

1. INTRODUCTION

The word "computer" comes from the word "compute" which means to calculate. So a computer is normally considered to be a calculating device that can perform arithmetic operations at enormous speed.

In fact, the original objective for inventing the computer was to create a fast calculating machine. But more than 80% of the work done by computers today is of non-mathematical or non-numerical nature. Hence, to define a computer merely as calculating device is to ignore over 80% of its work.

More accurately, a computer may be defined as a device that operates upon information or data. Data can be anything like bio-data of various applicants when the computer is used for recruiting personnel, or the marks obtained by various students in various subjects when the computer is used to prepare results, or the details (name, age, sex, etc.) of various passengers when the computer is employed for making airline or railway reservations, or numbers of different types in case of application of computers for scientific research problems, etc.

Thus, data comes in various shapes and sizes

depending upon the type of computer application. A computer can store, process, and retrieve data as and when desired. The fact that computers process data is so fundamental that many people have started calling it a *data processor*.

The name data processor is more inclusive because modern computers not only compute in the usual sense but also perform other functions with the data that flow to and from them. For example, data processors may gather data from various incoming sources, merge (process of mixing or putting together) them all, sort (process of arranging in some sequence - ascending or descending) them in the desired order and finally print them in the desired format. None of these operations involve the arithmetic operations normally associated with a computing device but the term computer is often applied anyway.

CHARACTERISTICS OF COMPUTERS

1. **Speed.** A computer is a very fast device. It can perform in a few seconds the amount of work that a human being can do in an entire year - if he worked day and night and did nothing else. To put it in a different manner, a

computer does in one minute what would take a man his entire lifetime.

While talking about the speed of a computer, we do not talk in terms of seconds or even milliseconds (10^{-3}). Our units of speed are the microseconds (10^{-6}), the nanoseconds (10^{-9}), and even the picoseconds (10^{-12}). A powerful computer is capable of performing about 3 to 4 million simple arithmetic operations per second.

2. Accuracy. The accuracy of a computer is consistently high and the degree of accuracy of a particular computer depends upon its design. But for a particular computer, each and every calculation is performed with the same accuracy.

Errors can occur in a computer, but these are mainly due to human rather than technological weaknesses, that is, due to imprecise thinking by the programmer (a person who writes instructions for a computer to solve a particular problem) or due to inaccurate data.

3. Diligence. Unlike human beings, a computer is free from monotony, tiredness, lack of concentration, etc., and hence can work for hours together without creating any error and without grumbling. Due to this property, computers obviously score over human beings in doing routine type of jobs which require great accuracy. If ten million calculations have to be performed, a computer will perform the ten millionth calculation with exactly the same accuracy and speed as the first one.

4. Versatility. Versatility is one of the most wonderful things about the computer. One moment, it is preparing the results of particular examination, the next moment it is busy preparing electricity bills, and in between, it may be helping an office secretary to trace an important letter in seconds. All that is required to change its talent is to slip in a new program (a sequence of instructions for the computer) into it. Briefly, a computer is capable of performing almost any task provided that the task can be reduced to a series of logical steps.

5. Power of Remembering. As a human being acquires new knowledge, the brain subconsciously selects what it feels to be important and worth retaining in its memory, and relegates unimportant details to the back of the mind or just forgets them. With computers, this is not the case. A computer can store and recall any amount of information because of its secondary storage (a type of detachable memory) capability. Every piece of information can be retained as long as desired by the user and can be recalled as and when required. Even after several years, the information recalled will be as accurate as on the day when it was fed to the computer. A computer forgets or loses

certain information only when it is asked to do so. So it is entirely upto the user to make a computer retain or forget a particular information.

6. No I.Q. A computer is not a magical device. It can only perform tasks that a human being can. The difference is that it performs these tasks with unthinkable speed and accuracy. It possesses no intelligence of its own. Its I.Q. is zero, at least till today. It has to be told what to do and in what sequence. Hence, only the user can determine what tasks a computer will perform. A computer cannot take its own decision in this regard.

7. No Feelings. Computers are devoid of emotions. They have no feelings and no instincts because they are machine. Although men have succeeded in building a memory for the computer, but no computer possesses the equivalent of a human heart and soul. Based on our feelings, taste, knowledge, and experience, we often make certain judgements in our day to day life. But computers cannot make such judgements on their own. Their judgement is based on the instructions given to them in the form of programs that are written by us. They are only as good as man makes and uses them.

THE EVOLUTION OF COMPUTERS

Necessity is the mother of invention. The saying holds true for computers also because computers were invented as a result of man's search for fast and accurate calculating devices.

The earliest device that qualifies as a digital computer is the "abacus" also known as "soroban". This device permits the users to represent numbers by the position of beads on a rack. Simple addition and subtraction can be carried out rapidly and efficiently by positioning the beads appropriately. Although, the abacus was invented around 600 B.C., it is interesting to note that it is still used in the Far East and its users can calculate at amazing speeds.

Another manual calculating device was John Napier's bone or cardboard multiplication calculator. It was designed in the early 17th century and its upgraded versions were in use even around 1890.

The first mechanical adding machine was invented by Blaise Pascal in 1642. Later, in the year 1671, Baron Gottfried Wilhelm von Leibniz of Germany invented the first calculator for multiplication. Keyboard machines originated in the United States around 1880 and are extensively used even today. Around this period only, Herman Hollerith came up with the concept of punched

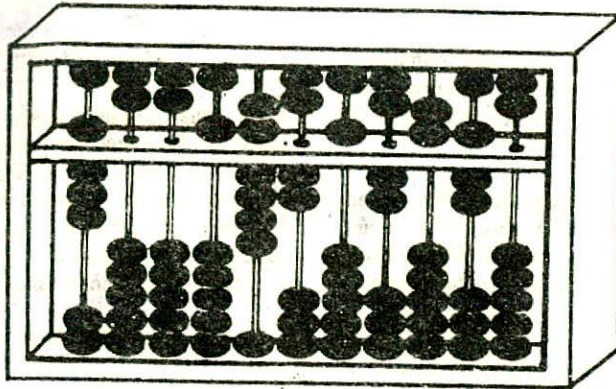


Figure 1.1. A thirteenth century abacus.

cards which are extensively used as input media in modern digital computers. Business machines and calculators made their appearance in Europe and America towards the end of the nineteenth century.

Charles Babbage, a nineteenth century Professor at Cambridge University, is considered to be the father of modern digital computers. During his period, mathematical and statistical tables were prepared by a group of clerks. Even the utmost care and precautions could not eliminate human errors. Babbage had to spend several hours checking these tables. Soon he became dissatisfied and exasperated with this type of monotonous job. The result was that he started thinking to build a machine which could compute tables guaranteed to be error-free. In this process, Babbage designed a "Difference Engine" in the year 1822 which could produce reliable tables. In 1842, Babbage came out with his new idea of Analytical Engine that was intended to be completely automatic. It was to be capable of performing the basic arithmetic functions for any mathematical problem and it was to do so at an average speed of 60 additions per minute. Unfortunately, he was unable to produce a working model of this machine mainly because the precision engineering required to manufacture the machine was not available during that period. However, his efforts established a number of principles which have been shown to be fundamental to the design of any digital computer. We will now discuss about some of the well known early computers.

THE MARK I COMPUTER (1937-44)

Also known as Automatic Sequence Controlled

calculator, this was the first fully automatic calculating machine designed by Howard A. Aiken of Harvard University in collaboration with IBM (International Business Machines) corporation. Its design was based on the techniques already developed for punched card machinery.

Although this machine proved to be extremely reliable, it was very complex in design and huge in size. It used over 3000 electrically actuated switches to control its operations and was approximately 50 feet long and 8 feet high. It was capable of performing five basic arithmetic operations : addition, subtraction, multiplication, division, and table reference. A number as big as 23 decimal digits could be used in this machine. It took approximately 0.3 second to add two numbers and 4.5 seconds for multiplication of two numbers. Hence, the machine was very slow as compared to today's computers.

It was basically an electro-mechanical device since both mechanical and electronic components were used in its design. Although its operations were not controlled electronically, Aiken's machine is often classified as computer because its instructions, which were entered by means of punched paper tape, could be altered.

THE ATANASOFF - BERRY COMPUTER (1939-42)

This electronic machine was developed by Dr. John Atanasoff to solve certain mathematical equations. It was called the Atanasoff - Berry Computer, or ABC, after its inventor's name and his assistant, Clifford Berry. It used 45 vacuum tubes for internal logic and capacitors for storage.

THE ENIAC (1943-46)

The Electronic Numerical Integrator And Calculator (ENIAC) was the first all electronic computer. It was constructed at the Moore School of Engineering of the University of Pennsylvania, U.S.A by a design team led by Professors J. Presper Eckert and John Mauchly.

ENIAC was developed as a result of military need. It took up the wall space in a 20 X 40 square feet room and used 18,000 vacuum tubes. The addition of two numbers was achieved in 200 microseconds, and multiplication in 2000 microseconds.

Although, much faster in speed as compared to Mark I computer, ENIAC had two major shortcomings : it could store and manipulate only a very limited amount of information, and its programs were wired on boards. These limitations made it difficult to detect errors and to change the programs. Hence its use was limited. However, whatever be the shortcomings of ENIAC, it represented an impressive feat of electronic engineering and was used for many years to solve ballistic problems.

THE EDVAC (1946-52)

The operation of ENIAC was seriously handicapped by the wiring board. This problem was later overcome by the new concept of "stored program" developed by Dr. John Von Neumann. The basic idea behind the stored program concept is that a sequence of instructions as well as data can be stored in the memory of the computer for the purpose of automatically directing the flow of operations. The stored program feature considerably influenced the development of modern digital computers and because of this feature we often refer to modern digital computers as stored program digital computers. The Electronic Discrete Variable Automatic Computer (EDVAC) was designed on stored program concept. Von Neumann has also got a share of the credit for introducing the idea of storing both instructions and data in the binary form (a system that uses only two digits - 0 & 1 to represent all characters) instead of the decimal numbers or human readable words.

