elevator class. However, we have two types of buttons: the elevator buttons in the hallways and the floor buttons inside the elevator. It seems then that we can have a button class and two classes that inherit from the button class: an elevator button class and a floor button class. The first class diagram that we can create for the elevator problem is therefore that shown in Figure 10.7.

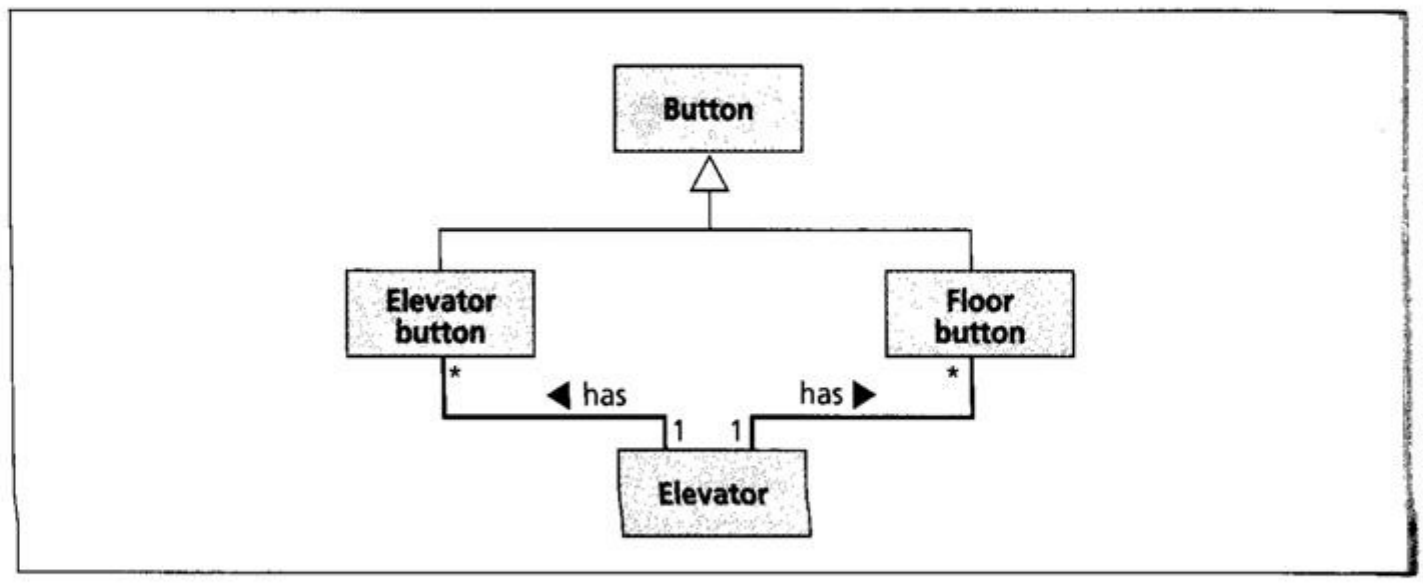


Figure 10.7 An example of a class diagram

Note that the elevator button class and the floor button class are subclasses of the button class. However, the relationship between the elevator class and the two button classes (elevator button and floor button) is a one-to-many relation (see Appendix B). The class diagram for the elevator system can of course be extended, but we leave this to books on software engineering.

tate chart

After the class diagram is finalized, a **state chart** can be prepared for each class in the class diagram. A state chart in object-oriented analysis plays the same role as the state diagram in procedure-oriented analysis. This means that for the class diagram of Figure 10.7, we need to have a four-state chart.

10.3 DESIGN PHASE

The **design phase** defines *how* the system will accomplish *what* was defined in the analysis phase. In the design phase, all components of the system are defined.

Procedure-oriented design

In **procedure-oriented design** we have both procedures and data to design. We discuss a category of design methods that concentrate on procedures. In procedure-oriented design, the whole system is divided into a set of procedures or modules.

ructure harts

A common tool for illustrating the relations between modules in procedure-oriented design is a **structure chart**. For example, the elevator system whose state diagram is shown in Figure 10.5 can be designed as a set of modules shown in the structure chart in Figure 10.8. Structure charts are discussed in Appendix D.

