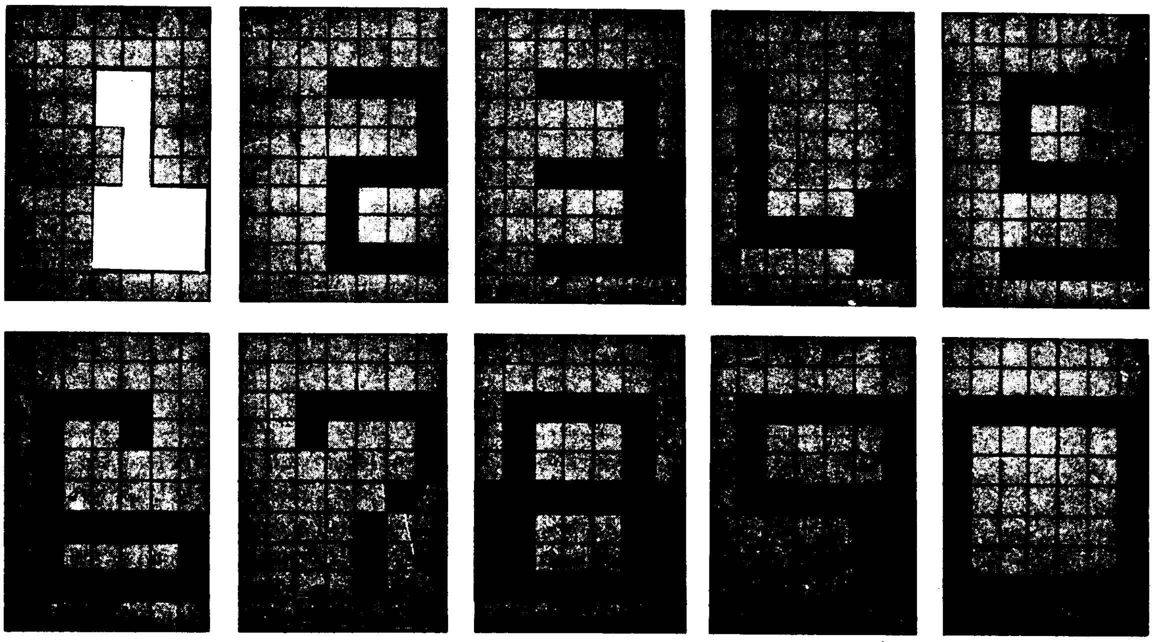


COMPUTER OPERATION

The computer industry continues to grow at record speeds. Computer shipments exceeded $25 billion in 1983, and the rapid growth in personal computer sales will push this number still higher each year.

Well over a million people now work in the computer industry, and this does not encompass the millions who work with computers indirectly, including bank clerks who put all their transactions into computers, airlines and motel employees who work with computers to make reservations, and machinists who use computer-controlled power tools.

In fact, computers now route long-distance telephone calls, process and issue the checks in banks, schedule planes and trains, make weather forecasts, predict and process the elections, and figure in so many things that entire books will be (and have been) written just documenting the types of applications.

Computers now use the major share of the electronics components being manufactured, and this share will continue to rise. The need for computer personnel in all areas continues to grow: over a quarter of a million new programmers are needed each year; the federal government's Department of Labor continues to maintain business machine service personnel and electronic computer operating personnel in first and second place in a list of five fastest-growing employment areas.

OBJECTIVES

1 This chapter first presents some background and historical information on the development of computer science. Early calculators and the first digital computer projects are described briefly.

1 COMPUTER OPERATION

2 How a computer is used to solve a scientific problem or how an office procedure might be organized for computer usage is discussed.

3 An overview of several general categories of computer systems is presented including interactive systems, batch processing, and timeshared systems.

4 The general breakdown of a computer into five sections is discussed: these are input, control, memory, the arithmetic-logic unit, and output.

5 Programming and programming languages are defined and some basic concepts in this area introduced. A brief introduction to assembly language is followed by a discussion of a higher-level language, Pascal.

ELECTRONIC DIGITAL COMPUTERS

1.1 The history of attempts to make machines that could perform long sequences of calculations automatically is fairly long. The best known early attempt was made in the 19th century by Charles Babbage, an English scientist and mathematician. Babbage attempted to mechanize sequences of calculations, eliminating the operator and designing a machine so that it would perform all the necessary operations in a predetermined sequence. The machine designed by Babbage used cardboard cards with holes punched in them to introduce both instructions and the necessary data (numbers) into the machine. The machine was to perform the instructions dictated by the cards automatically, not stopping until an entire sequence of instructions had been completed. The punched cards used to control the machine already had been used to control the operation of weaving machines. Surprisingly enough, Babbage obtained some money for his project from the English government and started construction. Although he was severely limited by the technology of his time and the machine was never completed, Babbage succeeded in establishing the basic principles on which modern computers are constructed. There is even some speculation that if he had not run short of money, he might have constructed a successful machine. Although Babbage died without realizing his dream, he had established the fundamental concepts which were used to construct machines elaborate beyond even his expectations.

By the 1930s, punched cards were in wide use in large businesses, and various types of punched-card-handling machines were available. In 1937 Howard Aiken, at Harvard, proposed to IBM that a machine could be constructed (by using some of the parts and techniques from the punched-card machines) which would automatically sequence the operations and calculations performed. This machine used a combination of electromechanical devices, including many relays. The machine was in operation for some time, generating many tables of mathematical functions (particularly Bessel functions), and was used for trajectory calculations in World War II.

Aiken's machine was remarkable for its time, but was limited in speed by its use both of relays rather than electronic devices and of punched cards for sequencing the operations. In 1943 S. P. Eckert and J. W. Mauchly, of the Moore School of Engineering of the University of Pennsylvania, started the Eniac, which used electronic components (primarily vacuum tubes) and therefore was faster, but which also used switches and a wired plug board to implement the programming of operations. Later Eckert and Mauchly built the Edvac, which had its program stored in the computer's memory, not depending on external sequencing. This was an important innovation, and a computer that stores its list of operations, or program, internally is called a *stored-program computer*. Actually the Edsac, at the University of Manchester, started later but completed before Edvac, was the first operational stored-program computer.

A year or so later, John Von Neumann, at the Institute for Advanced Study (IAS) in Princeton, started the IAS in conjunction with the Moore School of Engineering, and this machine incorporated most of the general concepts of parallel binary stored-program computers.

The Univac I was the first commercially available electronic digital computer, and it was designed by Eckert and Mauchly at their own company, which was later bought by Sperry Rand. The U.S. Board of the Census bought the first Univac. (Later Univac and half of Aiken's machine were placed in the Smithsonian Institution, where they may now be seen.) IBM entered the competition with the IBM 701, a large machine, in 1953 and in 1954 with the IBM 650, a much smaller machine which was very successful. The IBM 701 was the forerunner of the 704–709–7094 series of IBM machines, the first "big winners" in the large-machine category.

Quite a few vacuum-tube electronic computers were available and in use by the late 1950s, but at this time an important innovation in electronics appeared—the transistor. The replacement of large, expensive (hot) vacuum tubes with small, inexpensive, reliable, comparatively low heat-dissipating transistors led to what are called *second-generation computers*. The size and importance of the computer industry grew at amazing rates, while the costs of individual computers dropped substantially.

By 1965 a third generation of computers was introduced. (The IBM Corporation, in introducing the 360 series, used the term *third-generation* as a key phrase in their advertising, and it remains a catchword in describing all machines of this era.) The machines of this period began making heavy use of *integrated circuits* in which many transistors and other components are fabricated and packaged together in a single small container. The low prices and high packing densities of these circuits plus lessons learned from prior machines led to some differences in computer system design, and these machines proliferated and expanded the computer industry to its present multibillion-dollar size.

Present-day computers are less easily distinguished from earlier generations. There are some striking and important differences, however. The manufacture of integrated circuits has become so advanced as to incorporate hundreds of thousands of active components in volumes of a fraction of an inch, leading to what is called *large-scale integration* (LSI) and *very large-scale integration* (VLSI). This has led to small-size, lower-cost, large-memory, ultrafast computers.

APPLICATION OF COMPUTERS TO PROBLEMS

1.2 For many years large office forces have been employed in the accounting departments of business firms. The clerks employed by these businesses spend most of their time performing arithmetic computations and then entering results into company books and on paychecks, invoices, order forms, etc. Most of the arithmetic consists of repetitious sequences of simple calculations which the clerks perform over and over on different sets of figures. Few decisions are required, since rules usually have been defined covering almost all problems that might arise.

A typical task in a payroll office is the processing of paychecks for company employees who work at an hourly rate.[1] This job involves calculating total earnings by multiplying each employee's hourly wage rate by the number of hours worked, taking into consideration any overtime; figuring and then deducting taxes, insurance, contributions to charity, etc.; then making out the necessary check and entering a record of all figures. Figure 1.1 is a flowchart of a possible procedure. Flowcharts such as this are standard tools of business and are used often by the computing industry. Such flowcharts are very useful in reducing problems to the necessary steps required and are an invaluable aid in the field of programming. The example given deliberately omits overtime rates, irregular taxes such as FICA, and other such complicating features. The procedure followed by a clerk in performing this sequence of computations might be as follows:

1 The clerk looks up the employee's daily work record and adds the number of hours worked each day, obtaining the total number of hours worked during the week.

2 The total number of hours worked is multiplied by the pay rate, and the total earnings for the week are obtained.

3 The total earnings are multiplied by the tax rate for the employee, and the amount of withholding tax is found.

4 The withholding tax is subtracted from the total earnings.

5 Any regular deductions such as insurance are subtracted.

6 A record of each of the above operations is entered in the company books, and a check is made out for the correct amount.

Clearly almost all the above procedures can be mechanized by a machine which can be made to add, multiply, and subtract in the correct sequence. The machine also must have the following less obvious features:

1 The ability to remember the intermediate results that have been obtained. For instance, the total amount earned must be remembered while the tax is being figured. It is also convenient to keep the employee's pay rate, rate of withholding tax, insurance rates, and the amount regularly given to charity in the machine.

2 The ability to accept information. The records of time worked, changes in pay rates, deduction amounts, etc., must be entered into the machine.

3 The ability to print out the results obtained.

[1]It is interesting that 95 percent of the checks issued by the federal government are made out by computers.

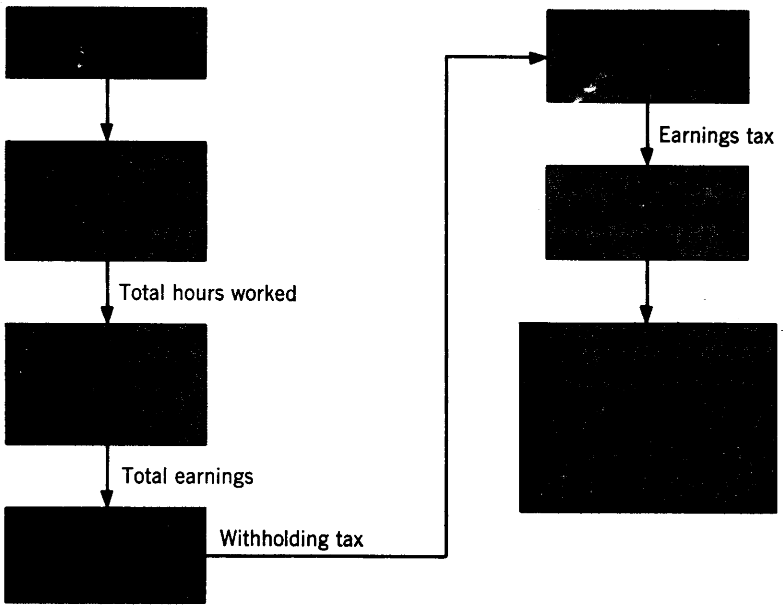


FIGURE 1.1

Flowchart of paycheck calculation.

The widespread acceptance of digital computers in payroll offices is largely due to the repetitious type of work normally done there. Mechanization of such tasks is straightforward, although often complicated; but the additional accuracy and speed, as well as the lower operating costs, which electronic business machines make possible, has made their use especially popular in this field.

BUSINESS APPLICATIONS

1.3 The main difference between the use of digital machines in business and in scientific work lies in the ratio of operations performed to total data processed. While the business machine performs only a few calculations using each datum, a great volume of data must be processed. The scientific problem generally starts with fewer data, but a great many calculations are performed using each datum. Both types of machines still fall under the heading of digital computers, and either type of work may be done on all computers, although some machines may be better adapted to one or the other type of problem.