THE EDSAC (1947-49)

Almost simultaneously with EDVAC of U.S.A., the Britishers developed the Electronic Delay Storage Automatic Calculator (EDSAC). The machine executed its first program in May 1949. In this machine, addition operation was accomplished in 1500 microseconds, and multiplication operation in 4000 microseconds. The machine was developed by a group of scientists headed by Professor Maurice Wilkes at the Cambridge University

Mathematical Laboratory.

MANCHESTER MARK I (1948)

This computer was a small experimental machine based on the stored program concept. It was designed at Manchester University by a group of scientists headed by Professor M.H.A. Newman. Its storage capacity was only 32 words, each of 31 binary digits. This was too limited to store data and instructions. Hence, the Manchester Mark I was hardly of any practical use.

THE UNIVAC I (1951)

The Universal Automatic Computer (UNIVAC) was the first digital computer which was not "one of a kind". Many UNIVAC machines were produced, the first of which was installed in the Census Bureau in 1951 and was used continuously for 10 years. The first business use of a computer, a UNIVAC I, was by General Electric Corporation in 1954.

In 1952, the International Business Machines (IBM) Corporation introduced the 701 commercial computer. In rapid succession, improved models of the UNIVAC I and other 700-series machines were introduced. In 1953, IBM produced the IBM-650 and sold over 1000 of these computers.

The commercially available digital computers, that could be used for business and scientific applications, had arrived.

THE COMPUTER GENERATIONS

"Generation" in computer talk is a step in technology. It provides a framework for the growth of the computer industry. Originally, the term 'generation' was used to distinguish between varying hardware technologies. But nowadays, it has been extended to include both the hardware and the software (see Chapter 10 for definition of hardware and software) which together make up an entire computer system.

The custom of referring to the computer era in terms of generations came into wide use only after 1964. There are totally five computer generations known till today. Each generation has been discussed below in detail along with their advantages and disadvantages. Although there is a certain amount of overlapping between the generations, the approximate dates shown against each are normally accepted.

FIRST GENERATION (1942-1955)

We have already discussed about some of the early computers - ENIAC, EDVAC, EDSAC, etc. These machines and other of their time were made possible by the invention of "vacuum tube", which was a fragile glass device that could control and amplify electronic signals. These vacuum tube computers are referred to as first-generation computers.

Advantages

1. Vacuum tubes were the only electronic components available during those days.
2. Vacuum tube technology made possible the advent of electronic digital computers.
3. These computers were the fastest calculating device of their time. They could perform computations in milliseconds.

Disadvantages

1. Too bulky in size
2. Unreliable
3. Thousands of vacuum tubes that were used emitted large amount of heat and burnt out frequently
4. Air conditioning required
5. Prone to frequent hardware failures
6. Constant maintenance required
7. Nonportable
8. Manual assembly of individual components into functioning unit required
9. Commercial production was difficult and costly
10. Limited commercial use

SECOND GENERATION (1955-1964)

The transistor, a smaller and more reliable successor to the vacuum tube, was invented in 1947. However, computers that used transistors were not produced in quantity until over a decade later. The second generation emerged with transistors being the brain of the computer.

With both the first and the second generation computers, the basic component was a discrete or separate entity. The many thousands of individual components had to be assembled by hand into functioning circuits. The manual assembly of individual components and the cost of labour involved at this assembly stage made the commercial production of these computers difficult and costly.

Advantages

1. Smaller in size as compared to first generation computers
2. More reliable
3. Less heat generated
4. These computers were able to reduce computational times from milliseconds to microseconds.
5. Less prone to hardware failures
6. Better portability
7. Wider commercial use

Disadvantages

1. Air-conditioning required
2. Frequent maintenance required
3. Manual assembly of individual components into a functioning unit was required
4. Commercial production was difficult and costly.

THIRD GENERATION (1964-1975)

Advances in electronics technology continued and the advent of "microelectronics" technology made it possible to integrate large number of circuit elements into very small (less than 5 mm square) surface of silicon known as "chips". This new technology was called "integrated circuits" (ICs). The third generation was based on IC technology and the computers that were designed with the use of integrated circuits were called third generation computers.

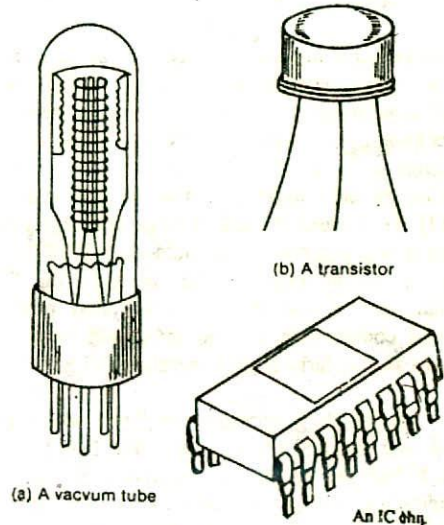


Figure 1.2. Electronics devices used for manufacturing computers of different generations.

Advantages

1. Smaller in size as compared to previous generation computers.
2. Even more reliable than second generation computers.
3. Even lower heat generated than second generation computers.
4. These computers were able to reduce computational times from microseconds to nanoseconds.
5. Maintenance cost is low because hardware failures are rare.
6. Easily portable.
7. Totally general purpose. Widely used for various commercial applications all over the world.
8. Less power requirement than previous generation computers.
9. Manual assembly of individual components into a functioning unit not required. So human labour and cost involved at assembly stage reduced drastically.
10. Commercial production was easier and cheaper.

Disadvantages

1. Air-conditioning required in many cases.
2. Highly sophisticated technology required for the manufacture of IC chips.

FOURTH GENERATION (1975 ONWARDS)

Initially, the integrated circuits contained only about ten to twenty components. This technology was named small scale integration (SSI). Later, with the advancement in technology for manufacturing ICs, it became possible to integrate upto a hundred components on a single chip. This technology came to be known as medium scale integration (MSI). Then came the era of large scale integration (LSI) when it was possible to integrate over 30,000 components onto a single chip. Effort is still on for further miniaturization and it is expected that more than one million components will be integrated on a single chip known as very large scale integration (VLSI).

A fourth generation computer, which is what we have now, has LSI chips as its brain. It is LSI technology which has led to the development of very small but extremely powerful computers. It was the start of a social revolution. A whole computer circuit was soon available on a single chip, the size of a postage stamp. Overnight computers became incredibly compact. They became inexpensive to make and suddenly it became possible for anyone and every one to own a computer.

Advantages

1. Smallest in size because of high component density
2. Very reliable
3. Heat generated is negligible
4. No air conditioning required in most cases
5. Much faster in computation than previous generations
6. Hardware failure is negligible and hence minimal maintenance is required
7. Easily portable because of their small size
8. Totally general purpose
9. Minimal labour and cost involved at assembly stage
10. Cheapest among all generations

Disadvantage

1. Highly sophisticated technology required for the manufacture of LSI chips.

FIFTH GENERATION (YET TO COME)

Scientists are now at work on the fifth generation computers - a promise, but not yet a reality. They aim to bring us machines with genuine I.Q., the ability to reason logically, and with real knowledge of the world. Thus, unlike the last four generations which naturally followed its predecessor, the fifth generation will be totally different, totally novel, totally new.

In structure it will be parallel (the present ones are serial) and will be able to do multiple tasks simultaneously. In functions, it will not be algorithmic (step by step, with one step at a time). In nature, it will not do just data processing (number crunching) but knowledge processing. In inference, it will not be merely deductive, but also inductive. In application, it will behave like an expert. In programming, it will interact with humans in ordinary language (unlike BASIC, COBOL, FORTRAN, etc. which present computers need). And in architecture, it will have KIPS (Knowledge Information Processing System) rather than the present DIPS/LIPS (Data/Logic Information Processing System).

The odds of coming out with a fifth generation computer are heaviest for Japan. They have already started work in this direction few years back. Japan has chosen the PROLOG (Programming in Logic) language as its operating software and plans to have the final machine talk with human beings, see and deliver pictures and hear the normal, natural language.

QUESTIONS

1. What is a computer ? Why is it also known as a data processor ?
2. List out and explain some of the important characteristics of a computer.
3. What is an abacus ?
4. Who is known as the father of modern digital computers and why ?
5. Why are modern digital computers often referred to as stored program digital computers ?
6. Give the full form of the following abbreviations used in computer terminology :
IBM, ENIAC, EDVAC, EDSAC, UNIVAC.
7. What is meant by 'generation' in computer terminology ? How many computer generations are there till now ?
8. List out the various computer generations along with their basic characteristics.
9. Write a short note on fifth generation computers.
10. What are the advantages of transistors over vacuum tubes ?
11. What is an IC ? How does it help in reducing the size of computers ?
12. List out some of the advantages of IC technology over transistor technology.
13. Give the full form of the following abbreviations used in computer terminology :
IC, SSI, MSI, LSI, VLSI, DIPS, LIPS, PROLOG.

2. BASIC COMPUTER ORGANIZATION

All computer systems perform the following five basic operations :

1. **Inputting.** The process of entering data and instructions into the computer system.
2. **Storing.** Saving data and instructions so that they are available for initial or for additional processing as and when required.
3. **Processing.** Performing arithmetic operations or logical operations (comparisons like equal to, less than, greater than, etc.) on data in order to convert them into useful information.
4. **Outputting.** The process of producing useful information or results for the user, such as a printed report or visual display.

5. **Controlling.** Directing the manner and sequence in which all of the above operations are performed.