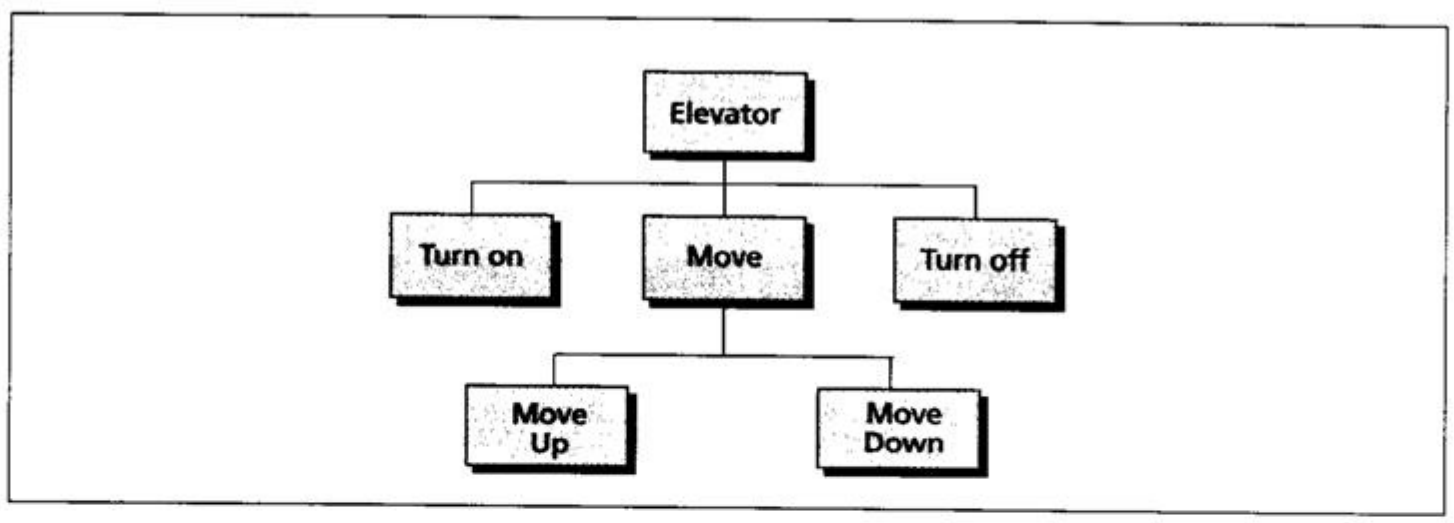


Figure 10.8 A structure chart

odularity

Modularity means breaking a large project into smaller parts that can be understood and handled easily. In other words, modularity means dividing a large task into small tasks that can communicate with each other. The structure chart discussed in the previous section shows the modularity in the elevator system. There are two main concerns when a system is divided into modules: *coupling* and *cohesion*.

Coupling

Coupling is a measure of how tightly two modules are bound to each other. The more tightly coupled, the less independent they are. Since the objective is to make modules as independent as possible, we want them to be loosely coupled. There are at least three reasons why loose coupling is desirable:

- Loosely coupled modules are more likely to be reusable.
- Loosely coupled modules are less likely to create errors in related modules.
- When the system needs to be modified, loosely coupled modules allow us to modify only modules that need to be changed without affecting modules that do not need to change.

Coupling between modules in a software system must be minimized.

Cohesion

Another issue in modularity is cohesion. **Cohesion** is a measure of how closely the modules in a system are related. We need to have maximum possible cohesion between modules in a software system.

Cohesion between modules in a software system must be maximized.

Object-oriented design

In **object-oriented design** the design phase continues by elaborating the details of classes. As we mentioned in Chapter 9, a class is made of a set of variables (attributes) and a set of methods. The object-oriented design phase lists details of these attributes and methods. Figure 10.9 shows an example of the details of our four classes used in the design of the old-style elevator.