The description of the use of a computer in figuring payrolls (Sec. 1.2) is an example of a business application and illustrates the similarity in programming the operation of a computer and figuring out employee office procedures. First, the problem to be solved is reduced to a series of simple operations: finding the name of the next employee whose wages are to be computed, figuring how many hours he or she has worked, and multiplying this figure by the hourly rate of pay. After the procedure to be used has been worked out and explained to the clerk, the clerk is provided with the necessary numerical information, such as pay rates, insurance rates, etc. If the operations are further simplified, each step in Fig. 1.1 may be performed by a different clerk. For instance, the first clerk may find the employee's record and send it to the second clerk, who computes the total number of hours worked and presents it to the next clerk, who multiplies by the wage rate, and so on, until all the operations have been performed. Thus the breaking down of business procedures into basic steps is a very old practice indeed.

The procedure for preparing a list of instructions for a digital computer is basically the same. All the operations the computer is to perform are written in flowchart form (Fig. 1.1). Then the problem is broken down into a list of instructions to the computer which specify exactly how the solution is to be obtained. After the problem has been programmed, the list of instructions is read into the computer. The computer automatically performs the required steps. Notice that once the procedure has been established and the programmed steps have been read in, the programming is finished until a change in procedure is desired. Changes in rates, for instance, can be inserted by simply reading the new pay rates into the machine. This does not affect the procedure.

SCIENTIFIC APPLICATIONS

1.4 Modern science and engineering use mathematics as a language for expressing physical laws in precise terms. The electronic digital computer is a valuable tool for studying the consequences of these laws. Often the exact procedure for solving a problem has been found, but the time required to perform the necessary calculations manually is prohibitive. Sometimes it is necessary to solve the same problem many times with different sets of parameters, and the computer is especially useful for solving problems of this type. Not only is the computer able to evaluate types of mathematical expressions at high speeds; but also if a set of calculations is performed repeatedly on different sets of numerical values, the computer can compare the results and determine the optimum values that were used.

An algebraic formula is an expression of a mathematical relationship. Many of the laws of physics, electronics, chemistry, etc., are expressed in this form, in which case digital computers may be easily used, because algebraic formulas may be directly changed to the basic steps they represent. Figure 1.2 is a flowchart illustrating the steps necessary to evaluate the expression $ax^3 + bx^2 + cx + d$, given numerical values for a, b, c, d, and x. The required steps are as follows:

1 Multiply a times x, yielding ax.

2 Add b, yielding $ax + b$.

3 Multiply this by x, forming $ax^2 + bx$.

4 Add c, yielding $ax^2 + bx + c$.

5 Multiply this by x: $x(ax^2 + bx + c)$, or $ax^3 + bx^2 + cx$.

6 Add d, obtaining $ax^3 + bx^2 + cx + d$.

It would take several minutes to perform the calculations necessary to evaluate this algebraic expression for a single set of values by using manually operated calculators, but practically any computer could perform this series of operations

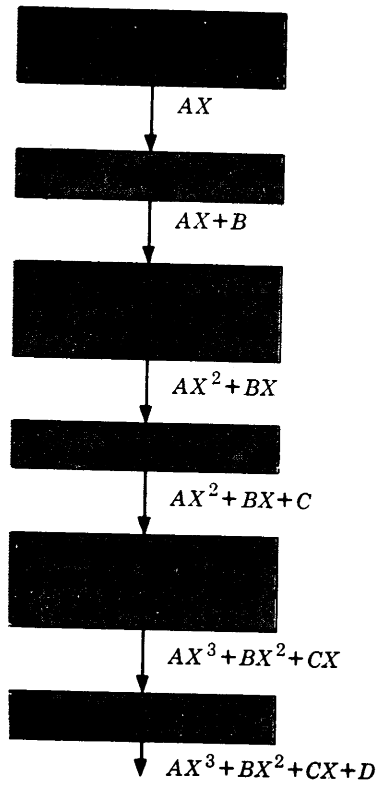


FIGURE 1.2

Flowchart of evaluation of expression.

several thousand times per second. Although the algebraic expression shown is certainly much simpler than many formulas encountered by members of the engineering and scientific professions, the value of using a computer for certain types of problems may be readily seen.

SOME DIFFERENT TYPES OF COMPUTER SYSTEMS

1.5 A familiar early use of computers is in operating programs punched into cards (or perhaps recorded on paper or magnetic tape) and run by a computer which then prepares printouts, checks, or some form of data presentation recording the results. This is an example of ***batch processing***. When a computer is used in this way, the input data (and often the program) are introduced to the computer and processed automatically, generally without operator intervention. Often many different jobs (or sets of data) are processed, one right after the other, or even at the same time, but without any interaction from the system's user during program operation. For instance, the keypunch operators may punch many decks of cards containing data on customers and claims. Then these cards are stacked and transported to a large computer which processes the cards, issuing checks, sending out bills, printing records for the company, etc. This is batch processing.

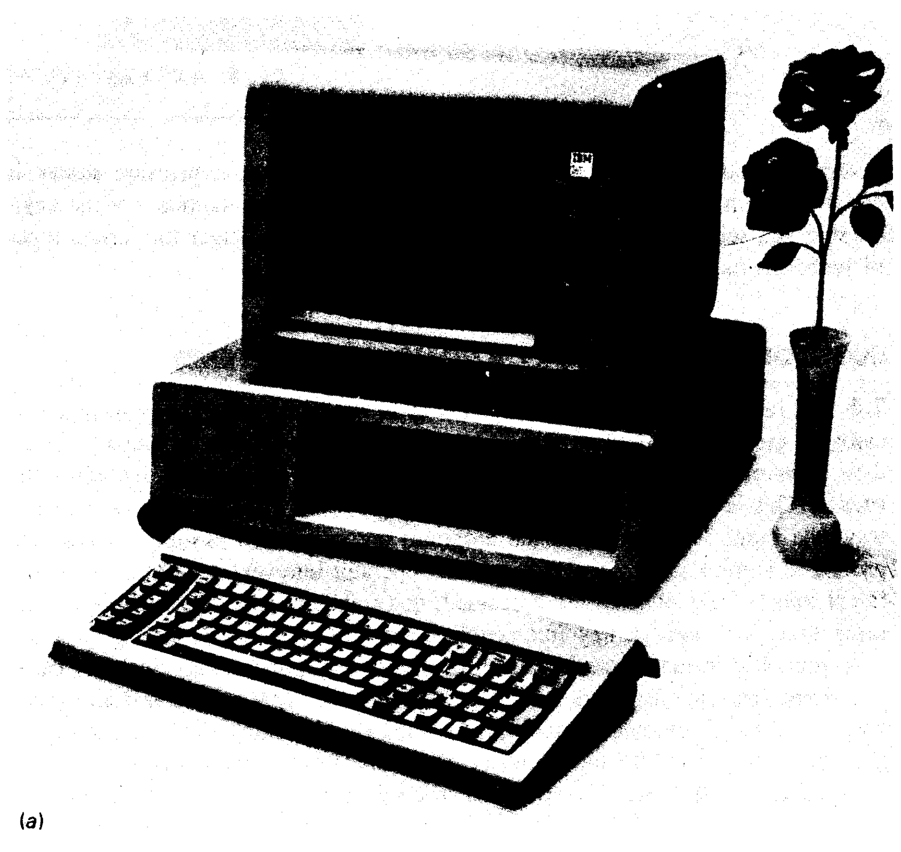
Similarly, the user of a computer in a scientific laboratory may submit program cards and data cards as a *job*, with an elastic band around these, along with any notes to the operators of the machine. A number of these decks of cards are collected, stacked, and finally run by the computer. Later the results are printed and, perhaps even the next day, delivered to the individual users. This is also batch processing.

In other types of systems, users interact with the computer directly, inserting and receiving the data as desired. For instance, an airline ticket agent wishes to make a reservation. The agent types the desired aircraft flight number and passenger identification on a special typewriter which communicates, via the telephone lines, with a computer. The computer looks in its memory; sees whether the flight is full, and, if not, enters the passenger's name on its list for the flight; and then communicates this fact back to the airline ticket agent. If no seats are available, the computer sends this information to the ticket agent, who attempts to interest the passenger in another flight. In this way an airline connects all its ticket agents, keeping a constant record of flights, passengers, and payments and doing all the bookkeeping. The terminals where the ticket agents are located are scattered throughout the world, but communicate with the computer via telephone lines. Motels, hotels, stockbrokers, and many businesses have similar systems for reservations, information transferral, and bookkeeping.

All these are called *interactive* systems, for the users of the system com-

FIGURE 1.3

(*a*) Personal computer with CRT display. (*b*) Computer terminal with a printer. (*DEC, Inc.*)


(*a*)

municate directly with the computer, and the computer responds directly. The development of these systems has progressed in parallel with the development of keyboard input devices, as well as output devices for users of various types, including cathode-ray tube (CRT) displays, printers, and other data display devices.

Interactive systems are widely employed in scientific applications where users operate their programs at a terminal connected to the computer, perhaps by telephone lines, trying changes and variations at will. Experimenters can try a set of inputs and study the results, then try other inputs and study these results. The technique of interactive computing is used by circuit designers, architects, and chemists, and in almost any area, including medical systems for hospital use.

Personal, or home, computers are also examples of interactive systems since the user communicates directly with the computer, inputting data and sometimes programs, and directly receiving results. Fig. 1.3(*a*) shows a personal computer with a keyboard and CRT display.

A widely used input-output device is the *terminal*, shown in Fig. 1.3(*b*). This is an example of a keyboard which is typewriterlike, generating a printed record when used, but also generating electric signals that can be used as computer input. Similarly, electronic signals from a computer can be used to control the

(*b*)

terminal, and the terminal's printer will type, under the computer's control, the results of calculations.

Terminals are sometimes used in systems in which the console terminal is some distance from the computer. A special attachment called a *modem* is used, which makes it possible to transmit the electric signals generated by the terminal to the computer and receive the computer's response back over telephone lines. At the computer another modem is located which also can transmit or receive, and this allows communication in both directions over telephone lines. The user of the terminal simply dials the number at which the computer is located, establishes a connection and the user's identity and right to use the computer (the computer generally checks a password), and then proceeds to use the computer.

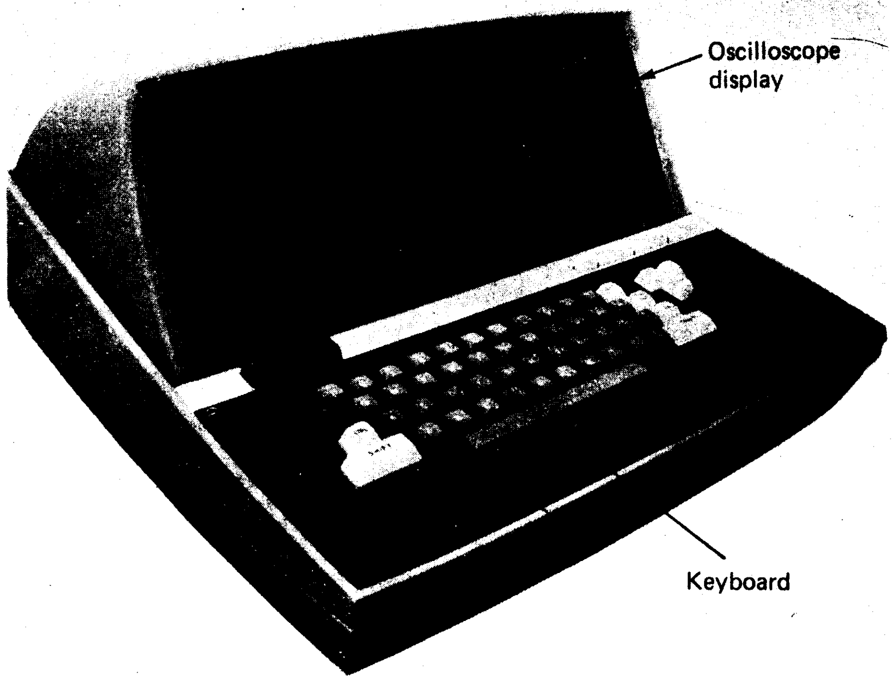
Terminal characteristics are fairly well standardized, and the same terminals often can use several different computers, when available. Now many companies provide computer service to users who have terminals at their disposal. The users simply dial the computer they prefer.