The goal of this chapter is to familiarize you with the computer system units that perform these functions. This chapter will provide you with an overview of computer systems as they are viewed by computer system architects. It is an introduction to chapters 7, 8 and 9 which describe the major units and their functions in more detail.

The internal architectural design of computers differs from one system model to another. However, the basic organization remains the same for all computer systems. A block diagram of the basic computer organization is shown in Figure 2.1. In this figure, the solid lines are used to indicate the flow of instruction and data, and the dotted lines represent the control exercised by the control unit. It displays the five major building blocks, or

functional units, of a digital computer system. These five units correspond to the five basic operations performed by all computer systems. The function of each of these units is described below:

INPUT UNIT

Data and instructions must enter the computer system before any computation can be performed on the supplied data. This task is performed by the input unit that links the external environment with the computer system. Data and instructions enter input units in forms that depend upon the particular device used. For example, data is entered from a keyboard in a manner similar to typing, and this differs from the way in which data is entered through a card reader which is another type of input device. However, regardless of the form in which they receive their inputs, all input devices must provide a computer with data that are transformed into the binary codes that the primary memory of a computer is designed to accept. This transformation is accomplished by units called *input interfaces*. Input interfaces are designed to match the unique physical or electrical characteristics of input devices to the requirements of the computer system.

In short, the following functions are performed by an input unit :

1. It accepts (or reads) the list of instructions and data from the outside world.
2. It converts these instructions and data in computer acceptable form.
3. It supplies the converted instructions and data to the computer system for further processing.

OUTPUT UNIT

The job of an output unit is just the reverse of that of an input unit. It supplies information and results of computation to the outside world. Thus it links the computer with the external environment. As computers work with binary code, the results produced are also in the binary form. Hence, before supplying the results to the outside world, it must be converted to human acceptable (readable) form. This task is accomplished by units called *output interfaces*. Output interfaces are designed to match the unique physical or electrical characteristics of output devices (terminals, printers, etc.) to the requirements of the external environment.

In short, the following functions are performed by an output unit :

1. It accepts the results produced by the computer which are in coded form and hence cannot be easily understood by us.
2. It converts these coded results to human acceptable (readable) form.
3. It supplies the converted results to the outside world.

STORAGE UNIT

The data and instructions that are entered into the computer system through input units have to be stored inside the computer before the actual processing starts. Similarly, the results produced by the computer after processing must also be kept somewhere inside the computer system before being passed on to the output units. Moreover, the intermediate results produced by the computer must also be preserved for ongoing processing. The storage unit or the primary/main storage of a computer system is designed to cater to all these needs. It provides space for storing data and instructions, space for intermediate results, and also space for the final results.

In short, the specific functions of the storage unit are to hold (store) :

1. All the data to be processed and the instructions required for processing (received from input devices).
2. Intermediate results of processing.
3. Final results of processing before these results are released to an output device.

ARITHMETIC LOGIC UNIT

The arithmetic logic unit (ALU) of a computer system is the place where the actual execution of the instructions takes place during the processing operation. To be more precise, all calculations are performed and all comparisons (decisions) are made in the ALU. The data and instructions, stored in the primary storage prior to processing, are transferred as and when needed to the ALU where processing takes place. No processing is done in the primary storage unit. Intermediate results generated in the ALU are temporarily transferred back to the primary storage until needed at a later time. Data may thus move from primary storage to ALU and back again to storage many times before the processing is over. After the completion of processing, the final results which are stored in the storage unit are released to an output device.

The type and number of arithmetic and logic operations that a computer can perform is determined by the engineering design of the ALU. However, almost all

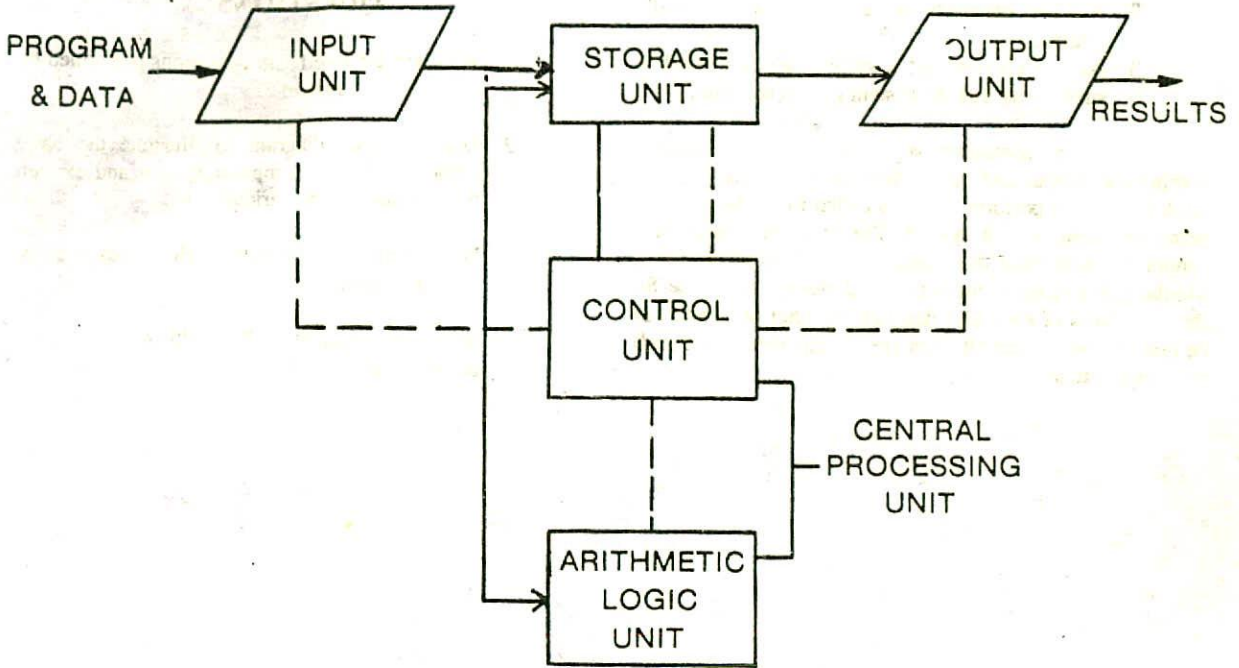


Figure 2.1. Basic organisation of a computer system

ALU's are designed to perform the four basic arithmetic operations - add, subtract, multiply, divide and logic operations or comparisons such as less than, equal to, or greater than.

CONTROL UNIT

How does the input device know that it is time for it to feed data into the storage unit? How does the ALU know what should be done with the data once they are received?

And how is it that only the final results are sent to the output device and not the intermediate results? All this is possible because of the control unit of the computer system. By selecting, interpreting, and seeing to the execution of the program instructions, the control unit is able to maintain order and direct the operation of the entire system. Although, it does not perform any actual processing on the data, the control unit acts as a central nervous system for the other components of the computer. It manages and coordinates the entire computer system. It obtains instructions from the program stored in main memory, interprets the instructions, and issues signals that cause other units of the system to execute them.

CENTRAL PROCESSING UNIT

The control unit and the arithmetic logic unit of a computer system are jointly known as the Central Processing Unit (CPU). The CPU is the brain of any computer system. In a human body, all major decisions are taken by the brain and the other parts of the body function as directed by the brain. Similarly, in a computer system, all major calculations and comparisons are made inside the CPU and the CPU is also responsible for activating and controlling the operations of other units of a computer system.

THE SYSTEM CONCEPT

You might have observed by now that we have been referring to a computer as a system (computer system). What can be the reason behind this? To know the answer let us first consider the definition of a system.

A *system* is a group of integrated parts that have the common purpose of achieving some objective(s). So, the following three characteristics are key to a system:

1. A system has more than one element.

2. All the elements of a system are logically related.
3. All the elements of a system are controlled in such a way that the system goal is achieved.

Since a computer is made up of integrated components (input and output devices, storage, CPU) that work together to perform the steps called for in the program being executed, it is a system. The input or output units cannot function until they receive signals from the CPU. Similarly, the storage unit or the CPU alone is of no use. So the usefulness of each unit depends on other units and can be realized only when all units are put together (integrated) to form a system.

QUESTIONS

1. What are the five basic operations performed by any computer system ?
2. Draw a block diagram to illustrate the basic organisation of a computer system and explain the functions of the various units.
3. What is an input interface ? How does it differ from an output interface ?
4. What is a system ? Why do we refer to a computer as a system ?

3. NUMBER SYSTEMS

We have already seen, in the previous chapter that inside a computer system, data is stored in a format that cannot be easily read by human beings. This is the reason why input and output (I/O) interfaces are required. Every computer stores numbers, letters, and other special characters in a coded form. Before going into the details of these codes, it is essential to have a basic understanding of the number system. So the goal of this chapter is to familiarize you with the basic fundamentals of number system. It also introduces some of the commonly used number systems by computer professionals and the relationship between them.

Number systems are basically of two types: non-positional and positional.

NON-POSITIONAL NUMBER SYSTEMS

In early days, human beings counted on fingers. When ten fingers were not adequate, stones, pebbles, or sticks were used to indicate values. This method of counting uses an additive approach or the non-positional number system. In this system, we have symbols such as I for 1, II for 2, III for 3, IIII for 4, IIIII for 5, etc. Each

symbol represents the same value regardless of its position in the number and the symbols are simply added to find out the value of a particular number. Since it is very difficult to perform arithmetic with such a number system, positional number systems were developed as the centuries passed.

POSITIONAL NUMBER SYSTEMS

In a positional number system, there are only a few symbols called digits, and these symbols represent different values depending on the position they occupy in the number. The value of each digit in such a number is determined by three considerations :

1. the digit itself,
2. the position of the digit in the number, and
3. the base of the number system (where base is defined as the total number of digits available in the number system).