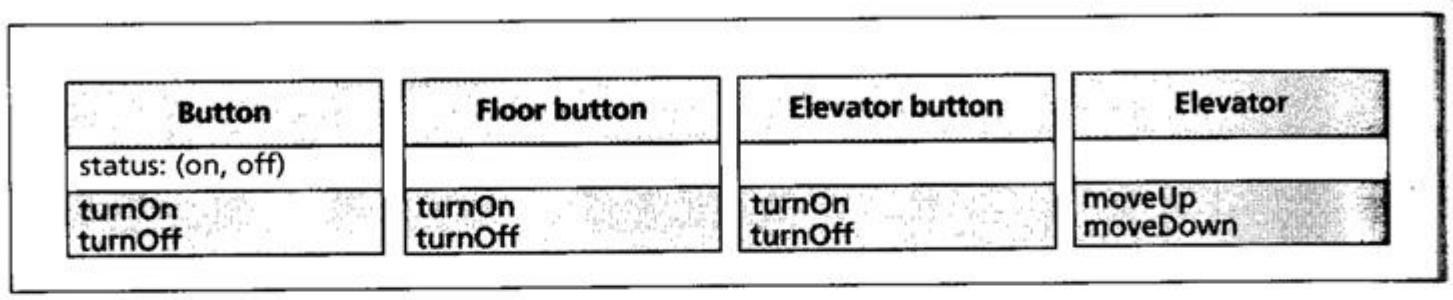


Figure 10.9 An example of classes with attributes and methods

10.4 IMPLEMENTATION PHASE

In the waterfall model, after the design phase is completed, the **implementation phase** can start. In this phase the programmers write the code for the modules in procedure-oriented design, or write the program units to implement classes in object-oriented design. There are several issues we need to mention in each case.

Choice of language

In a procedure-oriented development, the project team needs to choose a language or a set of languages from among the procedural languages discussed in Chapter 10. Although some languages like C++ are considered to be both a procedural and an object-oriented language, normally an implementation uses a purely procedural language such as C. In the object-oriented case, both C++ and Java are common.

Software quality

The quality of software created at the implementation phase is a very important issue. A software system of high quality is one that satisfies the user's requirements, meets the operating standards of the organization, and runs efficiently on the hardware for which it was developed. However, if we want to achieve a software system of high quality, we must be able to define some attributes of quality.

Software quality factors

Software quality can be divided into three broad measures: *operability, maintainability,* and *transferability.* Each of these measures can be further broken down as shown in Figure 10.10.