Figure 1.4 shows a terminal with a CRT display which is similar to a television, thus providing a temporary display (instead of *hard copy*, which is the printed page). This type of output device enables the computer to draw pictures or make graphs as well as use printed characters.

The computer terminal in Fig. 1.4 is portable. An attachment on the rear called an *acoustic coupler* which holds a telephone handset is provided so that

FIGURE 1.4

Portable computer terminal with CRT display. (*Logitron, Inc.*)



when the telephone handset is placed in the attachment, a computer can be dialed. Once the connection is made, the terminal then generates audio tones into the handset when keys are depressed on the keyboard. The terminal receiver also decodes coded tones representing characters generated by the computer, displaying the information received on the CRT display device. In this way the user of a terminal can "converse," or communicate, with any compatible computer connected to the telephone system.

An acoustic coupler generates and receives audio signals from the handset while a modem uses electric signals and is connected directly into a telephone jack. For this reason acoustic couplers are often used in portable terminals.

When a number of users share a computer, using the computer, sometimes via telephone lines, at the same time, the computer is said to be *timeshared*. By timesharing is meant that the computer is able to alternate and interweave the running of its programs so that several jobs or users can be on at the same time. This makes for more efficient use, and the airline, motel, and hotel reservation systems and most online systems use timesharing.

COMPUTERS IN CONTROL SYSTEMS

1.6 The ability of digital computers to make precise calculations and decisions at high speeds has made it possible to use them as parts of control systems. The Air Traffic Control System used at airports is an example of computer usage in a control system. In this system, data from a network of radar stations, which are used to detect the positions of all aircraft in the area, are fed via communication links into a high-speed computer. The computer stores all the incoming positional information from the radar stations and from this calculates the future positions of the aircraft, their speed and altitude, and all other pertinent information. A number of other types of information are also relayed into the computer, including information from picket ships, AEW aircraft, Ground Observer Corps aircraft spotters, flight plans for both military and civilian aircraft, and weather information.

A computer receives all this information and from it calculates a composite picture of the complete air situation. The computer then generates displays on special oscilloscopes which are used by air traffic controllers. By means of radio links, the computer automatically guides aircraft in and out of airports.

A system of this sort is called a *real-time control system* because information must be processed and decisions must be made in real time. When a computer is used to process business data or to perform most scientific calculations, time is not as critical a factor. In real-time systems, the computer must "keep up," processing all data at high speeds in order to be effective.[2]

Other examples of real-time control applications include oil refineries and other manufacturing areas which use the computer to control the manufacturing processes automatically. Digital computers are used to guide machine tools which perform precision-machining operations automatically, as shown in Fig. 1.5. Fur-

[2]Most reservation systems would be considered real-time systems by their operators, since delays of any consequence would be detrimental to business.

FIGURE 1.5

Computer-controlled machine tools with interactive terminal-based plant communication system. (*IBM.*)

ther, both staffed and nonstaffed space vehicles carry digital computers which perform the necessary guidance functions, while a network of computers on the ground monitors and directs the progress of the flight.

Most real-time control systems require an important device known as an *analog-to-digital (A/D) converter.* The inputs to these systems in many cases are in the form of *analog quantities* such as mechanical displacements (for instance, shaft positions) or temperatures, voltages, pressures, etc. Since the digital computer operates on digital rather than analog data, a fundamental "language" problem arises which requires the conversion of the analog quantities to digital representations. The A/D converter does this.

The same problem occurs at the computer output, where it is often necessary to convert numerical output data from the computer to mechanical displacements or analog-type electric signals. For instance, a "number" output from the computer might be used to rotate a shaft through the number of revolutions indicated by the output number. A device which converts digital-type information to analog quantities is called a *digital-to-analog (D/A) converter.* A description of both A/D and D/A converters is found in Chap. 7.

The basic elements of a control system using a digital computer consist of (1) the data-gathering devices which perform measurements on the external environment and, if necessary, also perform analog-to-digital conversion on the data from the system to be controlled; (2) the digital computer itself, which performs calculations on the data supplied and makes the necessary decisions; and (3) the means of communication with, or control over, certain of the elements in the external environment. If no person aids the computer in its calculations or decisions, the system is considered to be fully automatic; if a human being also enters the control loop, the system is defined as semiautomatic.

Figure 1.5 shows a computer being used in a manufacturing application. Because of their high speed, computers, often operating unattended, can measure, test, analyze, and control manufacturing functions as they occur. Computers can handle shop floor control, quality control testing, materials handling, and production monitoring. The typewriterlike station to the left in Fig. 1.5 is used to communicate with the system.

BASIC COMPONENTS OF A DIGITAL COMPUTER

1.7 The block diagram in Fig. 1.6 illustrates the five major operational divisions of an electronic digital computer. Although presently available machines vary greatly in the construction details of various components, the overall system concepts remain roughly the same.

A digital computer may be divided into the following fundamental units:

1 *Input* The input devices read the necessary data into the machine. In most general-purpose computers, the instructions that constitute the program must be read into the machine along with all the data to be used in the computations. Some of the more common input devices are keyboards, punched-card and punched-paper-tape readers, magnetic-tape readers, and various manual input devices such as toggle switches and pushbuttons.

2 *Control* The control section of a computer sequences the operation of the computer, controlling the actions of all other units. The control circuitry interprets

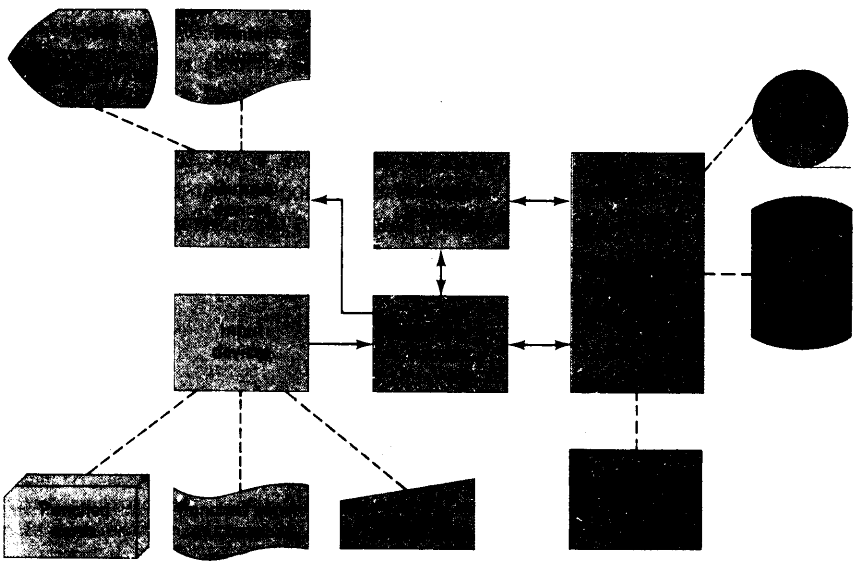


FIGURE 1.6

Block diagram of typical digital computer.

the instructions which constitute the program and then directs the rest of the machine in its operation.

3 *Memory* The memory, or storage, section of the computer consists of the devices used to store the information that will be used during the computations. The memory section of the computer is also used to hold both intermediate and final results as the computer proceeds through the program. Memory devices are constructed so that it is possible for the control unit to obtain any information in the memory. The time required to obtain information may vary somewhat, however, and is determined by the type of device used to store the information. Common storage devices are integrated-circuit memories, magnetic tape, and magnetic disks.

4 *Arithmetic-logic unit* The arithmetic-logic units of most computers are capable of performing addition, subtraction, division, and multiplication as well as some ''logical operations'' which are described later. The control unit tells the arithmetic-logic unit which operation to perform and then sees that the necessary numbers are supplied. The arithmetic element can be compared to the calculating machines described previously in that the numbers to be used are inserted, and it is then directed to perform the necessary operations.

5 *Output* The output devices are used to record the results obtained by the computer and present them to the outside world. Most output devices are directed by the control element, which also causes the necessary information to be supplied to them. Common output devices are CRT displays, printers, card-punching machines, and magnetic-tape drives. There are also many other types of output devices, such as lights, buzzers, and loudspeakers.

CONSTRUCTION OF MEMORY

1.8 The inner, or high-speed, memory is broken into a number of addresses, or locations. At each address a group of digits is stored and is handled by the computer as a unit. The group of digits stored at each address in memory is generally referred to as a memory *word*.[3] Each address in memory is assigned a number, and then the address is referred to by that number. We say that address 100 contains the value 300 or that address 50 contains an instruction word. In most computers, instruction words may be stored in the same locations as number or data words, which makes the memory more flexible. Notice also that instruction words, when stored in the computer, consist of a group of digits.

The time it takes to obtain a word from a storage device is called the *acces*

[3]There are actually two ways of organizing a memory now in general use. One way is to store enough bits for a character at each address (a character is a 1 or 2, an A or B, etc.). These systems are called *character-addressable* or byte-addressable, systems. The second way is to store a complete *operand*, or instruction word, at each address. These systems are called *word-addressable* systems. As might be guessed, word-addressable systems have more information at each address. The character-addressable systems are more used, but we simplify the initial description by assuming that each successive address contains a word, as in word-addressable systems, and describe other systems later.

time. The access time of the storage devices used in a machine has a profound effect on the speed of the computer. One factor which for a long time impeded the construction of high-speed computers was the lack of reliable storage devices with short access times. The development of storage devices (such as integrated-circuit memories) with very short access times plus the ability to store information for an indefinite time was a great forward step.

INSTRUCTIONS

1.9 Computers can make simple logical decisions. Many of these decisions are based on numbers and so are quantitative rather than qualitative. The sort of numerical decision the computer might make is whether one number is larger than another or whether the result of some series of calculations is positive or negative. However, most decisions made by clerical workers and scientists are also based on figures. For instance, once physical phenomena have been expressed in formulas, the solutions to specific problems are expressed by means of numbers.

As mentioned, the digital computer does not figure out its own solution to problems, but must be told exactly how to solve any given problem as well as how to make all decisions. Preparing a list of instructions which tells the computer how to perform its calculations is called programming. The procedure for programming a problem generally consists of two separate steps. The first step, planning the program, involves determining the sequence of operations required to solve the problem. It sometimes consists in breaking the problem down into flow diagrams such as those illustrated earlier. Once the problem has been reduced to this form, it is ready to be *coded;* this is the second step. *Coding* consists in writing the steps outlined in the flowchart in a special language which can be interpreted by the computer. The final coded program consists of a list of instructions to the computer, written in a special format which details the operations the computer is to perform.[4]

We now describe a type of computer language called *assembly language*. The instruction words which direct the computer are stored in the computer in numerical form. The programmer rarely writes instructions in numerical form, however; instead, each instruction to the computer is written by using a letter code to designate the operation to be performed, plus the address in memory of the number to be used in this step of the calculation.[5] Later the alphabetic section of the instruction word is converted to numerical form by a computer program called an *assembler*. This is described later.

An instruction word as written by the programmer consists of two parts: (1) the *operation-code* part, which designates the operation (addition, subtraction, multiplication, etc.) to be performed, and (2) the *address* of the number to be used.

[4]The instructions described are typical instructions for a *single-address* computer. Computers are also constructed which use two or more addresses in each computer instruction word. These computers are described in Chap. 10. The single-address type of instruction is very straightforward and is used in the illustrations in this chapter.

[5]The following is a description of *machine-language* or *assembly-language*, programming. Applications are often written in a *high-level* language such as BASIC, Fortran, or Pascal which then must be translated to machine language before the computer can run the program.

A typical instruction word written by the programmer is

ADD 535

This instruction word is divided into two main parts: first, the operation-code part, consisting of the letters ADD, which directs the computer to perform the arithmetic operation of addition; and second, the address part, which tells the computer the address in storage of the number to be used.

Note that the second section of the instruction word gives only the location (address) in storage of the number to be added. The number 535 in the instruction shown is not the actual number to be added, but only tells the computer where to find the desired number. To what is the number at address 535 added? It is to be added to the number which is already in the arithmetic-logic unit in a storage device, or register, called an *accumulator*. If the accumulator contains zero before the instruction is executed, the accumulator will contain the number which is stored at address 535 after the instruction has been performed. If the accumulator contains the number 500 before the instruction is performed and if the number stored at address 535 is 200, then the number stored in the accumulator after the ADD operation will be 700. To illustrate this principle more fully, several more instructions are explained in Table 1.1.