The number system that we use in our day-to-day life is called the *Decimal number system*. In this system, the base is equal to 10 because there are altogether ten symbols or digits (0,1,2,3,4,5,6,7,8,9) used in this system. You know that in the decimal system, the successive positions to the

left of the decimal point represent units, tens, hundreds, thousands, etc. But you may not have given much attention to the fact that each position represents a specific power of the base (10). For example, the decimal number 2586 (written as 2586_{10}) consists of the digit 6 in the units position, 8 in the tens position, 5 in the hundreds position, and 2 in the thousands position and its value can be written as:

$$(2 \times 1000) + (5 \times 100) + (8 \times 10) + (6 \times 1)$$

$$\text{or } 2000 + 500 + 80 + 6$$

$$\text{or } 2586$$

It may also be observed that the same digit signifies different values depending on the position it occupies in the number. For example,

$$\text{In } 2586_{10} \text{ the digit 6 signifies } 6 \times 10^0 = 6$$

$$\text{In } 2568_{10} \text{ the digit 6 signifies } 6 \times 10^1 = 60$$

$$\text{In } 2658_{10} \text{ the digit 6 signifies } 6 \times 10^2 = 600$$

$$\text{In } 6258_{10} \text{ the digit 6 signifies } 6 \times 10^3 = 6000$$

Thus any number can be represented by using the available digits and arranging them in various positions.

The principles that apply to the decimal system apply in any other positional number system. It is important only to keep track of the base of the number system in which we are working.

There are two characteristics of all number systems that are suggested by the value of the base. In all number systems, the value of the base determines the total number of different symbols or digits available in the number system. The first of these choices is always zero. The second characteristic is that the maximum value of a single digit is always equal to one less than the value of the base.

Some of the number systems commonly used in computer design and by computer professionals are discussed below.

BINARY NUMBER SYSTEM

The binary number system is exactly like the decimal system except that the base is 2 instead of 10. We have only two symbols or digits (0 and 1) that can be used in this number system. Note that the largest single digit is 1 (one less than the base). Again, each position in a binary

number represents a power of the base (2). As such, in this system, the rightmost position is the units (2^0) position, the second position from the right is the 2's (2^1) position and proceeding in this way we have 4's (2^2) position, 8's (2^3) position, 16's (2^4) position, and so on. Thus, the decimal equivalent of the binary number 10101 (written as 10101_2) is

$$(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

$$\text{or } 16 + 0 + 4 + 0 + 1$$

$$\text{or } 21$$

In order to be specific about which system we are referring to, it is common practice to indicate the base as a subscript. Thus we write:

$$10101_2 = 21_{10}$$

"Binary digit" is often referred to by the common abbreviation *bit*. Thus, a "bit" in computer terminology means either a 0 or a 1. A binary number consisting of n bits is called an n -bit number. Table 3.1 lists all the 3-bit numbers along with their decimal equivalent. Remember that we have only two digits, 0 and 1, in the binary system, and hence the binary equivalent of the decimal number 2 has to be stated as 10 (read as one, zero). Another important point to note is that with 3 bits (positions), only 8 (2^3) different patterns of 0's and 1's are possible and from Table 3.1 it may be seen that a 3-bit number can have one of the 8 values in the range 0 to 7. In fact, it can be shown that any decimal number in the range 0 to $2^n - 1$ can be represented in the binary form as an n -bit number.

Every computer stores numbers, letters, and other special characters in binary form. There are several occasions when computer professionals have to know the raw data contained in a computer's memory. A common way of looking at the contents of a computer's memory is to print out the memory contents on the line printer. This print out is called a *memory dump*. If memory dumps were to be printed using binary numbers, the computer professionals would be confronted with many pages of 0s and 1s. Working with these numbers would be very difficult and error prone.

Because of the quantity of printout that would be required in a memory dump of binary digits and the lack of digit variety (0s and 1s only), two number systems, *octal* and *hexadecimal*, are used as shortcut notation for binary. These number systems and their relationship with the binary number system will now be explained in this chapter.

Table 3.1. 3-bit Numbers With Their Decimal Values.

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

OCTAL NUMBER SYSTEM

In the octal number system the base is 8. So in this system there are only eight symbols or digits: 0, 1, 2, 3, 4, 5, 6 and 7 (8 and 9 do not exist in this system). Here also the largest single digit is 7 (one less than the base). Again, each position in an octal number represents a power of the base (8). Thus the decimal equivalent of the octal number 2057 (written as 2057_8) is:

$$(2 \times 8^3) + (0 \times 8^2) + (5 \times 8^1) + (7 \times 8^0)$$

$$\text{or } 1024 + 0 + 40 + 7$$

$$\text{or } 1071$$

$$\text{So we have } 2057_8 = 1071_{10}.$$

Observe that since there are only 8 digits in the octal number system, so 3 bits ($2^3 = 8$) are sufficient to represent any octal number in binary (see Table 3.1).

HEXADECIMAL NUMBER SYSTEM

The hexadecimal number system is one with a base of 16. The base of 16 suggests choices of 16 single-character digits or symbols. The first 10 digits are the digits of a decimal system 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The remaining six digits are denoted by A, B, C, D, E, F representing the decimal values 10, 11, 12, 13, 14, 15 respectively. In the hexadecimal number system, therefore, the letters A through F are number digits. The number A has a decimal equivalent value of 10 and the hexadecimal F has a decimal equivalent value of 15. Thus, largest single digit is F or 15 (one less than the base). Again, each position in a hexadecimal system represents a power of the base (16). Thus the decimal equivalent of the hexadecimal number 1AF (written as $1AF_{16}$) is:

$$(1 \times 16^2) + (A \times 16^1) + (F \times 16^0)$$

$$\text{or } (1 \times 256) + (10 \times 16) + (15 \times 1)$$

$$\text{or } 256 + 160 + 15$$

$$\text{or } 431$$

$$\text{Thus } 1AF_{16} = 431_{10}$$

Observe that since there are only 16 digits in the hexadecimal number system, so 4 bits ($2^4 = 16$) are sufficient to represent any hexadecimal number in binary.

CONVERTING FROM ONE NUMBER SYSTEM TO ANOTHER

Numbers expressed in decimal are much more meaningful to us than are values expressed in any other number system. This is mainly because of the fact that we have been using decimal numbers in our day-to-day life right from childhood. However, any number value in one number system can be represented in any other number system. Because the input and the final output values are to be in decimal, computer professionals are often required to convert numbers in other number systems to decimal and vice-versa. There are many methods or techniques that can be used to convert numbers from one base to another. We will see one technique used in converting to base 10 from any other base and a second technique to be used in converting from base 10 to any other base.

CONVERTING TO DECIMAL FROM ANOTHER BASE

The following three steps are used to convert to a base 10 value from any other number system:

- Step 1: Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).
- Step 2: Multiply the obtained column values (in Step 1) by the digits in the corresponding columns.
- Step 3: Sum the products calculated in Step 2. The total is the equivalent value in decimal.

Example 3.1. $11001_2 = ?_{10}$

Solution:

Step 1: Determine column values

Column Number (from right)	Column Value
1	$2^0 = 1$
2	$2^1 = 2$
3	$2^2 = 4$
4	$2^3 = 8$
5	$2^4 = 16$

Step 2: Multiply column values by corresponding column digits

$$\begin{array}{r}
 16 \ 8 \ 4 \ 2 \ 1 \\
 \times 1 \ \times 1 \ \times 0 \ \times 0 \ \times 1 \\
 \hline
 16 \ 8 \ 0 \ 0 \ 1
 \end{array}$$

Step 3: Sum the products

$$16 + 8 + 0 + 0 + 1 = 25$$

Hence, $11001_2 = 25_{10}$

Example 3.2. $4706_8 = ?_{10}$

Solution:

Step 1: Column Number (from right)	Column Value
1	$8^0 = 1$
2	$8^1 = 8$
3	$8^2 = 64$
4	$8^3 = 512$

$$\begin{array}{r}
 \text{Step 2:} \\
 512 \quad 64 \quad 8 \quad 1 \\
 \times 4 \quad \times 7 \quad \times 0 \quad \times 6 \\
 \hline
 2048 \quad 448 \quad 0 \quad 6
 \end{array}$$

Step 3: $2048 + 448 + 0 + 6 = 2502$

Hence, $4706_8 = 2502_{10}$

Example 3.3. $1AC_{16} = ?_{10}$

Solution:

$$\begin{aligned}
 1AC_{16} &= 1 \times 16^2 + A \times 16^1 + C \times 16^0 \\
 &= 1 \times 256 + 10 \times 16 + 12 \times 1 \\
 &= 256 + 160 + 12 \\
 &= 428_{10}
 \end{aligned}$$

Example 3.4. $4052_7 = ?_{10}$

Solution:

$$\begin{aligned}
 4052_7 &= 4 \times 7^3 + 0 \times 7^2 + 5 \times 7^1 + 2 \times 7^0 \\
 &= 4 \times 343 + 0 \times 49 + 5 \times 7 + 2 \times 1 \\
 &= 1372 + 0 + 35 + 2 \\
 &= 1409_{10}
 \end{aligned}$$

Example 3.5. $4052_6 = ?_{10}$

Solution:

$$\begin{aligned}
 4052_6 &= 4 \times 6^3 + 0 \times 6^2 + 5 \times 6^1 + 2 \times 6^0 \\
 &= 4 \times 216 + 0 \times 36 + 5 \times 6 + 2 \times 1 \\
 &= 864 + 0 + 30 + 2 \\
 &= 896_{10}
 \end{aligned}$$

Comparing this result with the result obtained in Example 3.4, we find that although the digits (4052) are same for both the numbers, but their decimal equivalent is different. This is because of the fact that the number in Example 3.4 is represented in base 7 number system whereas the number in Example 3.5 is represented in base 6 system.