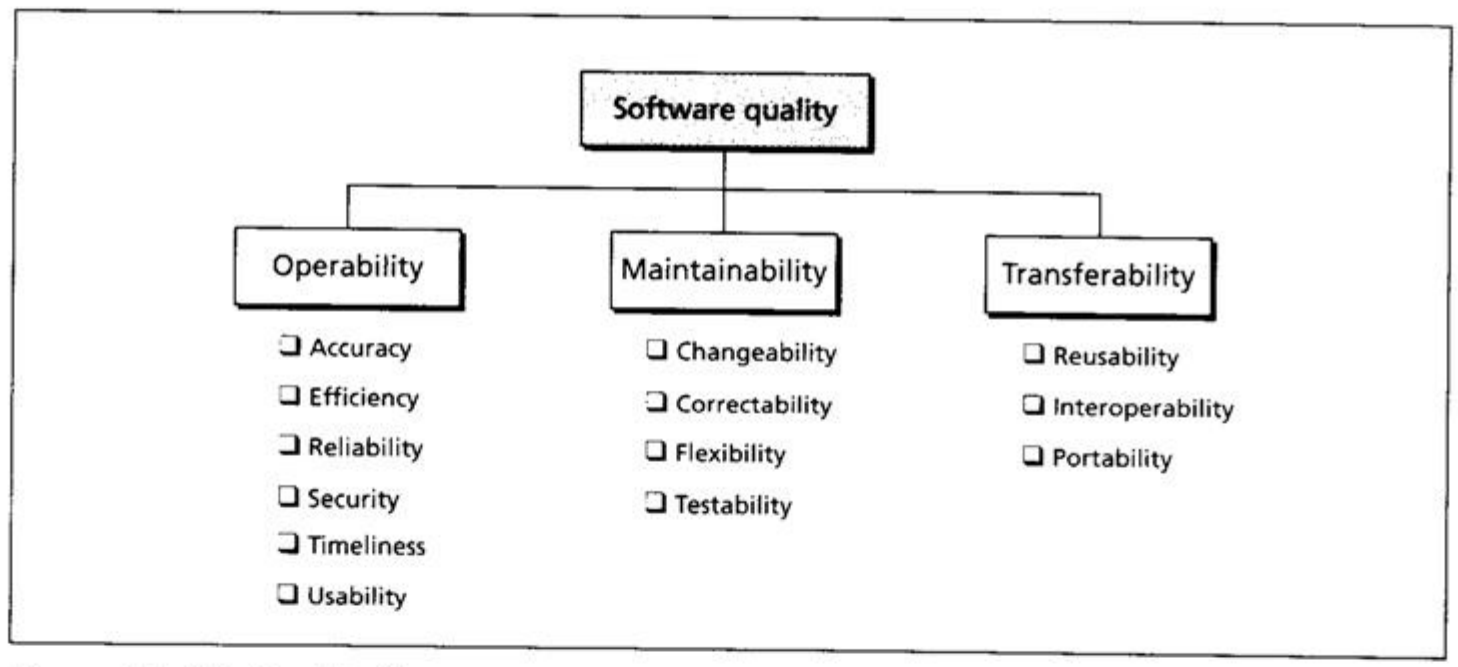


Figure 10.10 Quality factors

Operability

Operability refers to the basic operation of a system. Several measures can be mentioned for operability, as shown in Figure 10.10: *accuracy, efficiency, reliability, security, timeliness*, and *usability.*

- A system that is not *accurate* is worse than no system at all. Any system that is developed, therefore, must be thoroughly tested both by a system's test engineer and the user. *Accuracy* can be measured by such metrics as mean time between failures, number of bugs per thousand lines of code, and number of user requests for change.

- *Efficiency* is a subjective term. In some cases, the user will specify a performance standard, such as a real-time response that must be received within 1 second 95 percent of the time. This is certainly measurable.
- *Reliability* is really the sum of the other factors. If users count on the system to get their job done and are confident in it, then it is most likely reliable. On the other hand, some measures speak directly to a system's reliability, most notably, mean time between failures.
- How *secure* a system is refers to how easy it is for unauthorized people to access the system's data. Although this is a subjective area, there are checklists that assist in assessing the system's security. For example, does the system have and require passwords to identify users?
- *Timeliness* in software engineering can mean several different things. Does the system deliver its output in a timely fashion? For online systems, does the response time satisfy the users' requirements?
- *Usability* is another area that is highly subjective. The best measure of usability is to watch the users and see how they are using the system. User interviews will often reveal problems with the usability of a system.

Maintainability

Maintainability refers to the ease with which a system can be kept up to date and running correctly. Many systems require regular changes, not because they were poorly implemented, but because of changes in external factors. For example, the payroll system for a company might have to be changed often to meet changes in government laws and regulations.

- *Changeability* is a subjective factor. Experienced project leaders, however, are able to estimate how long a requested change will take to implement. If too long, it may indicate that the system is difficult to change. This is especially true of older systems. There are software measurement tools in the field today that will estimate a program's complexity and structure.
- One measure of *correctability* is *mean time to recovery*, which is the time it takes to get a program back into operation after it fails. Although this is a reactive definition, there are currently no predictors of how long it will take to correct a program when it fails.
- Users are constantly requesting changes to systems. *Flexibility* is a qualitative attribute that attempts to measure how easy it is to make these changes. If a program needs to be completely rewritten to effect a change, it is not flexible.
- We might think that *testability* is a highly subjective area, but test engineers have checklists of factors that can assess a program's testability.

Transferability

Transferability refers to the ability to move data and/or a system from one platform to another and to reuse code. In many situations this is not an important factor. On the other hand, if we are writing generalized software, it can be critical.

- If modules are written so that they can be reused in other systems, then they have a high level of *reusability*. Good programmers build libraries of functions that they can reuse for solving similar problems.
- *Interoperability* is the capability of sending data to other systems. In today's highly integrated systems, it is a desirable attribute. In fact, it has become so important that operating systems now support the ability to move data between systems, such as between a word processor and a spreadsheet.
- *Portability* is the ability to move software from one hardware platform to another.

10.5 TESTING PHASE

The goal of the **testing phase** is to find errors, which means that a good testing strategy is the one that finds most errors. There are two types of testing: *glass-box* and *black-box* (Figure 10.11).