TABLE 1.1

INSTRUCTION WORD		FUNCTION PERFORMED BY INSTRUCTION
OPERATION CODE	ADDRESS PART	
CLA	430	The accumulator is emptied of all previous numbers, and the number at address 430 is added to it. After the instruction is performed, the accumulator contains the number in storage at address 430. CLA is a mnemonic code for "clear and add."
ADD	530	The number located at address 530 is added to the number in the accumulator. After the instruction, the accumulator contains the sum of the number it previously contained and the number in address 530.
SUB	235	The number located at address 235 in the memory is subtracted from the number in the accumulator, and the difference is placed in the accumulator.
STO	433	The number in the accumulator is stored at address 433. Any information previously in this address is destroyed. The number which was in the accumulator before the instruction was performed remains in the accumulator. This is generally referred to as a STORE instruction.
HLT	000	The machine is ordered to stop. The number in the accumulator remains.

TABLE 1.2

ADDRESS IN MEMORY	INSTRUCTION WORD OPERATION CODE	INSTRUCTION WORD ADDRESS PART	CONTENTS OF ACCUMULATOR AFTER INSTRUCTION IS PERFORMED
1	CLA	6	200
2	ADD	7	500
3	ADD	8	900
4	STO	9	900
5	HLT	0	900
6 contains the number 200			
7 contains the number 300			
8 contains the number 400			

A short program which adds three numbers by using these instructions is shown in Table 1.2.

The program operates as follows: The control section starts with the instruction word at address 1, which clears the accumulator and then adds the number at address 6 into it. The instruction at address 2 adds the number at address 7 to the number already in the accumulator. This produces the sum of 200 + 300, or 500. The third instruction adds the contents of address 8 to this sum, giving 900 in the accumulator. This number is then stored in memory at location 9 in the memory. The machine is ordered to halt. Notice that the machine is stopped before it reaches the data. This is to prevent the control element from picking up the data, for instance, the number 200, which is at address 6, and trying to use it as an instruction.

There is no difference between a number and an instruction as far as storage is concerned. Both are stored in the same basic form. So the instructions are generally placed in a different section of the memory from the data to be used. The computer progresses through the instructions and is stopped before it reaches the data. The fact that either instructions or data may be stored at all addresses makes the machine more flexible. Either a large amount of data and a few instructions, or many instructions and few data, can be used as long as the total amount of storage available is not exceeded.

MULTIPLICATION INSTRUCTION

1.10 By adding another instruction, multiplication, it will be possible to write more sophisticated programs (see Table 1.3).

TABLE 1.3

INSTRUCTION WORD OPERATION CODE	INSTRUCTION WORD ADDRESS PART	FUNCTION
MUL	400	The number at address 400 is multiplied by the number already in the accumulator, and the product is placed in the accumulator.

TABLE 1.4

	INSTRUCTION WORD		
ADDRESS IN MEMORY	OPERATION CODE	ADDRESS PART	CONTENTS OF ACCUMULATOR AFTER INSTRUCTION IS PERFORMED
[illegible]	CLA	22	a
[illegible]	MUL	26	ax
[illegible]	ADD	23	$ax + b$
[illegible]	MUL	26	$x(ax + b)$ or $ax^2 + bx$
[illegible]	ADD	24	$ax^2 + bx + c$
[illegible]	MUL	26	$x(ax^2 + bx + c)$ or $ax^3 + bx^2 + cx$
[illegible]	ADD	25	$ax^3 + bx^2 + cx + d$
[illegible]	STO	27	$ax^3 + bx^2 + cx + d$
[illegible]	HLT	000	$ax^3 + bx^2 + cx + d$
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]

The program in Table 1.4 evaluates the expression $ax^3 + bx^2 + cx + d$. The actual quantities for a, b, c, d, and x are stored in memory at locations 22, 23, 24, 25, and 26, respectively. Notice that the program shown evaluates the expression for any values which might be read into these locations. The expression could be evaluated for any number of values for x by running the program for one value of x, then substituting the succeeding values of x into register 26, and re-running the program for each value. In practice, it is possible to have the program automatically repeat itself by means of special instructions. All the various desired values for x can be stored and the equation solved for each x without stopping the computer.

It can be seen from Table 1.4 that very complicated algebraic functions can be evaluated by using only a very few instructions. The program shown can be performed by a high-speed computer in less than 1/1,000,000 second(s). The value of such speed in the solution of the more complex problems encountered in engineering and science may be readily seen. Computers are making possible engineering techniques which were previously unusable because of the high costs in time and money of lengthy computations.

BRANCH, SKIP, OR JUMP INSTRUCTIONS

1.11 All the instructions explained so far have been used to perform problems in simple arithmetic. However, the computer is able to repeat the same sequence of instructions without being stopped and restarted. This facility is provided by a group of instructions referred to as *branch*, *skip*, or *jump* instructions. These instructions tell the computer not to perform the instruction at the address following that of the instruction being performed, but to skip to some other instruction. Some branch instructions are *unconditional* in nature and cause the computer to skip regardless of what the conditions may be. Other branch instructions are *conditional* and tell the computer to skip only if certain things are true. Branch instructions

TABLE 1.5

OPERATION CODE	ADDRESS PART	FUNCTION
BRA	420	This instruction tells the computer to perform the instruction at address 420 next. The computer will skip or branch from the instruction it would have performed and perform instruction 420 instead. Then the computer will perform instruction 421, followed by 422, etc.
BRM	420	The computer will branch to instruction 420 only if the number in the accumulator is negative. If the number is positive or zero, the computer will not branch, but will perform the instruction stored at the next address in memory. [illegible] an minus."

enable the computer to make logical choices which alter its future actions. Two typical instructions are shown in Table 1.5.

The short program shown in Table 1.6 illustrates several very important principles. The purpose of the program is to add all the even integers from 2 to 100. The program is of the repetitious sort where a few short orders are used to generate a program which runs for some time by repeating the same sequence of instructions. The program illustrates how the ability to branch on a negative number can be used to form a counter that will determine how many times a part of a program is repeated.

The number stored in location 39 increases by 2 each time the program runs through. The total of these numbers is stored in address 43 which, after the program has halted, contains the total of all the numbers that have been in location 39. The number stored at address 40 decreases in magnitude by 1 each time the program

TABLE 1.6

	INSTRUCTION WORD		CONTENTS OF ACCUMULATOR		
ADDRESS	OPERATION CODE	ADDRESS PART	1ST TIME	2D TIME	LAST TIME
1	CLA	39	0	2	98
2	ADD	41	2	4	100
3	[illegible]	39	2	4	100
4	ADD	43	2	6	2 + 4 + 6 + ··· + 100
5	STO	43	2	6	2 + 4 + 6 + ··· + 100
6	[illegible]	[illegible]	−50	−49	−1
7	[illegible]	42	−49	−48	+0
8	[illegible]	40	−49	−48	+0
9	[illegible]	1	−49	−48	+0
10	[illegible]	000			
[illegible]					

runs through, until the number stored at 40 is no longer negative. When the program "falls through" the BRANCH WHEN MINUS (BRM) instruction, it performs the next instruction in sequence, which is a HALT instruction. Zero is considered to be a positive number, although this varies with different machines. Notice that the first section of the program will cycle a number of times equal to the negative number stored in register 40. A simulated counter is formed by the −50 stored at address 40, the 1 stored in location 42, and the instructions at addresses 6 through 9. Any sequence of instructions which precedes a counter of this sort will be run through the number of times determined by the counter. This is an especially useful device for iterative schemes when the number of iterations required is known.

PROGRAMMING SYSTEMS

1.12 The preceding discussion showed a basic procedure for writing an assembly-language program. There are, however, various types of *programming languages* which greatly facilitate the actual writing of programs. One of the first things the programming profession discovered was that the greatest aid to programming was the computer itself. It was found to be capable of translating written programs from a language which was straightforward and natural for the programmer to computer, or machine, language.

As a result, programs were written whose purpose was to read other programs written in a language natural for the programmer and to translate them into the computer's language. The program systems now in use are primarily of two types: *assemblers* and *compilers.*[6] The assembler and the compiler are intended for the same basic purpose: Each is a program designed to read a program written in a programming language and to translate it. The assembler or compiler is read into the computer first and then is followed by the program to be translated. After translation the assembler or compiler generally stores the machine-language program on magnetic disks or tape or in some other kind of memory so that it can be performed when desired.

The purpose of this procedure is to enable programmers to write the operations they want the computer to perform in a manner that is simpler than machine language. The language which the programmer writes is called a *programming language,* and a program written in such a language is called a *source program.* The translated program in machine (or some intermediate) language is called an *object program.*

To return to the subject of the translator programs, an assembly language differs from a compiler language in that most assembly languages closely resemble machine languages, primarily because each instruction to the computer in assembly language is translated to a single computer word. In compiler systems, a single instruction to the computer may be converted to many computer words.

[6]There is also a translator type of program that translates one line or statement at a time, called an *interpreter,* and is used with such programming languages as BASIC. Each statement is run by the computer after translation. Interpreters perform the some general functions as compilers but need not translate an entire program before operation.

1.13 Each instruction to the computer in a programming language is called a *statement*. The basic characteristic which most distinguishes an assembly language is that each statement is translated by the assembly program to a single machine instruction word.[7] As a result, an assembly language resembles machine language. The facilities offered the programmer are substantial, however, and generally include the following:

1 *Mnemonic operation codes* The programmer can write instructions to the computer using letters instead of binary numbers, and the letters which designate a given operation are arranged into a mnemonic code that conveys the "sense" of the instruction. In the preceding example of coding, the mnemonic codes ADD, MUL, CLA, etc., were used. The assembler would translate these mnemonic codes to the correct machine binary numbers and "package" these into the instruction words constituting the object program.

2 *Symbolic referencing of storage addresses* One of the greatest facilities offered the programmer is the ability to name the different pieces of data used in the program and to have the assembler automatically assign addresses to each name.

If we wish to evaluate the algebraic expression $y = ax^3 + bx^2 + cx + d$ as in Sec. 1.10, the program can appear as shown in Table 1.7.

Notice that the address of the first instruction was simply given the name FST, consisting of three letters, and that no further addresses in memory were specified. If we tell the assembler that FST = 1, the assembler will see that the instructions are placed in memory as in the program in Sec. 1.10. Notice also that the operands were simply given the variable names X, A, B, C, and D, as in the equation, instead of assigning addresses in memory to them. The assembly program will assign addresses to these names of variables, and if it assigns A to 22, B to 23, C to 24, etc., the final program will look as in Sec. 1.10.

[7]This is not, of course, strictly the case (sometimes a single statement may be translated to several words), but generally translation is into a single instruction word that can require one or more memory locations or words.

TABLE 1.7

	INSTRUCTION WORD	
ADDRESS IN MEMORY	OPERATION	OPERAND
FST	[illegible]	A
	[illegible]	[illegible]
	[illegible]	[illegible]
	[illegible]	[illegible]
	[illegible]	C
	[illegible]	[illegible]
	[illegible]	D
	STO	Y
	HLT	

The assembler will also see that actual arithmetic values for X, A, B, C, and D are placed in the correct locations in memory when the data are read into the computer.

3 *Convenient data representation* This simply means that the programmer can write input data as, for instance, decimal numbers or letters, or perhaps in some form specific to the problem, and that the assembly program will convert the data from this form to the form required for machine computation.

4 *Program listings* An important feature of most assemblers is their ability to print for the programmer a listing of the source program and the object program, which is in machine language. A study of these listings will greatly help the programmer in finding any errors made in writing the program and in modifying the program when this is required.

5 *Error detection* An assembler program will notify the programmer if an error has been made in the usage of the assembly language. For example, the programmer may use the same variable name, for instance, X, twice and then give X two different values; or the programmer may write illegal operation codes, etc. This sort of diagnosis of a program's errors is very useful during the checking out of a new program.

Assemblers provide many other facilities which help the programmer, such as the ability to use programs which have already been written as part of a new program and the ability to use routines from these programs as part of a new program. Often programmers have a set of different programs which they will run together in different combinations. This is made possible simply by specifying to the assembler the variable names in the different programs which are to be the same variable, the entry and exit points for the programs, etc. Thus programs written in an assembly language can be linked together in various ways

Let us consider the short program in Table 1.6 which sums the even integers from 0 to 100. This will illustrate the use of symbolic names for addresses when a branch instruction is used. The assembly-language program is shown in Table 1.8.