Example 3.6. $11001_4 = ?_{10}$

Solution :

$$\begin{aligned} 11001_4 &= 1 \times 4^4 + 1 \times 4^3 + 0 \times 4^2 + 0 \times 4^1 + 1 \times 4^0 \\ &= 1 \times 256 + 1 \times 64 + 0 \times 16 + 0 \times 4 + 1 \times 1 \\ &= 256 + 64 + 0 + 0 + 1 \\ &= 321_{10} \end{aligned}$$

Compare the result with Example 3.1.

Example 3.7. $1AC_{13} = ?_{10}$

Solution :

$$\begin{aligned} 1AC_{13} &= 1 \times 13^2 + A \times 13^1 + C \times 13^0 \\ &= 1 \times 169 + 10 \times 13 + 12 \times 1 \\ &= 311_{10} \end{aligned}$$

Compare the result with Example 3.3.

CONVERTING FROM BASE 10 TO A NEW BASE (DIVISION - REMAINDER TECHNIQUE)

The following four steps are used to convert a number from base 10 to a new base:

- Step 1 :** Divide the decimal number to be converted by the value of the new base.
- Step 2 :** Record the remainder from step 1 as the rightmost digit (least significant digit) of the new base number.
- Step 3 :** Divide the quotient of the previous divide by the new base.
- Step 4 :** Record the remainder from step 3 as the next digit (to the left) of the new base number.

Repeat steps 3 and 4, recording remainders from right to left, until the quotient becomes zero in step 3. Note that the last remainder thus obtained will be the most

significant digit (MSD) of the new base number.

Example 3.8 $25_{10} = ?_2$

Solution :

- Steps 1 & 2: $25/2 = 12$ and remainder 1
 Steps 3 & 4: $12/2 = 6$ and remainder 0
 Steps 3 & 4: $6/2 = 3$ and remainder 0
 Steps 3 & 4: $3/2 = 1$ and remainder 1
 Steps 3 & 4: $1/2 = 0$ and remainder 1

As mentioned in Steps 2 & 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the least significant digit (LSD) and the last remainder becomes the most significant digit (MSD).

Hence, $25_{10} = 11001_2$

Compare the result with Example 3.1.

Example 3.9. $42_{10} = ?_2$

Solution :

2	42	Remainders

	21	0

	10	1

	5	0

	2	1

	1	0

	0	1

Hence, $42_{10} = 101010_2$

Example 3.10. $952_{10} = ?_8$

Solution :

8	952	Remainders
	119	0
	14	7
	1	6
	0	1

Hence, $952_{10} = 1670_8$ Example 3.11. $428_{10} = ?_{16}$

Solution :

16	428	Remainders in hexadecimal.
	26	12 = C
	1	10 = A
	0	1 = 1

Hence, $428_{10} = 1AC_{16}$

Compare the result with Example 3.3.

Example 3.12. $100_{10} = ?_5$

Solution :

5	100	Remainders
	20	0
	4	0
	0	4

Hence, $100_{10} = 400_5$ Example 3.13. $100_{10} = ?_4$

Solution :

4	100	Remainders

25	0
	6
	1
	0

Hence, $100_{10} = 1210_4$

Compare this result with the result obtained in Example 3.12.

Example 3.14. $1715_{10} = ?_{12}$

Solution:

12	1715	Remainders in Base 12
	142	11 = B
	11	10 = A
	0	11 = B

Hence, $1715_{10} = BAB_{12}$

CONVERTING FROM A BASE OTHER THAN 10 TO A BASE OTHER THAN 10

The following two steps are used to convert a number from a base other than 10 to a base other than 10.

Step 1: Convert the original number to a decimal number (base 10).

Step 2: Convert the decimal number so obtained to the new base.

Example 3.15. $545_6 = ?_4$

Solution :

Step 1: Convert from base 6 to base 10
 $545 = 5 \times 6^2 + 4 \times 6^1 + 5 \times 6^0$
 $= 5 \times 36 + 4 \times 6 + 5 \times 1$
 $= 180 + 24 + 5$

$$= 209_{10}$$

Step 2 : Convert 209_{10} to base 4

4	209	Remainders
	52	1
	13	0
	3	1
	0	3

Hence, $209_{10} = 3101_4$

So, $545_6 = 209_{10} = 3101_4$

Thus, $545_6 = 3101_4$

Example 3.16. $101110_2 = ?_8$

Solution :

Step 1 : Convert 101110_2 to base 10

$$\begin{aligned} 101110_2 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 32 + 0 + 8 + 4 + 2 + 0 \\ &= 46_{10} \end{aligned}$$

Step 2 : Convert 46_{10} to base 8.

8	46	Remainders
	5	6
	0	5

Hence, $46_{10} = 56_8$

So, $101110_2 = 46_{10} = 56_8$

Thus, $101110_2 = 56_8$

Example 3.17. $11010011_2 = ?_{16}$

Solution :

Step 1 : Convert 11010011_2 to base 10

$$11010011_2 = 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 1 \times 128 + 1 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$$

$$= 128 + 64 + 0 + 16 + 0 + 0 + 2 + 1$$

$$= 211_{10}$$

Step 2 : Convert 211_{10} to base 16

16	211	Remainders
	13	3 = 3 in Hexadecimal
	0	13 = D in Hexadecimal

Hence, $211_{10} = D3_{16}$

So, $11010011_2 = 211_{10} = D3_{16}$

Thus, $11010011_2 = D3_{16}$

Example 3.16 illustrates the method of converting a number from binary to octal. Similarly, Example 3.17 shows how to convert a number from binary to hexadecimal. However, these are lengthy procedures and shortcut methods can be used when we desire such conversions. We will now discuss these shortcut methods.

SHORTCUT METHOD FOR BINARY TO OCTAL CONVERSION

The following steps are used in this method :

Step 1 : Divide the binary digits into groups of three (starting from the right).

Step 2 : Convert each group of three binary digits into one octal digit. (Refer to Table 3.1 and try to remember that since there are only 8 digits (0 to 7) in the octal number system, so 3 bits ($2^3 = 8$) are sufficient to represent any octal number in binary). Since decimal digits 0 to 7 are equal to octal digits 0 to 7 so binary to decimal conversion can be used in this step.

Example 3.22. $11010011_2 = ?_{16}$

Solution :

Step 1 : Divide the binary digits into groups of 4.

$$\underline{1101} \quad \underline{0011}$$

Step 2 : Convert each group of 4 binary digits to 1 hexadecimal digit.

$$\begin{aligned} 1101_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 8 + 4 + 0 + 1 \\ &= 13_{10} \\ &= D_{16} \end{aligned}$$

$$\begin{aligned} 0011_2 &= 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 0 + 0 + 2 + 1 \\ &= 3_{16} \end{aligned}$$

Hence, $11010011_2 = D3_{16}$

Compare the result with the result of Example 3.17.

Example 3.23. $10110101100_2 = ?_{16}$

Solution :

$$10110101100_2 = \underline{0101} \quad \underline{1010} \quad \underline{1100}$$

(Group 4 digits from right)

$$\begin{aligned} &= 5 \text{ A C} \\ &\text{(Convert each group to a hexadecimal digit)} \end{aligned}$$

Hence, $10110101100_2 = 5AC_{16}$

SHORTCUT METHOD FOR HEXADECIMAL TO BINARY CONVERSION

The following steps are used in this method :

Step 1 : Convert the decimal equivalent of each hexadecimal digit to 4 binary digits.

Step 2 : Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Example 3.24. $2AB_{16} = ?_2$

Solution :

Step 1 : Convert the decimal equivalent of each hexadecimal digit to 4 binary digits

$$2_{16} = 2_{10} = 0010_2$$

$$A_{16} = 10_{10} = 1010_2$$

$$B_{16} = 11_{10} = 1011_2$$

Step 2 : Combine the binary groups

$$2AB_{16} = \underline{0010} \quad \underline{1010} \quad \underline{1011}$$

2 A B

Hence, $2AB_{16} = 001010101011_2$

Example 3.25. $ABC_{16} = ?_2$

Solution :

$$ABC_{16} = \underline{1010} \quad \underline{1011} \quad \underline{1100}$$

A B C

$$= 101010111100_2$$

Hence, $ABC_{16} = 101010111100_2$

Finally, Table 3.2 summarises the relationship between the decimal, binary, hexadecimal, and octal number systems. Note that the maximum value for a single digit of octal (7) is equal to the maximum value of three digits of binary. The value range of one digit of octal duplicates the value range of three digits of binary. If octal digits are substituted for binary digits, the substitution is on a one-to-three basis. Thus, computers that print octal numbers instead of binary, while taking memory dump, save one-third of the printing space and time.

Similarly, note that the maximum value of one digit in hexadecimal is equal to the maximum value of four digits in binary. Thus the value range of one digit of hexadecimal is equivalent to the value range of four digits of binary. Therefore, hexadecimal shortcut notation is a one-to-four reduction in the space and time required for memory dump.

FRACTIONAL NUMBERS

In binary number system, fractional numbers are formed in the same general way as in the decimal system. Just as in the decimal system,

$$0.235 = (2 \times 10^{-1}) + (3 \times 10^{-2}) + (5 \times 10^{-3})$$

and

$$68.53 = (6 \times 10^1) + (8 \times 10^0) + (5 \times 10^{-1}) + (3 \times 10^{-2})$$

Similarly in the binary system,

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

and

$$10.01 = (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})$$

Thus, the binary point serves the same purpose as the decimal point. Some of the positional values in the binary system are given below.