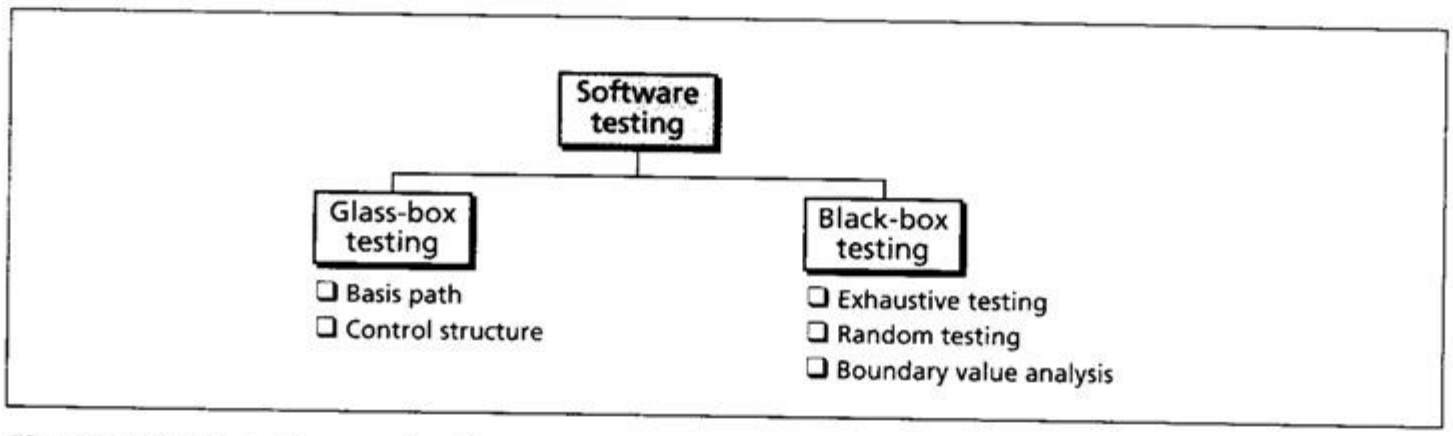


Figure 10.11 Software testing

Glass-box testing

Glass-box testing (or **white-box testing)** is based on knowing the internal structure of the software. The testing goal is to check to determine whether all components of the software do what they are designed to. Glass-box testing assumes that the tester knows everything about the software. In this case, the software is like a glass box in which everything inside the box is visible. Glass-box testing is done by the software engineer or a dedicated team. Glass-box testing that uses the structure of the software is required to guarantee that at least the following four criteria are met:

- All independent paths in every module are tested at least once.
- All the decision constructs (two-way and multiway) are tested on each branch.
- Each loop construct is tested.
- All data structures are tested.

Several testing methodologies have been designed in the past. We briefly discuss two of them: *basis path* testing and *control structure* testing.

Basis path testing

Basis path testing was proposed by Tom McCabe. This method creates a set of test cases that executes *every statement* in the software at least once.

Basis path testing is a method in which each statement in the software is executed at least once.

Basis path testing uses *graph theory* (see Chapter 12) and *cyclomatic complexity* to find the independent paths that must be followed to guarantee that each statement is executed at least once.

Example 10.1

To give the idea of basis path testing and finding the independent paths in part of a program, assume that a system is made up of only one program and that the program is only a single loop with the UML diagram shown in Figure 10.12.