TABLE 1.8

	INSTRUCTION WORDS		
ADDRESS	OPERATION	OPERAND	COMMENTS
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	Last value of integer
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	Stores sum for [illegible]
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	

Notice that the values of the variables were specified before the program was begun; DEC indicates that the values given for A, B, C, D, and E are in decimal. This enables the assembler to locate the variables in the memory and assign values to them.[8] Also note that the transfer instruction BRM was to the symbolic address N.

If the assembler were told to start the program at address 1 in the memory, conversion to object or machine language would make it look similar to the one in Sec. 1.10, provided the assembler decided to store A, B, C, D, and E in locations 39 through 43.

HIGH-LEVEL LANGUAGES

1.14 More advanced types of programming languages are called *compiler languages, high-level languages,* or *problem-oriented languages.* These are the simplest languages to use for most problems and are the simplest to learn. These languages often reveal very little about the digital machines on which they are run, however. The designer of the language generally concentrates on specifying a programming language that is simple enough for the casual user of a digital computer and yet has enough facilities to make the language and its associated compiler valuable to professional programmers. In fact, many languages are almost completely computer-independent, and programs written in one of these languages may be run on any computer that has a compiler or translator for the language in its program library.

Certain languages have been very successful and have found extremely wide usage in the computer industry. The best known are Fortran, Pascal, BASIC, and Cobol. A program written in these langauges can be run on most computers that have a memory size large enough to accommodate a compiler, because most manufacturers will prepare a compiler for each of these for their computer.

The following section gives an introduction to a specific high-level language. Those who are familiar with languages such as BASIC, Fortran, or Cobol may omit it. The text is arranged so that subsequent material in the book does not depend on the following section, and this section can be omitted or studied at a later time.

A SHORT INTRODUCTION TO PASCAL

***1.15**[9] The distinguishing feature of a compiler and its associated programming language is that a single statement which the programmer writes can be converted by the compiler to a number of machine-language instructions. In Pascal, for instance, a single statement to the computer can generate quite a number of instructions in object, or machine, language. As a result, the language is not particularly dependent on the structure of the computer on which the program is run, and the following programs may be run on any computer with a Pascal compiler.

[8]In a sense, the operation code DEC says "assign the decimal value in the operand column to the variable name in the address column."

[9]All sections marked with an asterisk can be omitted on a first reading without loss of continuity or of overall understanding.

The Pascal compiler is written in assembly or machine language, and a Pascal program is ultimately run in machine language. For this reason, knowledge of the computer and its organization can be of great use to those writing and checking out programs. Further, for the systems programmers (those who maintain, modify, and prepare the compilers, assemblers, load programs, etc.) a knowledge of the computer on which the program is operated is indispensable. The fact that compilers and computers are backed up by an army of technical personnel—from systems programmers through design and maintenance engineers, technicians, and computer operators—is often overlooked by the user of the machine whose program is miraculously debugged and operated. Like most electronic devices, digital computers are not as independent and self-supporting as they may appear to casual users.

Thus forewarned, let us examine the structure of Pascal in a little detail, leaving a more complete exposition to one of the references.

To begin, consider the following simple complete Pascal program:

```
PROGRAM ADDNUMS;
  VAR
    A, B, Y: INTEGER;
  BEGIN
    A := 50;
    B :=20;
    Y := A + B
  END.
```

The first line of any Pascal program must begin with the word PROGRAM, followed by the name of the program. The name of the above program is ADDNUMS. The section of program

```
VAR
  A, B, Y: INTEGER;
```

is called a *variable declaration section*. This declares the variables A, B, and Y to be of type INTEGER, specifying that A, B, and Y may assume integral values. The words BEGIN and END specify the starting and ending points of the section of the program which is ultimately run. The three lines

```
A := 50;
B := 20;
Y := A + B
```

are program *statements*. The statements specify the actions to be taken by the computer. When the program is run, the statements are executed in order, starting with the first statement after BEGIN and ending with the statement before END. Note that a semicolon is placed after each statement except the final END statement. Semicolons are used to separate statements in Pascal.

The three statements of this program are called *assignment* statements. An assignment statement acts to change the value of a given variable. The assignment operator is the ":=" sign. The variable name on the left side of the := symbol is the variable whose value is to be changed. The right side of the := symbol is an *expression* that determines the new value which the variable is to assume. For example, when writing Y := A + B; we mean "replace the current value of Y with that of A + B." Thus, after the three statements

```
A := 50;
B := 20;
Y := A + B
```

are executed, A will be equal to 50, B to 20, and Y to 70. As a further example, we can increase, decrease, or otherwise change the current value of Y by adding to or subtracting from it. Consider

```
Y := 30;
A := 40;
Y := Y + A
```

After these statements are executed, Y will have the value of 70; that is, the location in memory that has been used to store Y will have the value 70 in it. Here is one further example:

```
Y := 20;
Z := 50;
W := Y + Z;
M := W - 30
```

After these statements are run, W will have the value 70 and M the value 40.

In Pascal, the addition symbol is the familiar plus sign; the subtraction symbol is the usual minus sign. Multiplication is indicated by an asterisk. Thus A * B means "multiply A by B." Therefore the program statements

```
A := 20;
B := 30;
C := A * B
```

give a value of 600 for C.

Let us examine one simple way to form a loop in the program, that is, to repeat a sequence of instructions until we desire to stop. This can be accomplished through use of the WHILE statement:

```
WHILE X <= Y DO
  BEGIN
    <sequence of statements>
  END;
```

The WHILE statement says "execute the set of statements between BEGIN and END while X is less than or equal to Y." In other words, repeat the sequence of statements until Y is greater than X. If we write the statements

```
T := 0;
M := 4;
N := 2;
P := 6;
WHILE N <= M DO
  BEGIN
    S := P * N;
    T := T + S;
    N := N + 1
  END;
Y := T * 2;
```

then the statements between BEGIN and END will be repeated until N is greater than M. Since N starts with the value 2 (and as 1 is added each time) while M starts with the value 4, N will take the values 2, 3, 4, 5. But when N equals 5, it will be greater than M, and the program will proceed with the instruction Y: = T * 2. The statements between BEGIN and END, therefore, will be repeated three times. The first time S will equal P times N, or 12; T will be equal to 0 + S, or 12; and N will be increased from 2 to 3. The second time S will equal P times N, or 6 times 3, which is 18; T will be equal to 12 + 18, or 30; and N will be increased to 4. The third time S will equal 6 times 4, or 24; T will take the value 30 + 24, or 54; and N will be increased to 5. Then N will be greater than M, and the next statement after END; will be operated. Now T, which has the value 54, will be multiplied by 2, giving 108, and Y will be assigned this value.

Let us examine two other features. To read data, we simply write

```
READ (X, Y, Z);
```

This will tell the compiler to arrange for reading the values of X, Y, and Z from a keyboard and continue with the values read as the current values of X, Y, and Z. So we must supply values of X, Y, and Z from the keyboard. The advantage is that we can change the values of X, Y, and Z by simply typing new values each time the program is run. If we write

```
READ (X, Y, Z);
M := X + (Y * Z);
```

and input the values X = 20, Y = 30, and Z = 2, we have M = 80. If we change our input values to read X = 5, Y = 3, and Z = 4, then we have M = 17 after we run the above.

To print out data, we write the statement

```
WRITE (X, Y, Z, A);
```

and the computer will print out the current values of X, Y, Z, A.

Note that the READ and WRITE statements assume that the programmer will be satisfied with the standard format for the input data and print statements. Assuming that this is the case, we can write the following statements which will first evaluate the equation $y = ax^3 + bx^2 + cx + d$ for values of A, B, D, and X which are read in from the terminal and for C = 1. If the value of Y for these particular values is greater than 2000, the value of Y will be printed as calculated and also the value C + 1. If, however, Y is less than or equal to 2000, the statements will calculate the smallest positive integer which, when substituted into C, will make $ax^3 + bx^2 + cx + d$ greater than 2000. The WRITE statement will print this value of C and the value of $ax^3 + bx^2 + cx + d$ associated with the value of C.

```
READ (A, B, D, X);
C := 0;
Y := 0;
WHILE Y <= 2000 DO
  BEGIN
    C := C + 1;
    Y := A * X * X * X + B * X * X + C * X + D;
  END;
WRITE (C, Y);
```

We have only touched on the power of this language. It is not possible in a short exposition to do more than show several statements and give a general idea of how such a language operates. Nevertheless, a clever programmer could do quite a lot with the limited vocabulary we have introduced.

There is a Pascal microprocessor version of Pascal which is widely used; it was written at the University of California at San Diego (UCSD). The translators (compilers) for UCSD Pascal produce machine language that can be run on a microprocessor.

SUMMARY

1.16 Although the electronic computer is the newest and most important technological development in the last fifty years, the history of the computer extends back to the 19th century, when computers were first designed. The modern electronic digital computer is made possible by more recent developments in solid-state physics and engineering, however, and dates only to the 1940s and 1950s.

In order to use a computer to solve a scientific problem or implement an office procedure, it is necessary to organize the problem into clearly defined steps. A flowchart is most useful in organizing these steps. Then the procedure can be programmed, which consists of writing down the steps to be taken by the computer in a special language developed for computer usage.

The two most used classes of computer languages are *assembly language* and *higher-level,* or *application-oriented,* language. Assembly language resembles true machine language, mirroring the internal structure of the computer for which it is written. The higher-level languages such as Fortran, BASIC, Pascal, and Cobol are translated by a computer to the particular machine-language representation which the computer can run. The higher-level languages are specifically designed to facilitate their use by the programmer. Assembly languages also provide many aids to the programmer, but they basically correspond in structure to the specific computer on which they will run.

The five sections of a computer and typical devices that might be used in these sections were described. Detailed descriptions of the operation of these sections and devices are presented in the following chapters.

One thing which should become apparent from this brief introduction is that once the statement types and details of a high-level language are learned, the job of programming a given problem is greatly facilitated.

QUESTIONS

1.1 Discuss possible applications of microprocessors in real-time control systems.

1.2 The computer's ability to translate languages such as Pascal and Fortran greatly simplifies programming. Comment on the difficulty a computer might have in translating English. What about ambiguities? Must programming languages avoid them?

1.3 Sometimes the same computer is used by several different companies during the day, and the computer is timeshared between these companies. Discuss problems which might arise in billing the companies for the computer's services.

1.4 Discuss interactive computer systems and give examples of businesses and industries that might use interactive systems.

1.5 Discuss batch processing and give several examples of businesses and industries that might use it.

1.6 Give examples of industries that might use real-time control systems for manufacturing.

1.7 Values for X, Y, and Z are stored at memory addresses 40, 41, and 42, respectively. Using the instructions for the generalized single-address computer described in Secs. 1.9 to 1.11, write a program that will form the sum X + Y + Z and store it at memory address 43.

1.8 Explain the difference between the *address* of a word in memory and the *word* itself.

1.9 Given that values for X, Y, and Z are stored in locations 20, 21, and 22, respectively, use the instructions for the computer given in Secs. 1.9 to 1.11 to write a program that will form $X^2 + Y^2 + Z^2$ and store this at memory address 40.

1.10 The program in Table 1.4 is run with the following values when it is started: 5 in address 22, 4 in address 23, 6 in address 24, 4 in address 25, and 2 in address 26. What value will be in address 27 after the program has been run?

1.11 Convert the flowchart in Fig. 1.2 to a computer assembly-language program, using the instructions described in the chapter.

1.12 If the program in Table 1.4 is started with location 22 containing 4, location 23 containing 4, location 24 containing 4, location 25 containing 1, and location 26 containing 2, what number will be stored in location 27 after the program is run?

1.13 Write a program that will store $X^5 + X$ in register 40, using the assembly language in Secs. 1.9 to 1.11, given that X is in register 20. Use fewer than 20 instructions. Now rewrite this program, using the assembly language in Sec 1.15.

1.14 If the program in Table 1.6 were operated but the number at address 40 were -30 instead of -50 when the program started, the number at address 43 would be the sum of all even integers from 2 to __ instead of 2 to 100.

1.15 Draw a flowchart for the program in Sec. 1.11.

1.16 Given that a value for X is stored at address 39, write a program that will form X^4 and store it at address 42, using the assembly language in Secs. 1.9 to 1.11.

1.17 Draw a flowchart showing how to find the largest number in a set of five numbers stored at locations 30, 31, 32, 33, and 34 in memory.

1.18 Values for X and Y are stored at addresses 30 and 31. Write a program that will store the larger of the two values at address 40, using the assembly language in Secs. 1.9 to 1.11.