					Binary Point ↓					
Position	4	3	2	1	0	.	-1	-2	-3	-4
Position Value	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
Quantity Represented	16	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

In general, a number in a number system with base b would be written as :

$$a_n a_{n-1} \dots a_0 . a_{-1} a_{-2} \dots a_{-m}$$

and would be interpreted to mean

$$a_n \times b^n + a_{n-1} \times b^{n-1} + \dots + a_0 \times b^0 + a_{-1} \times b^{-1} + a_{-2} \times b^{-2} + \dots + a_{-m} \times b^{-m}$$

The symbols $a_n, a_{n-1}, \dots, a_{-m}$ used in the above representation should be one of the b symbols allowed in the number system.

Thus, as per the above mentioned general rule,

$$46.32_8 = (4 \times 8^1) + (6 \times 8^0) + (3 \times 8^{-1}) + (2 \times 8^{-2})$$

and

$$5A.3C_{16} = (5 \times 16^1) + (A \times 16^0) + (3 \times 16^{-1}) + (C \times 16^{-2})$$

Example 3.26.

Find the decimal equivalent of the binary number 110.101

Solution :

$$\begin{aligned} 110.101_2 &= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} \\ &\quad + 1 \times 2^{-3} \\ &= 4 + 2 + 0 + .5 + 0 + .125 \\ &= 6 + 0.5 + 0.125 \\ &= 6.625_{10} \end{aligned}$$

Example 3.27.

Find the decimal equivalent of the octal number 127.54

Solution :

$$\begin{aligned} 127.54_8 &= 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 5 \times 8^{-1} + 4 \times 8^{-2} \\ &= 64 + 16 + 7 + 5/8 + 4/64 \\ &= 87 + 0.625 + 0.0625 \\ &= 87.6875_{10} \end{aligned}$$

Example 3.28.

Find the decimal equivalent of the hexadecimal number 2B.C4

Solution :

$$\begin{aligned} 2B.C4_{16} &= 2 \times 16^1 + B \times 16^0 + C \times 16^{-1} + 4 \times 16^{-2} \\ &= 32 + B + C/16 + 4/256 \\ &= 43 + 0.75 + 0.015625 \\ &= 43.765625_{10} \end{aligned}$$

Table 3.2. Relationship between Decimal, Binary, Hexadecimal and Octal Number Systems

Decimal	Hexa Decimal	Binary	Octal
0	0	0	0
1	1	1	1
2	2	10	2
3	3	11	3
4	4	100	4
5	5	101	5
6	6	110	6
7	7	111	7
8	8	1000	10
9	9	1001	
10	A	1010	
11	B	1011	
12	C	1100	
13	D	1101	
14	E	1110	
15	F	1111	
16	10	10000	

QUESTIONS

- What is the difference between a positional and a non-positional number system? Give examples of both types of number systems.
- What is meant by the base of a number system? Give examples to illustrate the role of base in positional number systems.
- What is the value of the base for decimal, hexadecimal, binary and octal number systems?
- Give an example for octal number system to show that the same digit may signify different values depending on the position it occupies in the number.
- What will be the total number of different symbols or digits and the maximum value of a single digit for the following number systems:
 - number system with base 5.
 - number system with base 20.
 - number system with base 9.
 - number system with base 12.
- What is a bit in computer terminology? How many different patterns of bits are possible with
 - 5 bits
 - 7 bits
 - 8 bits?
- Explain the meaning of the term "memory dump".
- Why are octal and/or hexadecimal number systems used as shortcut notation?
- Find out the decimal equivalents of the following binary numbers.

a) 1101011	b) 11010
c) 10110011	d) 11011101
e) 1110101	f) 1000
g) 10110001100	
h) 1010101100	
i) 110001	j) 111
- Find out the octal equivalents of the binary numbers of Question 9.
- Find out the hexadecimal equivalents of the binary numbers of Question 9.

12. Convert the following numbers to decimal :

a) 110110_2 b) 2573_6

c) $2A3B_{16}$ d) 1234_9

13. Convert the following decimal numbers to binary :

a) 435_{10} b) 1694_{10}

c) 32_{10} d) 135_{10}

14. Convert the decimal numbers of Question 13 to octal.

15. Convert the decimal numbers of Question 13 to hexadecimal.

16. a) $125_6 = ?_4$

b) $24_9 = ?_3$

c) $ABC_{16} = ?_8$

17. Convert the following numbers to their binary equivalent :

a) $2AC_{16}$ b) FAB_{16}

c) 261_48 d) 562_8

18. Find the decimal equivalent of the following numbers

a) 111.01_2 b) 1001.011_2

c) 247.65_8 d) $A2B.D4_{16}$

4. COMPUTER CODES

In the previous chapter, we have discussed about true or "pure" binary numbers. In this chapter, we will see how these binary numbers are coded to represent characters in the computer memory. Thus, the goal of this chapter is to present the formats used in computer memory to record data. Although many coding schemes have been developed over the years, we will be discussing only the most commonly used computer codes.

Numeric data is not the only form of data that is to be handled by a computer. We often require to process alphanumeric data also. An alphanumeric data is a string of symbols where a symbol may be one of the letters A,B,C,...,Z or one of the digits 0,1,2,...,9 or a special character, such as + - * / , . () = (space or blank) etc. An alphabetic data consists of only the letters A,B,C,...,Z and the blank character. Similarly, numeric data consists of only numbers 0,1,2,...,9. However, any data must be represented internally by the bits 0 & 1. As such, binary coding schemes are used to represent data internally in the computer memory. In binary coding, every symbol that appears in the data is represented by a group of bits. The group of bits used to represent a symbol is called a byte. To indicate the number of bits in a group, sometimes a byte is

referred to as "n-bit byte" where the group contains n bits. However, the term byte is commonly used to mean an 8-bit byte (a group of 8 bits) because most of the modern computers use 8 bits to represent a symbol.

BCD CODE

The Binary Coded Decimal (BCD) code is one of the early memory codes. It is based on the idea of converting each digit of a decimal number into its binary equivalent rather than converting the entire decimal value into a pure binary form. This facilitates the conversion process to a great extent.

The BCD equivalent of each decimal digit is shown in Table 4.1. Since 8 and 9 require 4 bits, all decimal digits are represented in BCD by 4 bits. You have seen in Example 3.9 that 42_{10} is equal to 101010_2 in a pure binary form.

Converting 42_{10} into BCD, however, produces the following result :

$$42_{10} = \begin{array}{cc} \underline{0100} & \underline{0010} \\ 4 & 2 \end{array}$$

or 01000010 in BCD

Note that each decimal digit is independently converted to a 4 bit binary number and hence the conversion process is very easy. Also note that when 4 bits are used, altogether 16 (2^4) configurations are possible (refer to hexadecimal number system). But from Table 4.1 you can see that only the first 10 of these combinations are used to represent decimal digits. The remaining 6 arrangements (1010,

Table 4.1. BCD Equivalent of Decimal Digits

Decimal Digits	BCD Equivalent
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

1011, 1100, 1101, 1110 and 1111) have decimal values from 10 to 15. These arrangements are not used in BCD coding. That is, 1010 does not represent 10_{10} in BCD. Instead,

$$10_{10} = \begin{array}{cc} \underline{0001} & \underline{0000} \\ 1 & 0 \end{array}$$

or 00010000 in BCD

Similarly,

$$15_{10} = \begin{array}{cc} \underline{0001} & \underline{0101} \\ 1 & 5 \end{array}$$

or 00010101 in BCD

In the above discussion, we have used a group of 4 bits to represent a digit (character) in BCD. 4-bit BCD coding system can be used to represent only decimal numbers because 4 bits are insufficient to represent the various characters used by a computer. Instead of using 4 bits with only 16 possible characters, computer designers commonly use 6 bits to represent characters in BCD code. In the 6-bit BCD code, the four BCD numeric place positions are retained, but two additional *zone* positions are added. With 6 bits, it is possible to represent 64 (2^6)

different characters. This is a sufficient number to code the decimal digits (10), alphabetic letters (26), and other special characters (28). Table 4.2 illustrates the coding of alphabetic and numeric characters in BCD.

In Chapter 3, we have seen the use of octal and hexadecimal number systems as shortcut notation for binary. Because BCD is a 6-bit code, it can be easily divided into two 3-bit groups. Each of these 3-bit groups can be represented by 1 octal digit. Thus, octal number system is used as shortcut notation for memory dump by computers that use BCD code for internal representation of characters. This results in a one-to-three reduction in the volume of memory dump. Table 4.2 also shows the octal equivalent of the alphabetic and numeric characters coded in BCD.

Example 4.1. Show the binary digits used to record the word BASE in BCD.

Solution :

$$\begin{aligned} B &= 110010 \text{ in BCD binary notation} \\ A &= 110001 \text{ in BCD binary notation} \\ S &= 010010 \text{ in BCD binary notation} \\ E &= 110101 \text{ in BCD binary notation} \end{aligned}$$

So the binary digits

$$\begin{array}{cccc} \underline{110010} & \underline{110001} & \underline{010010} & \underline{110101} \\ B & A & S & E \end{array}$$

will record the word BASE in BCD.

Example 4.2. Using octal notation show the BCD coding for the word DIGIT.

Solution :

$$\begin{aligned} D &= 64 \text{ in BCD octal notation} \\ I &= 71 \text{ in BCD octal notation} \\ G &= 67 \text{ in BCD octal notation} \\ I &= 71 \text{ in BCD octal notation} \\ T &= 23 \text{ in BCD octal notation} \end{aligned}$$

So the BCD coding for the word DIGIT in octal notation will be :

$$\begin{array}{cccc} \underline{64} & \underline{71} & \underline{67} & \underline{71} & \underline{23} \\ D & I & G & I & T \end{array}$$

Table 4.2. Alphabetic And Numeric Characters In BCD Along With Their Octal Equivalent.