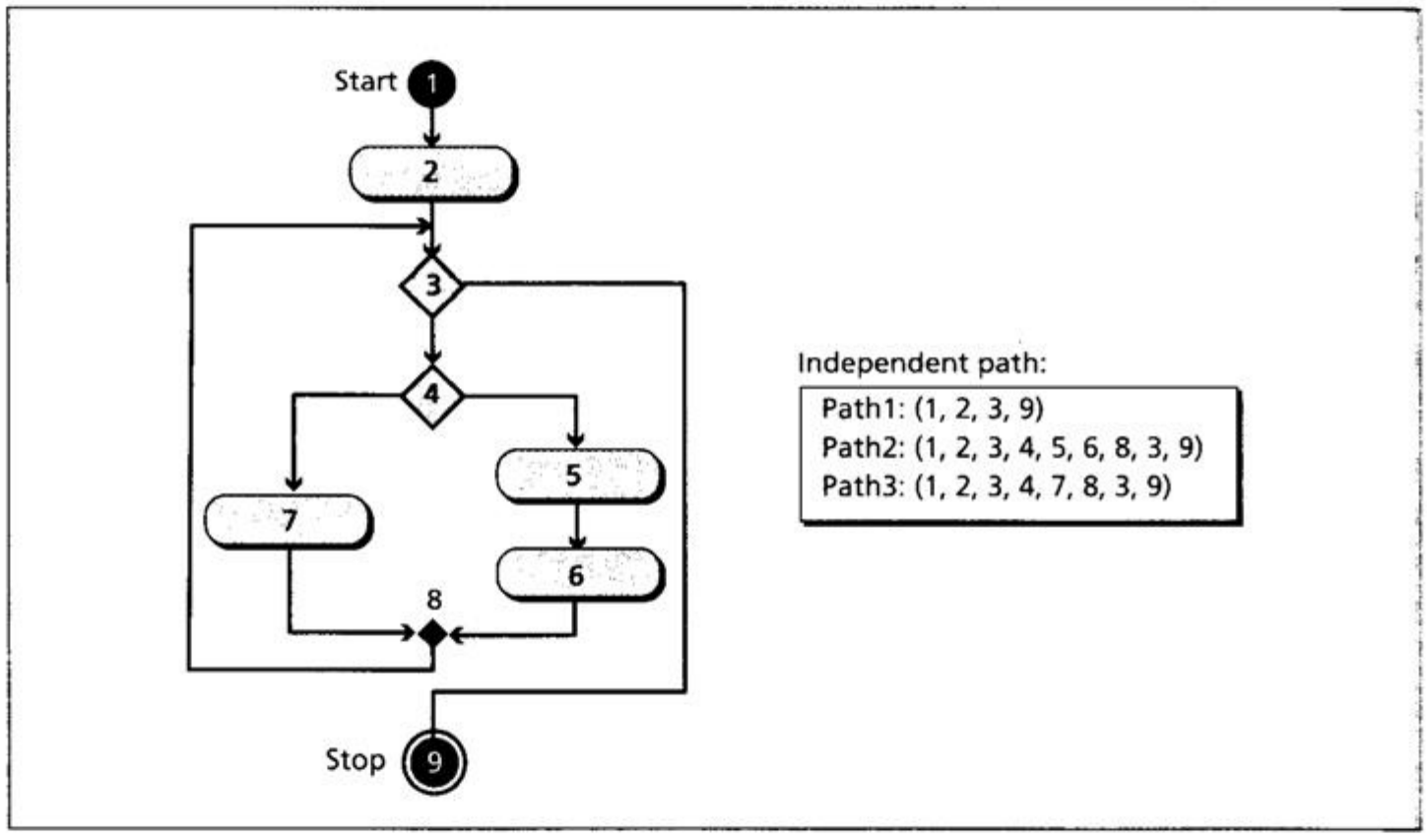


Figure 10.12 An example of basis path testing

In this simple program we have three independent paths. The first path is the case in which the loop is bypassed. The second path is where the loop is executed once through the right branch of the decision construct. The third path is when the loop is executed once but through the left branch of the decision construct. If there are more iterations, the paths created are not independent of these three paths: it is possible to prove this if we change the UML to a flow graph, but we must leave the proof to books on software engineering. The idea is to design test cases that cover all three paths in the basis path set, so that all statements are executed at least once.

Control structure testing is more comprehensive than basis path testing and includes it. This method uses different categories of tests that are briefly described below.

Condition testing

Condition testing applies to any condition expression in the module. A *simple condition* is a relational expression, while a *compound condition* is a combination of simple conditions and logical operators (see Chapter 9). Condition testing is designed to check whether all conditions are set correctly.

Data flow testing

Data flow testing is based on the flow of data through the module. This type of testing selects test cases that involve checking the value of variables when they are used on the left side of the assignment statement.

Loop testing

Loop testing uses test cases to check the validity of loops. All types of loops (*while*, *do*, and *for*) are carefully tested.