1.19 Given three different numbers, determine whether they are in ascending or descending order. Draw a flowchart for the problem, and write a program, using the assembly language in Secs. 1.9 to 1.11. Assume that the numbers are stored in memory locations 30, 31, and 32.

1.20 A value for Y is stored at location 55 and a value for A at location 59. Write an assembly-language program to store AY^3 at location 40.

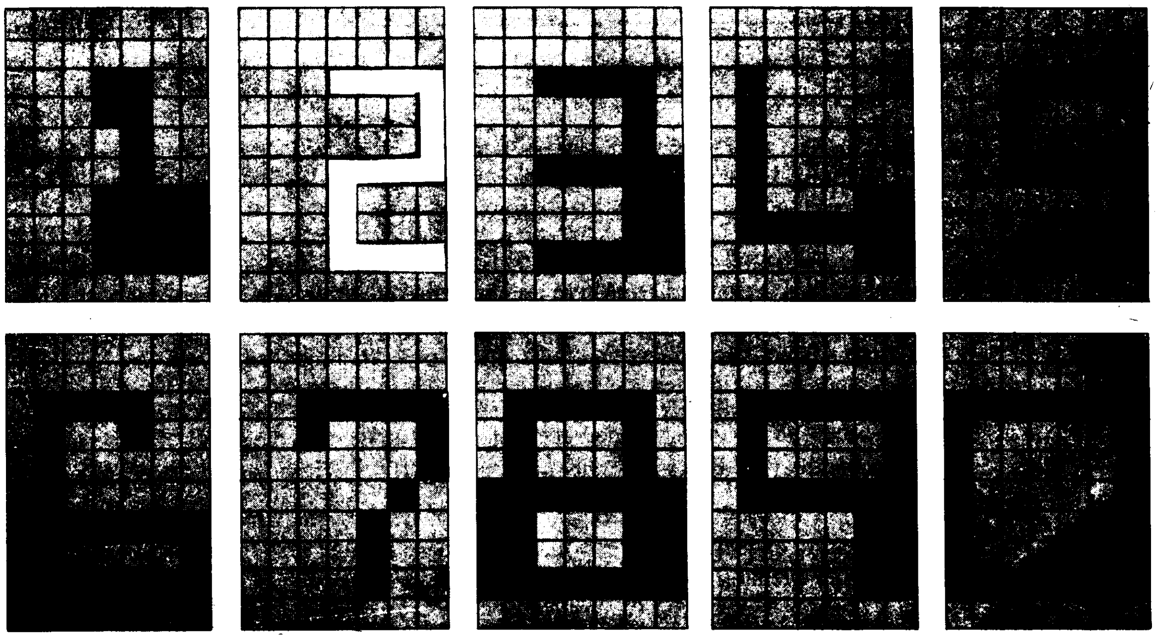
1.21 Write a program to find $ax^2 + by + cz^2$, with A in location 20, B in location 21, C in location 22, X in location 23, Y in location 24, and Z in location 25. Store your result in location 40.

1.22 A value for X is stored at address 40. Write a program that will store X^9 at address 45, using fewer than 10 instruction words.

1.23 Using the assembly language described in Secs. 1.9 to 1.11, write a program that will branch to location 300 if the number stored in memory register 25 is larger than the number stored in register 26 and which will transfer or branch to location 400 if the number at address 26 is equal to or larger than the number at address 25.

1.24 Write a program that will produce the value $Y - X$ or X, whichever is larger, in both the assembly language in Secs. 1.9 to 1.11 and the compiler language in Sec. 1.15. Store this value in location 300 for the assembly program, and assign the variable B to this value for the program written in the compiler language. For the assembly program, assume that X is in location 100 and Y in 101.

1.25 Calculate the largest of the three numbers A, B, and C and assign the largest



NUMBER SYSTEMS

It is India that gave us the ingenious method
of expressing all numbers by means of ten symbols,
each symbol receiving a value of position as well as an absolute
value; a profound and important idea which appears so simple
to us now that we ignore its true merit.

Marquis de Laplace

As a mathematician, Laplace could well appreciate the decimal number system. He was fully aware of the centuries of mental effort and sheer good luck which had gone into the development of the number system we use, and he was in a position to appreciate its advantages. Our present number system provides modern mathematicians and scientists with a great advantage over those of previous civilizations and is an important factor in our rapid advancement.

Since hands are the most convenient tools nature has provided, human beings have always tended to use them in counting. So the decimal number system followed naturally from this usage.

An even simpler system, the binary number system, has proved the most natural and efficient system for computer use, however, and this chapter develops this number system along with other systems used by computer technology.

2 NUMBER SYSTEMS

OBJECTIVES

1 An explanation of positional notation is given, and the idea of the base, or radix, of a number system is presented.

2 The binary number system is explained as well as how to add, subtract, multiply, and divide in this system. Then techniques for converting from binary to decimal and decimal to binary are given.

3 Negative numbers are represented in computers by using a sign bit, and this concept is explained. Negative numbers are often represented by using a complemented form rather than a signed-magnitude form. The two major complemented forms, true complement and radix minus one, are described.

4 The representation of decimal numbers using bistable devices can be accomplished with a binary-coded-decimal (BCD) system, and several of these are explained.

5 The octal and hexadecimal number systems are widely used in computer literature and manufacturer's manuals. These number systems are explained along with conversion techniques to and from decimal and binary.

DECIMAL SYSTEM

2.1 Our present system of numbers has 10 separate symbols, 0, 1, 2, 3, . . . , 9, which are called Arabic numerals. We would be forced to stop at 9 or to invent more symbols if it were not for the use of *positional notation*. An example of earlier types of notation can be found in Roman numerals, which are essentially additive: III = I + I + I, XXV = X + X + V. New symbols (X, C, M, etc.) were used as the numbers increased in value: thus V rather than IIIII is equal to 5. The only importance of position in Roman numerals lies in whether a symbol precedes or follows another symbol (IV = 4, while VI = 6). The clumsiness of this system can be seen easily if we try to multiply XII by XIV. Calculating with roman numerals was so difficult that early mathematicians were forced to perform arithmetic operations almost entirely on abaci, or counting boards, translating their results back to Roman numeral form. Pencil-and-paper computations are unbelievably intricate and difficult in such systems. In fact, the ability to perform such operations as addition and multiplication was considered a great accomplishment in earlier civilizations.

Now the great beauty and simplicity of our number system can be seen. It is necessary to learn only the 10 basic numerals and the *positional notational system* in order to count to any desired figure. After memorizing the addition and multiplication tables and learning a few simple rules, we can perform all arithmetic operations. Notice the simplicity of multiplying 12 × 14 by using the present system:

$$\begin{array}{r} 14 \\ \underline{12} \\ 28 \\ \underline{14} \\ 168 \end{array}$$

The actual meaning of the number 168 can be seen more clearly if we notice that it is spoken as "one hundred and sixty-eight." Basically, the number is a contraction of $1 \times 100 + 6 \times 10 + 8$. The important point is that the value of each digit is determined by its position. For example, the 2 in 2000 has a different value than the 2 in 20. We show this verbally by saying "two thousand" and "twenty." Different verbal representations have been invented for numbers from 10 to 20 (eleven, twelve, . . .), but from 20 upward we break only at powers of 10 (hundreds, thousands, millions, billions). Written numbers are always contracted, however, and only the basic 10 numerals are used, regardless of the size of the integer written. The general rule for representing numbers in the decimal system by using positional notation is as follows: $a_{n-1} 10^{n-1} + a_{n-2} 10^{n-2} + \cdots + a_0$ is expressed as $a_{n-1} a_{n-2} \cdots a_0$, where n is the number of digits to the left of the decimal point.

The *base*, or *radix*, of a number system is defined as the number of different digits which can occur in each position in the number system. The decimal number system has a base, or radix, of 10. Thus the system has 10 different digits (0, 1, 2, . . . , 9), any one of which may be used in each position in a number. History records the use of several other number systems. The quinary system, which has 5 for its base, was prevalent among Eskimos and North American Indians. Examples of the duodecimal system (base 12) may be seen in clocks, inches and feet, and dozens or grosses.

BISTABLE DEVICES

2.2 The basic elements in early computers were relays and switches. The operation of a switch, or relay, can be seen to be essentially bistable, or binary in nature; that is, the switch is either on (1) or off (0). The principal circuit elements in more modern computers are transistors. The desire for reliability led designers to use these devices so that they were always in one of two states, fully conducting or nonconducting. A simple analogy may be made between this type of circuit and an electric light. At any given time the light (or transistor) is either on (conducting) or off (not conducting). Even after a bulb is old and weak, it is generally easy to tell whether it is on or off.

Because of the large number of electronic parts used in computers, it is highly desirable to utilize them in such a manner that slight changes in their characteristics will not affect their performance. The best way of accomplishing this is to use circuits which are basically *bistable* (having two possible states).

COUNTING IN THE BINARY SYSTEM

2.3 The same type of positional notation is used in the binary number system as in the decimal system. Table 2.1 lists the first 20 binary numbers.

Although the same positional notation system is used, the decimal system uses powers of 10, and the binary system powers of 2. As was previously explained, the number 125 actually means $1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$. In the binary system, the same number (125) is represented as 1111101, meaning $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$.

TABLE 2.1

DECIMAL		BINARY	DECIMAL		BINARY
1	=	1	11	=	1011
2	=	10	12	=	1100
3	=	11	13	=	1101
4	=	100	14	=	1110
5	=	101	15	=	1111
6	=	110	16	=	10000
7	=	111	17	=	10001
8	=	1000	18	=	10010
9	=	1001	19	=	10011
10	=	1010	20	=	10100

To express the value of a binary number, therefore, $a_{n-1}\,2^{n-1} + a_{n-2}\,2^{n-2} + \cdots + a_0$ is represented as $a_{n-1}\,a_{n-1} \cdots a_0$, where a_i is either 1 or 0 and n is the number of digits to the left of the binary (radix) point.

The following examples illustrate the conversion of binary numbers to the decimal system:

$$\begin{aligned}
101 &= 1 \times 2^{3-1} + 0 \times 2^{3-2} + 1 \times 2^{3-3} \\
&= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 4 + 1 = 5 \\
1001 &= 1 \times 2^{4-1} + 0 \times 2^{4-2} + 0 \times 2^{4-3} + 1 \times 2^{4-4} \\
&= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 8 + 1 = 9 \\
11.011 &= 1 \times 2^{2-1} + 1 \times 2^{2-2} + 0 \times 2^{2-3} + 1 \times 2^{2-4} + 1 \times 2^{2-5} \\
&= 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\
&= 2 + 1 + \tfrac{1}{4} + \tfrac{1}{8} \\
&= 3\tfrac{3}{8}
\end{aligned}$$

Note that fractional numbers are formed in the same general way as in the decimal system. Just as

$$0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

in the decimal system,

$$0.101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

in the binary system.

BINARY ADDITION AND SUBTRACTION

2.4 Binary addition is performed in the same manner as decimal addition. Actually, binary arithmetic is much simpler to learn. The complete table for binary addition is as follows:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ plus a carry-over of 1

"Carry-overs" are performed in the same manner as in decimal arithmetic. Since 1 is the largest digit in the binary system, any sum greater than 1 requires that a digit be carried over. For instance, 100 plus 100 binary requires the addition of the two 1s in the third position to the left, with a carry-over. Since 1 + 1 = 0 plus a carry-over of 1, the sum of 100 and 100 is 1000. Here are three more examples of binary addition:

DECIMAL	BINARY	DECIMAL	BINARY	DECIMAL	BINARY
5	101	15	1111	$3\frac{1}{4}$	11.01
6	110	20	10100	$5\frac{3}{4}$	101.11
11	1011	35	100011	9	1001.00

Subtraction is the inverse operation of addition. To subtract, it is necessary to establish a procedure for subtracting a larger from a smaller digit. The only case in which this occurs with binary numbers is when 1 is subtracted from 0. The remainder is 1, but it is necessary to borrow 1 from the next column to the left. This is the binary subtraction table.