Character	BCD Code		Octal Equivalent
	Zone	Digit	
A	11	0001	61
B	11	0010	62
C	11	0011	63
D	11	0100	64
E	11	0101	65
F	11	0110	66
G	11	0111	67
H	11	1000	70
I	11	1001	71
J	10	0001	41
K	10	0010	42
L	10	0011	43
M	10	0100	44
N	10	0101	45
O	10	0110	46
P	10	0111	47
Q	10	1000	50
R	10	1001	51
S	01	0010	22
T	01	0011	23
U	01	0100	24
V	01	0101	25
W	01	0110	26
X	01	0111	27
Y	01	1000	30
Z	01	1001	31
1	00	0001	01
2	00	0010	02
3	00	0011	03
4	00	0100	04
5	00	0101	05
6	00	0110	06
7	00	0111	07
8	00	1000	10
9	00	1001	11
0	00	1010	12

EBCDIC

The major problem with BCD code is that only 64 (2^6) different characters can be represented in it. This is not sufficient for providing decimal numbers (10), lower-case letters (26), capital letters (26), and a fairly large number of other special characters (28+).

Hence, the BCD code was extended from a 6-bit code to an 8-bit code. The added 2 bits are used as additional zone bits, expanding the zone to 4 bits. The resulting code is called the *extended binary-coded decimal interchange code (EBCDIC)*. In this code, it is possible to represent 256 (2^8) different characters instead of 64 (2^6). In addition to the various character requirements mentioned above, this also allows a large variety of printable characters and several nonprintable control characters. The *control characters* are used to control such activities as printer vertical spacing, movement of cursor on the terminal screen, etc. All of the 256 bit combinations have not yet been assigned characters, so the code can still grow as new requirements develop.

Because EBCDIC is an 8-bit code, it can be easily divided into two 4-bit groups. Each of these 4-bit groups can be represented by 1 hexadecimal digit (refer to Chapter 3). Thus, hexadecimal number system is used as shortcut notation for memory dump by computers that use EBCDIC for internal representation of characters. This results in a one-to-four reduction in the volume of memory dump. Table 4.3 shows the alphabetic and numeric characters in EBCDIC along with their hexadecimal equivalent.

Developed by IBM, EBCDIC code is used in most IBM models and in many other computers.

ZONED AND PACKED DECIMAL NUMBERS

From Table 4.3, it can be observed that in the EBCDIC code, the digit values are the same as the numeric characters - 0 through 9 (0000-1001). But numeric values need some special consideration because we must have a way of indicating whether the number is positive, negative, or unsigned (implies positive). Thus, when a numeric value is represented in EBCDIC, a sign indicator is used in the zone position of the rightmost digit. A sign indicator of hexadecimal C is a plus sign, hexadecimal D is a minus sign, and a hexadecimal F means the number is unsigned. Table 4.4 illustrates the representation of numeric values in EBCDIC. Note that the only zone affected by the sign is the zone of the rightmost digit. All other zones remain as F, the zone value for numeric characters in EBCDIC. Because each decimal digit has a zone with it, numbers coded in EBCDIC are called *zoned decimal numbers*. Numeric data

input into the computer are usually zoned decimal numbers. Printers can print only those numeric characters that are in a zoned-decimal format.

However, most computers cannot perform arithmetic operations on zoned-decimal data. Thus, before any arithmetic operation can be performed, the data must be converted to a format on which arithmetic operations are possible. One such acceptable format is the *packed decimal* format. The following steps are used to convert a zoned decimal number to a packed decimal number :

Step 1 : The zone half and the digit half of the rightmost byte are reversed. This moves the sign to the extreme right of the number.

Step 2 : All remaining zones are dropped out.

Table 4.5 illustrates the conversion process of zoned decimal data to packed data.

It may be observed that packed data requires fewer number of bytes (group of 8 bits) as compared to zoned data. In the zoned format, there is only one digit per byte (each digit along with the zone requires 8 bits). But there are two digits in each byte in the packed format (each digit requires 4 bits). If the packing process does not completely fill a byte, it is filled with a zero. For example, the zoned data F3F4F5F6 will convert to packed data 03456F. Observe that in this example, the zoned data requires 4 bytes and the packed data requires only 3 bytes.

Example 4.3. Using binary notation, write the EBCDIC coding for the word BIT. How many bytes are required for this representation?

Solution :

B = 1100 0010 in EBCDIC binary notation

I = 1100 1001 in EBCDIC binary notation

T = 1110 0011 in EBCDIC binary notation

So the EBCDIC coding for the word BIT in binary notation will be :

11000010 11001001 11100011
B I T

3 bytes will be required for this representation because each letter requires 1 byte (or 8 bits) for its representation.

Table 4.3. Alphabetic And Numeric Characters In EBCDIC Along With Their Hexadecimal Equivalent.

Character	EBCDIC Code		Hexadecimal Equivalent
	Zone	Digit	
A	1100	0001	C1
B	1100	0010	C2
C	1100	0011	C3
D	1100	0100	C4
E	1100	0101	C5
F	1100	0110	C6
G	1100	0111	C7
H	1100	1000	C8
I	1100	1001	C9
J	1101	0001	D1
K	1101	0010	D2
L	1101	0011	D3
M	1101	0100	D4
N	1101	0101	D5
O	1101	0110	D6
P	1101	0111	D7
Q	1101	1000	D8
R	1101	1001	D9
S	1110	0010	E2
T	1110	0011	E3
U	1110	0100	E4
V	1110	0101	E5
W	1110	0110	E6
X	1110	0111	E7
Y	1110	1000	E8
Z	1110	1001	E9
0	1111	0000	F0
1	1111	0001	F1
2	1111	0010	F2
3	1111	0011	F3
4	1111	0100	F4
5	1111	0101	F5
6	1111	0110	F6
7	1111	0111	F7
8	1111	1000	F8
9	1111	1001	F9

Table 4.4. Numeric Values In EBCDIC In Hexadecimal Notation

Numeric Value	EBCDIC	Sign Indicator
345	F3F4F5	F for unsigned
+345	F3F4C5	C for positive
-345	F3F4D5	D for negative

Table 4.5. Zoned And Packed Decimal Numbers

Numeric Value	Zoned Format	Packed Format
345	F3F4F5	345F
+345	F3F4C5	345C
-345	F3F4D5	345D
3456	F3F4F5F6	03456F

Example 4.4. Write the EBCDIC coding for the word ZONE (use hexadecimal notation). How many bytes will be required for this representation?

Solution :

- Z = E9 in EBCDIC hexadecimal notation
- O = D6 in EBCDIC hexadecimal notation
- N = D5 in EBCDIC hexadecimal notation
- E = C5 in EBCDIC hexadecimal notation

So the EBCDIC coding for the word ZONE in hexadecimal notation will be :

E9 D6 D5 C5
Z O N E

Each hexadecimal digit requires 4 bits and there are altogether 8 hexadecimal digits. So in all $8 \times 4 = 32$ bits will be required. But $8 \text{ bits} = 1 \text{ byte}$. So $32 \text{ bits} = 4 \text{ bytes}$. Hence, 4 bytes will be required for this representation.

We may also write directly that since each letter requires 1 byte for its representation in EBCDIC and there are 4 letters in the word ZONE, so 4 bytes will be required

for this representation.

Example 4.5. Write the EBCDIC zoned-decimal coding for the value +256 (use hexadecimal). How many bytes will be required for this representation?

Solution :

+256 = F2F5C6 in EBCDIC

Each hexadecimal digit requires 4 bits and there are altogether 6 hexadecimal digits. So in all $6 \times 4 = 24$ bits or 3 bytes ($8 \text{ bits} = 1 \text{ byte}$) will be required for this representation.

We may also write directly that since each digit requires 1 byte for its representation in the EBCDIC zoned decimal coding and there are 3 digits in the given number, so 3 bytes will be required for this representation.

Example 4.6. Write -128 as packed decimal number (use hexadecimal). How many bytes will be required for this representation?

Solution :

-128 = F1F2D8 in EBCDIC
= 128D in packed format.

Each hexadecimal digit requires 4 bits and there are altogether 4 hexadecimal digits. So in all $4 \times 4 = 16$ bits or 2 bytes ($1 \text{ byte} = 8 \text{ bits}$) will be required for this representation.

ASCII

Another computer code that is very widely used is the American Standard Code for Information Interchange (ASCII). ASCII has been adopted by several American computer manufacturers as their computers' internal code. This code is popular in data communications, is used almost exclusively to represent data internally in microcomputers, and is frequently found in the larger computers produced by some vendors.

ASCII is of two types : ASCII-7 and ASCII-8. ASCII-7 is a 7 bit code that allows 128 (2^7) different characters. The first 3 bits are used as zone bits and the last 4 bits indicate the digit. Microcomputers using 8-bit byte (group of 8 bits for one byte) use the 7-bit ASCII by leaving the leftmost first bit of each byte as a zero. Table

4.6 shows the alphabetic and numeric characters in ASCII-7 notation.

ASCII-8 is an extended version of ASCII-7. It is an 8-bit code that allows 256 (2^8) different characters rather than 128. The additional bit is added to the zone bits. Table 4.7 shows the alphabetic and numeric characters in ASCII-8 notation. Observe that other than the zone-value differences, ASCII-7 and ASCII-8 are identical. ASCII also uses hexadecimal as its four-to-one shortcut notation for memory dump. Tables 4.6 and 4.7 also show the hexadecimal equivalent of the ASCII notations.

Example 4.7. Write the binary coding for the word BOY in ASCII-7. How many bytes are required for this representation?

Solution :

B = 1000010 in ASCII-7 binary notation

O = 1001111 in ASCII-7 binary notation

Y = 1011001 in ASCII-7 binary notation.

Hence the binary coding for the word BOY in ASCII-7 will be :

<u>1000010</u>	<u>1001111</u>	<u>1011001</u>
B	O	Y

As each character in ASCII-7 requires one byte for its representation and since there are 3 characters in the word BOY, so 3 bytes will be required for this representation.