Black-box testing

Black box testing gets its name from the concept of testing software without knowing what is inside it and without knowing how it works. In other words, the software is like a black box into which the tester cannot see. Black-box testing tests the functionality of the software in terms of what the software is supposed to accomplish, such as its inputs and outputs. Several methods are used in black-box testing, discussed below.

ustive ing

The best black-box test method is to test the software for all possible values in the input domain. However, in complex software the input domain is so huge that it is often impractical to do so.

dom ing

In random testing, a subset of values in the input domain is selected for testing. It is very important that the subset be chosen in such a way that the values are distributed over the domain input. The use of random number generators can be very helpful in this case.

ndary- e testing

Errors often happen when boundary values are encountered. For example, if a module defines that one of its inputs must be greater than or equal to 100, it is very important that module be tested for the boundary value 100. If the module fails at this boundary value, it is possible that some condition in the module's code such as $x \geq 100$ is written as $x > 100$.

10.6 DOCUMENTATION

For software to be used properly and maintained efficiently, **documentation** is needed. Usually, three separate sets of documentation are prepared for software:

user documentation, system documentation, and technical documentation. However, note that documentation is an ongoing process. If the software has problems after release, they must be documented too. If the software is modified, all modifications and their relationship to the original package must also be documented. Documentation only stops when the package becomes obsolete.

Documentation is an ongoing process.

User documentation

To run the software system properly, the users need documentation, traditionally called a *user guide*, that shows how to use the software step by step. User guides usually contains a tutorial section to guide the user through each feature of the software.

A good user guide can be a very powerful marketing tool: the importance of user documentation in marketing cannot be overemphasized. User guides should be written for both the novice and the expert users, and a software system with good user documentation will definitely increase sales.

System documentation

System documentation defines the software itself. It should be written so that the software can be maintained and modified by people other than the original developers. System documentation should exist for all four phases of system development.

In the analysis phase, the information collected should be carefully documented. In addition, the analysts should define the sources of information. The requirements and methods chosen in this phase must be clearly stated with the rationale behind them.

In the design phase, the tools used in the final copy must be documented. For example, if a chart undergoes several changes, the final copy of the chart should be documented with complete explanations.

In the implementation phase, every module of the code should be documented. In addition, the code should be self-documenting as far as possible using comments and descriptive headers.

Finally, the developers must carefully document the testing phase. Each type of test applied to the final product should be mentioned along with its result. Even unfavorable results and the data that produced them must be documented.

Technical documentation

Technical documentation describes the installation and the servicing of the software system. Installation documentation defines how the software should be installed on each computer, for example, servers and clients. Service documentation defines how the system should be maintained and updated if necessary.

10.9 SUMMARY

- The software lifecycle is a fundamental concept in software engineering. Software, like many other products, goes through a cycle of repeating phases.
- The development process in the software lifecycle involves four phases: analysis, design, implementation, and testing. Several models have been used in relation to these phases. We discussed the two most common: the waterfall model and the incremental model.
- The development process starts with the analysis phase. The analyst prepares a specification document that shows *what* the software will do without specifying *how* it will be done. The analysis phase can be done in two ways: procedure-oriented analysis and object-oriented analysis.
- The design phase defines *how* the system will accomplish what was defined in the analysis phase. In procedure-oriented design, the whole project is divided into a set of procedure or modules. In object-oriented design, the design phase continues by elaborating the details of classes.
- Modularity means breaking a large project into smaller parts that can be understood and handled easily. Two issues are important when a system is divided into modules: coupling and cohesion. Coupling is a measure of how tightly two modules are bound to each other. Coupling between modules in a software system must be minimized. Cohesion is a measure of how closely the modules in a system are related. Cohesion between modules in a software system should be maximized.
- In the implementation phase, programmers write the code for the modules in procedure-oriented design, or write the program units to implement classes in the object-oriented design.
- The quality of software is important. Software quality can be divided into three broad measures: operability, maintainability, and transferability.
- The goal of the testing phase is to find errors. There are two types of testing: glass-box and black-box. Glass-box testing (or white-box testing) is based on knowing the internal structure of the software. Glass-box testing assumes that the tester knows everything. Black box testing means testing the software without knowing what is inside it and without knowing how it works.

10.10 PRACTICE SET

Review questions

1. Define "software lifecycle".
2. Distinguish between the waterfall model and the incremental development model.
3. List the four phases in the development process.
4. Define the purpose of the analysis phase and describe two trends in this phase.
5. Define the purpose of the design phase and describe two trends in this phase.
6. Describe modularity and mention two issues related to modularity.
7. Distinguish between coupling and cohesion.
8. Define the purpose of the implementation phase and describe the issue of quality in this phase.
9. Define the purpose of the testing phase and list two categories of testing.
10. Distinguish between glass-box testing and black-box testing.