$0 - 0 = 0$
$1 - 0 = 1$
$1 - 1 = 0$
$0 - 1 = 1$ with a borrow of 1

A few examples will make the procedure for binary subtraction clear:

DECIMAL	BINARY	DECIMAL	BINARY	DECIMAL	BINARY
9	1001	16	10000	$6\frac{1}{4}$	110.01
−5	−101	−3	−11	$-4\frac{1}{2}$	−100.1
4	100	13	1101	$1\frac{3}{4}$	1.11

BINARY MULTIPLICATION AND DIVISION

2.5 The table for binary multiplication is very short, with only four entries instead of the 100 necessary for decimal multiplication:

$0 \times 0 = 0$
$1 \times 0 = 0$
$0 \times 1 = 0$
$1 \times 1 = 1$

The following three examples of binary multiplication illustrate the simplicity of each operation. It is only necessary to copy the multiplicand if the digit in the multiplier is 1 and to copy all 0s if the digit in the multiplier is a 0. The ease with which each step of the operation is performed is apparent.

```
DECIMAL     BINARY     DECIMAL       BINARY       DECIMAL    BINARY
  12          1100       102        1100110         1.25      1.01
 ×10         ×1010       ×8           ×1000         ×2.5     ×10.1
 ---        ------      ---      ----------         ----     -----
 120          0000       816     1100110000          625       101
             1100                                   250       1010
            0000                                   -----     ------
           1100                                    3.125     11.001
          -------
          1111000
```

Binary division is, again, very simple. As in the decimal system (or in any other), division by zero is meaningless. The complete table is

$$0 \div 1 = 0$$
$$1 \div 1 = 1$$

Here are two examples of division:

```
DECIMAL      BINARY
    5            101
  5)25      101)11001
                101
                  101
                  101
```

```
  DECIMAL              BINARY
     2.416···              10.011010101···
  12)29.0000         1100)11101.00
     24                   1100
     --                   ----
      50                   10100
      48                    1100
      --                    ----
       20                   10000
       12                    1100
       --                    ----
        80                   10000
        72                    1100
        --                    ----
         8                     ···
```

To convert the quotient obtained in the second example from binary to decimal, we would proceed as follows:

$$
\begin{aligned}
10.011010101 = 1 \times 2^{1} &= 2.0 \\
0 \times 2^{0} &= 0.0 \\
0 \times 2^{-1} &= 0.0 \\
1 \times 2^{-2} &= 0.25 \\
1 \times 2^{-3} &= 0.125 \\
0 \times 2^{-4} &= 0.0 \\
1 \times 2^{-5} &= 0.03125 \\
0 \times 2^{-6} &= 0.0 \\
1 \times 2^{-7} &= 0.0078125 \\
0 \times 2^{-8} &= 0.0 \\
1 \times 2^{-9} &= \underline{0.001953125} \\
& 2.416015625
\end{aligned}
$$

Therefore, 10.011010101 binary equals approximately 2.416 decimal.

2.6 There are several methods for converting a decimal number to a binary number. The first and most obvious method is simply to subtract all powers of 2 which can be subtracted from the decimal number until nothing remains. The highest power of 2 is subtracted first, then the second highest, etc. To convert the decimal integer 25 to the binary number system, first the highest power of 2 which can be subtracted from 25 is found. This is $2^4 = 16$. Then $25 - 16 = 9$. The highest power of 2 which can be subtracted from 9 is 2^3, or 8. The remainder after subtraction is 1, or 2^0. The binary representation for 25 is, therefore, 11001.

This is a laborious method for converting numbers. It is convenient for small numbers when it can be performed mentally, but is less used for larger numbers. Instead, the decimal number is repeatedly divided by 2, and the remainder after each division is used to indicate the coefficients of the binary number to be formed. Notice that the binary number derived is written from the bottom up.

$$\begin{aligned}
125 \div 2 &= 62 + \text{remainder of } 1\\
62 \div 2 &= 31 + \text{remainder of } 0\\
31 \div 2 &= 15 + \text{remainder of } 1\\
15 \div 2 &= 7 + \text{remainder of } 1\\
7 \div 2 &= 3 + \text{remainder of } 1\\
3 \div 2 &= 1 + \text{remainder of } 1\\
1 \div 2 &= 0 + \text{remainder of } 1
\end{aligned}$$

The binary representation of 125 is, therefore, 1111101. Checking this result gives

$$\begin{aligned}
1 \times 2^6 &= 64\\
1 \times 2^5 &= 32\\
1 \times 2^4 &= 16\\
1 \times 2^3 &= 8\\
1 \times 2^2 &= 4\\
0 \times 2^1 &= 0\\
1 \times 2^0 &= \underline{1}\\
& 125
\end{aligned}$$

This method will not work for mixed numbers. If similar methods are to be used, first it is necessary to divide the number into its whole and fractional parts; that is, 102.247 would be divided into 102 and 0.247. The binary representation for each part is found, and then the two parts are added.

The conversion of decimal fractions to binary fractions may be accomplished by using several techniques. Again, the most obvious method is to subtract the highest negative power of 2 which may be subtracted from the decimal fraction. Then the next highest negative power of 2 is subtracted from the remainder of the first subtraction, and this process is continued until there is no remainder or to the desired precision.

$$\begin{aligned}
0.875 - 1 \times 2^{-1} &= 0.875 - 0.5 = 0.375\\
0.375 - 1 \times 2^{-2} &= 0.375 - 0.25 = 0.125\\
0.125 - 1 \times 2^{-3} &= 0.125 - 0.125 = 0
\end{aligned}$$

Therefore, 0.875 decimal is represented by 0.111 binary. A much simpler method for longer fractions consists of repeatedly "doubling" the decimal fraction. If a 1 appears to the left of the decimal point after a multiplication by 2 is performed, a 1 is added to the right of the binary fraction being formed. If after a multiplication by 2, a 0 remains to the left of the decimal point of the decimal number, a 0 is added to the right of the binary number. The following example illustrates the use of this technique in converting 0.4375 decimal to the binary system:

	BINARY REPRESENTATION
$2 \times 0.4375 = 0.8750$	0.0
$2 \times 0.875 = 1.750$	0.01
$2 \times 0.75 = 1.50$	0.011
$2 \times 0.5 = 1.0$	0.0111

The binary representation of 0.4375 is, therefore, 0.0111.

NEGATIVE NUMBERS

2.7 A standard convention adopted for writing negative numbers consists of placing a *sign symbol* before a number that is negative. For instance, negative 39 is written as -39. If -39 is to be added to $+70$, we write

$$+70 + (-39) = 31$$

When a negative number is subtracted from a positive number, we write $+70 - (-39) = +70 + 39 = 109$. The rules for handling negative numbers are well known and are not repeated here, but since negative numbers constitute an important part of our number system, the techniques used to represent negative numbers in digital machines are described.

In binary machines, numbers are represented by a set of bistable storage devices, each of which represents one binary digit. As an example, given a set of five switches, any number from 00000 to 11111 may be represented by the switches if we define a switch with its contacts closed as representing a 1 and a switch with open contacts as representing a 0. If we desire to increase the total range of numbers that we can represent so that it will include the negative numbers from 00000 to -11111, another bit (or switch) will be required. We then treat this bit as a *sign bit* and place it before the magnitude of the number to be represented.

Generally, the convention is adopted that when the sign bit is a 0, the number represented is positive, and when the sign bit is a 1, the number is negative. If the previous situation, where five switches are used to store the magnitude of a number, is extended so that both positive and negative numbers may be stored, then a sixth switch will be required. When the contacts of this switch are open, the number will be a positive number equal to the magnitude of the number stored in the other five switches; and if the switch for the sign bit is closed, the number represented by the six switches will be a negative number with a magnitude determined by the other five switches. An example is shown in Fig. 2.1.

Sets of storage devices which represent a number or are handled as an entity

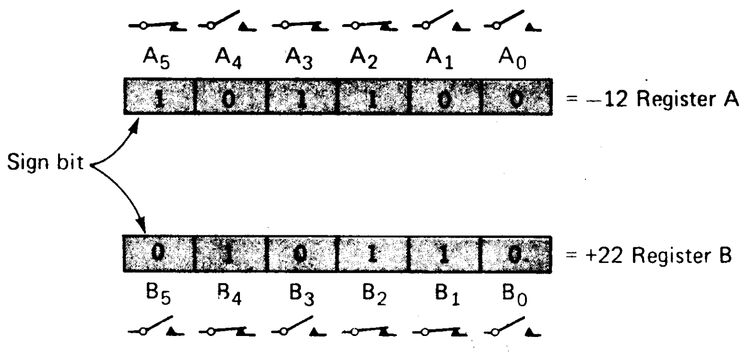


FIGURE 2.1

Example of negative-number representation.

are referred to as *registers,* and they are given names such as register A, register B, register C, etc. We can then write that register A contains −12 and register B contains +22. In writing a signed number in binary form, the sign bit is set apart from the magnitude of the number by means of an underscore, so that 00111 represents +0111, or positive 7 decimal, and 10111 represents −0111, or negative 7 decimal.

The use of the an underscore to mark the sign bit does not indicate that there is any difference between this bit and the other bits as the number is stored in a computer. Each and every binary bit is simply stored in a separate bistable device. A symbol other than the underscore could be used to separate the sign and magnitude bits, such as a hyphen, period, or a star. Then, −1011 (negative 11 decimal) could be written 1-1011 or 1*1011 or 1.1011 and +1100 as 0-1100 or 0*1100 or 0.1100. In fact, no marking whatever could be used, but it is felt that some indication makes for easier reading of signed numbers which use a sign bit.

USE OF COMPLEMENTS TO REPRESENT NEGATIVE NUMBERS

2.8 The convention of using a sign bit to indicate whether a stored number is negative or positive has been described. The magnitude of the number stored is not always represented in normal form, however, but quite often negative numbers are stored in *complemented form.* By using this technique, a machine can be made to both add and subtract, using only circuitry for adding. The actual technique involved is described in Chap. 5.

There are two basic types of complements which are useful in the binary and decimal number systems. In the decimal system the two types are referred to as the 10s *complement* and the 9s *complement.*

The 10s complement of any number may be formed by subtracting each digit of the number from 9 and then adding 1 to the least significant digit of the number thus formed. For instance, the 10s complement of 87 is 13, and the 10s complement of 23 is 77.

To subtract[1] one positive number (the minuend) from another (the subtra-

[1]The number of digits in each number must be the same. If one number has fewer digits than the other, then 0s can be added to the left until the number of digits is the same.

hend), first the 10s complement of the subtrahend is formed, and then this 10s complement is added to the minuend. If there is a carry from the addition of the most significant digits, then it is discarded, the difference is positive, and the result is correct. If there is no carry, the difference is negative, the 10s complement of this number is formed, and a minus sign is placed before the result.

Here are some examples.

```
NORMAL SUBTRACTION     10s COMPLEMENT SUBTRACTION
       89                 89      89
      -23                -23  =  +77
      ---                ---     ---
       66                     ┌── 166
                              └─→ the carry is dropped
       98                 98      98
      -87                -87  =  +13
      ---                ---     ---
       11                     ┌── 111
                              └─→ the carry is dropped
       49                 49
      -62                +38
      ---                ---
      -13                 87       no carry, so result is negative
                                   10s complement, or -13
       54                 54
      -81                +19
      ---                ---
      -27                 73       no carry, so result is negative
                                   10s complement, or -27
```

The 9s complement of a decimal number is formed by subtracting each digit of the number from 9. For instance, the 9s complement of 23 is 76, and the 9s complement of 87 is 12. When subtraction[2] is performed by using the 9s complement, the complement of the subtrahend is added as in 10s complement subtraction, but any carry generated must be added to the rightmost digit of the result. As is the case with 10s complement subtraction, if no carry is generated for the addition of the most significant digits, then the result is negative, the 9s complement of the result is formed, and a minus sign is placed before it.

```
NORMAL SUBTRACTION     9s COMPLEMENT SUBTRACTION
       89                  89      89
      -23                 -23  =  +76
      ---                 ---     ---
       66                      ┌── 165
                               └─→   1
                                   ---
                                    66

       98                  98      98
      -87                 -87  =  +12
      ---                 ---     ---
       11                      ┌── 110
                               └─→   1
                                   ---
                                    11
```

[2]The number of digits in the subtrahend and minuend must be the same. If one number has more digits than the other, zeros are added to the left of the shorter number until it has the same number of digits.

$$\begin{array}{r} 15 \\ \underline{-37} \\ -22 \end{array} \qquad \begin{array}{r} 15 \\ \underline{-37} \\ \end{array} = \begin{array}{r} 15 \\ \underline{+62} \\ 77 \end{array}$$

no carry, so difference is negative 9s complement, which is −22

$$\begin{array}{r} 27 \\ \underline{-44} \\ -17 \end{array} \qquad \begin{array}{r} 27 \\ \underline{-44} \\ \end{array} = \begin{array}{r} 27 \\ \underline{+55} \\ 82 \end{array}$$

no carry, so difference is negative 9s complement, which is −17

Complete rules for handling signs during the subtraction process and for handling all combinations of positive and negative numbers are explained in Chap. 5. If this seems, at first, to be an unwieldy technique, note that the majority of computers now being constructed subtract by using a complemented number.

COMPLEMENTS IN OTHER NUMBER SYSTEMS

2.9 There are two types of complements for each number system. Since now only binary and BCD machines are being constructed in quantity, only these number systems are explained in any detail. The two types of complements and the rules for obtaining them are as follows:

1 *True complement* This is formed by subtracting each digit of the number from the radix minus one of the number system and then adding 1 to the least significant digit of the number formed. The true complement of a number in the decimal system is referred to as the 10s complement and in the binary system as the 2s complement.

2 *Radix-minus-one complement* The radix minus one is 9 for the decimal system and 1 for the binary system. The complement in each system is formed by subtracting each digit of the number from the radix minus one. For instance, the radix-minus-one complement of decimal 72 is 27.

BINARY NUMBER COMPLEMENTS

2.10 According to the rule in the preceding section, the 2s complement of a binary number is formed by simply subtracting each digit (bit) of the number from the radix minus one and adding a 1 to the least significant bit. Since the radix in the binary number system is 2, each bit of the binary number is subtracted from 1. The application of this rule is actually very simple; every 1 in the number is changed to a 0 and every 0 to a 1. Then a 1 is added to the least significant bit of the number formed. For instance, the 2s complement of 10110 is 01010, and the 2s complement of 11010 is 00110. Subtraction using the 2s complement system involves forming the 2s complement of the subtrahend and then adding this "true

TABLE 2.4

DECIMAL DIGIT	EXCESS-3 CODE	9s COMPLEMENT
0	0011	1100
1	0100	1011
2	0101	1010
3	0110	1001
4	0111	1000
5	1000	0111
6	1001	0110
7	1010	0101
8	1011	0100
9	1100	0011

TABLE 2.5 **2, 4, 2, 1, CODE**

	CODED BINARY			
	WEIGHT OF BIT			
DECIMAL	2	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	0	1
8	1	1	1	0
9	1	1	1	1

listed. Note the 9s complement of each code group may be formed by changing each 0 to a 1 and each 1 to a 0 in the code group.

The excess-3 code is not a weighted code, however, because weights cannot be assigned to the bits so that their sum equals the decimal digits represented.

A weighted code in which the 9s complement may be formed by complementing each binary digit is the 2, 4, 2, 1 code (see Table 2.5). If each bit of a code group is complemented, the 9s complement of the decimal digit represented is formed. For instance, 0010 (2 decimal) complemented is 1101 (7 decimal), and 1011 (5 decimal) complemented is 0100 (4 decimal). This code is widely used in instruments and electronic calculators.

The following convention is generally adopted to distinguish binary from decimal. A binary number is identified by a subscript of 2 placed at the end of the number (00110_2), and a decimal number by the subscript 10 (for instance, decimal 948 may be written 948_{10}). So we may write 0111_2 as 7_{10}. We use this convention when necessary.

OCTAL AND HEXADECIMAL NUMBER SYSTEMS

2.12 Two other number systems are very useful in the computer industry: the octal number system and the hexadecimal number system.

The octal number system has a base, or radix, of 8; eight different symbols

TABLE 2.6

OCTAL	DECIMAL	OCTAL	DECIMAL
0	0	11	9
1	1	12	10
2	2	13	11
3	3	14	12
4	4	15	13
5	5	16	14
6	6	17	15
7	7	20	16
10	8	21	17

are used to represent numbers. These are commonly 0, 1, 2, 3, 4, 5, 6, and 7. We show the first octal numbers and their decimal equivalents in Table 2.6.

To convert an octal number to a decimal number, we use the same sort of polynomial as was used in the binary case, except that we now have a radix of 8 instead of 2. Therefore, 1213 in octal is $1 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 3 \times 8^0 = 512 + 128 + 8 + 3 = 651$ in decimal. Also, 1.123 in octal is $1 \times 8^0 + 1 \times 8^{-1} + 2 \times 8^{-2} + 3 \times 8^{-3}$, or $1 + \frac{1}{8} + \frac{2}{64} + \frac{3}{512} = 1\frac{83}{512}$ in decimal.

There is a simple trick for converting a binary number to an octal number. Simply group the binary digits into groups of 3, starting at the octal point, and read each set of three binary digits according to Table 2.7.

Let us convert the binary number 011101. First, we break it into 3s (thus 011 101). Then, converting each group of three binary digits, we get 35 in octal. Therefore 011101 binary = 35 octal. Here are several more examples:

$$\begin{aligned}
111110111_2 &= 767_8\\
110110101_2 &= 665_8\\
11011_2 &= 33_8\\
1001_2 &= 11_8\\
10101.11_2 &= 25.6_8\\
1100.111_2 &= 14.7_8\\
1011.1111_2 &= 13.74_8
\end{aligned}$$

Conversion from decimal to octal can be performed by repeatedly dividing the decimal number by 8 and using each remainder as a digit in the octal number

TABLE 2.7

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

being formed. For instance, to convert 200_{10} to an octal representation, we divide as follows:

$200 \div 8 = 25$	remainder is 0
$25 \div 8 = 3$	remainder is 1
$3 \div 8 = 0$	remainder is 3

Therefore, $200_{10} = 310_8$.

Notice that when the number to be divided is less than 8, we use 0 as the quotient and the number as the remainder. Let us check this:

$$310_8 = 3_{10} \times 8_{10}^2 + 1_{10} \times 8_{10}^1 + 0_{10} \times 8_{10}^0 = 192_{10} + 8_{10} = 200_{10}$$

Here is another example. We wish to convert 3964_{10} to octal:

$3964 \div 8 = 495$	with a remainder of 4
$495 \div 8 = 61$	with a remainder of 7
$61 \div 8 = 7$	with a remainder of 5
$7 \div 8 = 0$	with a remainder of 7

Therefore, $7574_8 = 3964_{10}$. Checking, we find

$$\begin{aligned} 7574_8 &= 7_{10} \times 8_{10}^3 + 5_{10} \times 8_{10}^2 + 7_{10} \times 8_{10} + 4_{10} \\ &= 7_{10} \times 512_{10} + 5_{10} \times 64_{10} + 7_{10} \times 8_{10} + 4_{10} \times 1_{10} \\ &= 3584_{10} + 320_{10} + 56_{10} + 4_{10} \\ &= 3964_{10} \end{aligned}$$

There are several other techniques for converting octal to decimal and decimal to octal, but they are not used very frequently manually, and tables prove to be of about as much value as anything in this process. Octal-to-decimal and decimal-to-octal tables are readily available in a number of places, including the manuals distributed by manufacturers of binary machines.

An important use for octal is in listings of programs and for memory "dumps" for binary machines, thus making the printouts more compact. The manuals for several of the largest manufacturers use octal numbers to represent binary numbers because of ease of conversion and compactness.

The hexadecimal number system is useful for the same reasons. Most minicomputers and microcomputers have their memories organized into sets of *bytes*, each consisting of eight binary digits. Each byte either is used as a single entity to represent a single alphanumeric character or is broken into two 4-bit pieces. (We examine the coding of alphanumeric characters using bytes in Chap. 7.) When the bytes are handled in two 4-bit pieces, the programmer is given the option of declaring each 4-bit character as a piece of a binary number or as two BCD numbers. For instance, the byte 00011000 can be declared a binary number, in which case it is equal to 24 decimal, or as two BCD characters, in which case it represents the decimal number 18.

When the machine is handling numbers in binary but in groups of four digits,

TABLE 2.8

BINARY	HEXADECIMAL	DECIMAL
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

it is convenient to have a code for representing each of these sets of four digits. Since 16 possible different numbers can be represented, the digits 0 through 9 will not suffice; so the letters A, B, C, D, E, and F are used also (see Table 2.8).

To convert binary to hexadecimal, we simply break a binary number into groups of four digits and convert each group of four digits according to the preceding code. Thus $10111011_2 = BB_{16}$, $10010101_2 = 95_{16}$, $11000111_2 = C7_{16}$, and $10001011_2 = 8B_{16}$. The mixture of letters and decimal digits may seem strange at first, but these are simply convenient symbols, just as decimal digits are.

The conversion of hexadecimal to decimal is straightforward but time-consuming. For instance, BB represents $B \times 16^1 + B \times 16^0 = 11 \times 16 + 11 \times 1 = 176 + 11 = 187$. Similarly,

$$\begin{aligned} AB6_{16} &= 10_{10} \times 16^2_{10} + 11_{10} \times 16_{10} + 6_{10} \\ &= 10_{10} \times 256_{10} + 176_{10} + 6_{10} \\ &= 2560_{10} + 176_{10} + 6_{10} \\ &= 2742_{10} \end{aligned}$$

To convert, for instance, $3A6_{16}$ to decimal:

$$\begin{aligned} 3A6_{16} &= 3_{10} \times 16^2_{10} + 10_{10} \times 16_{10} + 6_{10} \\ &= 3_{10} \times 256_{10} + 10_{10} \times 16_{10} + 6_{10} \\ &= 768_{10} + 160_{10} + 6_{10} \\ &= 934_{10} \end{aligned}$$

Again, tables are convenient for converting hexadecimal to decimal and decimal to hexadecimal. Table 2.9 is useful for converting in either direction.

The chief use of the hexadecimal system is in connection with byte-organized machines. And since most computers are now byte-organized, a knowledge of hexadecimal is essential to using manufacturers' manuals and to reading the current literature.

TABLE 2.9 **HEXADECIMAL-TO-DECIMAL CONVERSION TABLE**

A. INTEGER CONVERSION

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15
Hexadecimal positions	4		3		2		1

EXAMPLE: 2322_{16} is $8192_{10} + 768_{10} + 32_{10} + 2_{10} = 8994.0$.

B. FRACTIONAL CONVERSION

HEX	0123 DEC	HEX	4567 DECIMAL		HEX	0123 DECIMAL			HEX	4567 DECIMAL EQUIVALENT			
.0	.0000	.00	.0000	0000	.000	.0000	0000	0000	.0000	.0000	0000	0000	0000
.1	.0625	.01	.0039	0625	.001	.0002	4414	0625	.0001	.0000	1525	8789	0625
.2	.1250	.02	.0078	1250	.002	.0004	8828	1250	.0002	.0000	3051	7578	1250
.3	.1875	.03	.0117	1875	.002	.0007	3242	1875	.0003	.0000	4577	6367	1875
.4	.2500	.04	.0156	2500	.004	.0009	7656	2500	.0004	.0000	6103	5156	2500
.5	.3125	.05	.0195	3125	.005	.0012	2070	3125	.0005	.0000	7629	3945	3125
.6	.3750	.06	.0234	3750	.006	.0014	6484	3750	.0006	.0000	9155	2734	3750
.7	.4375	.07	.0273	4375	.007	.0017	0898	4375	.0007	.0001	0681	1523	4375
.8	.5000	.08	.0312	5000	.008	.0019	5312	5000	.0008	.0001	2207	0312	5000
.9	.5625	.09	.0351	5625	.009	.0021	9726	5625	.0009	.0001	3732	9101	5625
.A	.6250	.0A	.0390	6250	.00A	.0024	4140	6250	.000A	.0001	5258	7890	6250
.B	.6875	.0B	.0429	6875	.00B	.0026	8554	6875	.000B	.0001	6784	6679	6875
.C	.7500	.0C	.0468	7500	.00C	.0029	2968	7500	.000C	.0001	8310	5468	7500
.D	.8125	.0D	.0507	8125	.00D	.0031	7382	8125	.000D	.0001	9836	4257	8125
.E	.8750	.0E	.0546	8750	.00E	.0034	1796	8750	.000E	.0002	1362	3046	8750
.F	.9375	.0F	.0585	9375	.00F	.0036	6210	9375	.000F	.0002	2888	1835	9375
Hexidecimal positions	1		2			3				4			

Quite a large number of questions have been included for this chapter. For those desiring to study octal and hexadecimal number systems further, Questions 2.58 through 2.67 contain information and exercises on octal addition and multiplication, and Questions 2.68 through 2.72 can be used to supplement the study of the hexadecimal system.