Multiple-choice questions

11. One phase in system development is _____.
    a. analysis
    b. testing
    c. design
    d. all of the above
12. Defining the users, requirements, and methods is part of the _____ phase.
    a. analysis
    b. design

c. implementation
d. testing

In the system development process, writing the program is part of the _____ phase.
a. analysis
b. design
c. implementation
d. testing

In the system development process, structure charts are tools used in the _____ phase.
a. analysis
b. design
c. implementation
d. testing

Testing a software system can involve _____ testing.
a. black-box
b. glass-box
c. neither a nor b
d. a and b

_____ is the breaking up of a large project into smaller parts.
a. Coupling
b. Incrementing
c. Obsolescence
d. Modularization

_____ is a measure of how tightly two modules are bound to each other.
a. Modularity
b. Coupling
c. Interoperability
d. Cohesion

_____ is a measure of how closely the processes in a program are related.
a. Modularity
b. Coupling
c. Interoperability
d. none of the above

_____between modules in a software system must be minimized.
a. Coupling
b. Cohesion
c. neither a nor b
d. a and b

20. _____between modules in a software system must be maximized.
a. Coupling
b. Cohesion
c. neither a nor b
d. a and b

Exercises

21. In Chapter 9 we explained that the use of constant values are preferred to literals. What it the effect of this preference on the software lifecycle?
22. In Chapter 9 we showed that communication between two modules can take place either by pass-by-value or pass-by-reference. Which method provides less coupling between the two modules?
23. In Chapter 9 we showed that communication between two modules can take place either by pass-by-value or pass-by-reference. Which method provides more cohesion between the two modules?
24. Draw a use case diagram for a simple library.
25. Draw a use case diagram for a small grocery store.
26. Show the data flow diagram for a simple mathematical formula $x + y$.
27. Show the data flow diagram for a simple mathematical formula $x \times y + z \times t$.
28. Show the data flow diagram for a library.
29. Show the data flow diagram for a small grocery store.
30. Create a structure chart for Exercise 28.
31. Create a structure chart for Exercise 29.
32. Show a state diagram for a stack of fixed capacity (see Chapter 12).
33. Show a state diagram for a queue of fixed capacity (see Chapter 12).
34. Create a class diagram for a library.
35. Create a class diagram for a small grocery store.
36. Show the details of classes in Exercise 34.
37. Show the details of classes in Exercise 35.
38. The input data to a program is made up of a combination of three integers in the range 1000 to 1999 (inclusive). Find the number of exhaustive tests to test all combinations of these numbers.

39. List the boundary-value tests required for Exercise 38.

40. A random number generator creates a number between 0 and 0.999. How can this random number generator be used to do random testing for the system described in Exercise 38?

"An excellent textbook, well written and with a wide range of interesting material."

– Prof. S.J. Maybank, Birkbeck College, London

Based on the ACM model curriculum guidelines, this easy-to-read and easy-to-navigate text covers all the fundamentals of computer science required for first year students embarking on a computing degree. Divided into five parts – computer and data, computer hardware, computer software, data organization and with an introduction to some of the more advanced topics – *Foundations of Computer Science* gives students a bird's eye view of the subject. Each chapter includes key terms, summaries, review questions, multiple-choice questions, and exercises to enhance learning, while introducing tools such as UML, structure charts and pseudocode, which students will need in order to succeed in later courses. For students and instructors, there are more figures, examples, exercises, as well as selected solutions and a test bank.

Updated to cover the latest technologies and changes to course requirements, this second edition features new and updated coverage of:

- **Artificial Intelligence**
- **Computer Ethics and Crimes**
- **Networking,**
- **LINUX,**
- **Security**

Behrouz Forouzan worked in industry for more than ten years before switching to teach computer science courses. In the twenty years since then, he has written eleven textbooks on the subject. **Firouz Mosharraf** is Professor of Engineering at Rio Hondo College in California, USA.

www.thomsonlearning.co.uk/forouzan

Thomson Learning is dedicated to providing innovative approaches to lifelong learning

THOMSON

For your learning solutions:
www.thomsonlearning.co.uk

ISBN: 978-1-84480-700-0

9 781844 807000 >