Example 4.8. Write the hexadecimal coding for the word GIRL in ASCII-7. How many bytes are required for this representation?

Solution :

G = 47 in ASCII-7 hexadecimal notation

I = 49 in ASCII-7 hexadecimal notation

R = 52 in ASCII-7 hexadecimal notation

L = 4C in ASCII-7 hexadecimal notation

Hence the hexadecimal coding for the word GIRL in ASCII-7 will be :

<u>47</u>	<u>49</u>	<u>52</u>	<u>4C</u>
G	I	R	L

As each character in ASCII-7 requires one byte for its representation and since there are 4 characters in the word

Table 4.6. Numeric And Alphabetic Characters In ASCII-7 Notation Along With Their Hexadecimal Equivalent.

Character	ASCII-7 Code		Hexadecimal Equivalent
	Zone	Digit	
0	011	0000	30
1	011	0001	31
2	011	0010	32
3	011	0011	33
4	011	0100	34
5	011	0101	35
6	011	0110	36
7	011	0111	37
8	011	1000	38
9	011	1001	39
A	100	0001	41
B	100	0010	42
C	100	0011	43
D	100	0100	44
E	100	0101	45
F	100	0110	46
G	100	0111	47
H	100	1000	48
I	100	1001	49
J	100	1010	4A
K	100	1011	4B
L	100	1100	4C
M	100	1101	4D
N	100	1110	4E
O	100	1111	4F
P	101	0000	50
Q	101	0001	51
R	101	0010	52
S	101	0011	53
T	101	0100	54
U	101	0101	55
V	101	0110	56
W	101	0111	57
X	101	1000	58
Y	101	1001	59
Z	101	1010	5A

GIRL, so 4 bytes will be required for this

representation.

Example 4.9. Write the binary coding for the word SKY in ASCII-8. How many bytes are required for this representation?

Solution :

S = 10110011 in ASCII-8 binary notation
 K = 10101011 in ASCII-8 binary notation
 Y = 10111001 in ASCII-8 binary notation

Hence the binary coding for the word SKY in ASCII-8 will be :

10110011 10101011 10111001
 S K Y

As each character in ASCII-8 requires one byte for its representation and since there are 3 characters in the word SKY, so 3 bytes will be required for this representation.

Example 4.10. Write the hexadecimal coding for the word STAR in ASCII-8. How many bytes are required for this representation?

Solution :

S = B3 in ASCII-8 hexadecimal notation
 T = B4 in ASCII-8 hexadecimal notation
 A = A1 in ASCII-8 hexadecimal notation
 R = B2 in ASCII-8 hexadecimal notation

Hence the hexadecimal coding for the word STAR in ASCII-8 will be :

B3 B4 A1 B2
 S T A R

As each character in ASCII-8 requires one byte for its representation and since there are 4 characters in the word STAR, so 4 bytes will be required for this representation.

COLLATING SEQUENCE

The value of an alphanumeric or alphabetic data element is usually the name of some object. Obviously one would not like to perform any arithmetic on such data but one may like to compare them in order to arrange them in some desired sequence. Now, if we compare the alphabetic

values A and B, which one will be treated as greater by the computer? For an answer to such questions, it is necessary to have some assigned ordering among the characters used by the computer. This ordering is known as the *collating sequence*.

Collating sequence may vary from one computer system to another depending on the type of computer code used by a particular computer. To illustrate this, let us consider the computer codes already discussed in this chapter. Observe from Tables 4.2 and 4.3 that the zone values of the characters A through 9 decreases in BCD code from the equivalent of decimal 3 down to 0, while in EBCDIC, the zone values of the characters A through 9 increases from the equivalent of decimal 12 to 15. This means that a computer which uses BCD code for its internal representation of characters will treat alphabetic characters (A, B, ..., Z) to be greater than numeric characters (0, 1, ..., 9). On the other hand, a computer which uses EBCDIC for its internal representation of characters will treat numeric characters to be greater than alphabetic characters. Similarly, observe from Tables 4.6 and 4.7 that a computer which uses ASCII for its internal representation of characters will place numbers ahead of letters during a sort (ascending) because the number characters have a zone value that is less than the zone value for letters.

However, whatever may be the type of computer code used, in most (not all - in BCD 0>9) collating sequences the following rules are observed :

1. Letters are considered in alphabetic order (A<B<C<...<Z)
2. Digits are considered in numeric order (0<1<2<...<9).

Example 4.11. Suppose a computer uses EBCDIC as its internal representation of characters. In which order will this computer sort the following strings : 23, A1, 1A ?

Solution :

In EBCDIC, numeric characters are treated to be greater than alphabetic characters. Hence the numeric characters will be placed after the alphabetic characters. So the computer will treat the given string as :

A1<1A<23

Hence the sorted sequence will be : A1, 1A, 23.

Table 4.7. Numeric And Alphabetic Characters In ASCII-8 Notation Along With Their Hexadecimal Equivalent.

Character	ASCH-8 Code		Hexa- decimal Equivalent
	Zone	Digit	
0	0101	0000	50
1	0101	0001	51
2	0101	0010	52
3	0101	0011	53
4	0101	0100	54
5	0101	0101	55
6	0101	0110	56
7	0101	0111	57
8	0101	1000	58
9	0101	1001	59
A	1010	0001	A1
B	1010	0010	A2
C	1010	0011	A3
D	1010	0100	A4
E	1010	0101	A5
F	1010	0110	A6
G	1010	0111	A7
H	1010	1000	A8
I	1010	1001	A9
J	1010	1010	AA
K	1010	1011	AB
L	1010	1100	AC
M	1010	1101	AD
N	1010	1110	AE
O	1010	1111	AF
P	1011	0000	B0
Q	1011	0001	B1
R	1011	0010	B2
S	1011	0011	B3
T	1011	0100	B4
U	1011	0101	B5
V	1011	0110	B6
W	1011	0111	B7
X	1011	1000	B8
Y	1011	1001	B9
Z	1011	1010	BA

Example 4.12. Suppose a computer uses ASCII for its internal representation of characters. In which order will this computer sort the following strings: 23, A1, 1A ?

Solution :

In ASCII, numeric characters are treated to be less than alphabetic characters. Hence the numeric characters will be placed before the alphabetic characters. So this computer will treat the given string as : 23 < 1A < A1.

Hence the sorted sequence will be : 23, 1A, A1.

QUESTIONS

1. Define the term 'byte'. What is the difference between a bit and a byte ?
2. Write the 4-bit BCD code for the following numbers : (a) 25_{10} , (b) 64_{10} , (c) 128_{10} , (d) 1024_{10}
3. Using binary notation, show the BCD coding for the following words : (a) BIT (b) BYTE (c) CODE.
4. Using octal notation, show the BCD coding for the following words : (a) COMPUTER (b) INPUT (c) VIDEO.
5. Why was BCD code extended to EBCDIC ?
6. How many different characters are possible in the following codes : PCD, EBCDIC, ASCII-7, and ASCII-8 ?
7. Suppose a new computer code is designed that uses 9 bits. How many different characters are possible in this code ?
8. Why are octal and hexadecimal shortcut notations used ? Identify the shortcut notations used for each of these computer codes : BCD, EBCDIC, ASCII-7, and ASCII-8.
9. Why do we have a packed decimal format ? How does it differ from a zoned decimal format ?

10. Using binary notation, write the EBCDIC coding for the following words : (a) SUN (b) MOON (c) AT. How many bytes are required for each of these representations ?
11. Using hexadecimal notation, write the EBCDIC coding for the following words : (a) PROGRAM (b) OUTPUT (c) BYTE. How many bytes are required for each of these representations ?
12. Using hexadecimal notation, write the zoned-decimal coding for the following numbers : (a) 1256 (b) +439 (c) -63. How many bytes will be required for each of these representations ?
13. Using hexadecimal notation, write the packed-decimal coding for the following numbers : (a) 12915 (b) +9876 (c) 872.
How many bytes will be required for each of these representations ?
14. List out the similarities and differences between 7-bit and 8-bit ASCII.
15. Using binary notation, write the ASCII-7 and ASCII-8 codes for the following words : (a) DRY (b) WET (c) DAMP.
- How many bytes are required for each of these representations ?
16. Using hexadecimal notation, write the ASCII-7 and ASCII-8 codes for the following words : (a) PRINT (b) TYPE (c) RUB.
How many bytes are required for each of these representations ?
17. Explain the meaning of the term "collating sequence".
18. A computer uses EBCDIC as its internal representation of characters. In which order will this computer sort the following strings : ABC, 123, 245, ADD ?
19. A computer uses ASCII. In which order will this computer sort the following strings : BED, 512, ADD, 128, BAD ?
20. Give the full form of the following abbreviations : BCD, EBCDIC, ASCII.

5. COMPUTER ARITHMETIC

In Chapter 4 you have seen that computers store numbers, letters, and other characters in coded form that is related to the binary number system. In this chapter you will learn why computers use binary numbers instead of decimal numbers and how the basic arithmetic operations are performed inside the computer using binary numbers.

WHY BINARY

You might have observed in Chapter 3 that the use of a smaller base may require more positions to represent a given value (recall the reason for using octal and hexadecimal shortcut notations). For example, $9_{10} = 1001_2$. Here four positions are required instead of one to represent the decimal number 9 in binary form. In spite of this fact, almost all computers use binary numbers. So the obvious question that arises to one's mind is that why do we go for binary numbers instead of decimal numbers? The reasons are as follows :

1. The first and the foremost reason is that electronic and electrical components, by their very nature, operate in a binary mode. Information is handled in the computer by

electronic/electrical components such as transistors, semiconductors, wires, etc. all of which can only indicate two states or conditions - on(1) or off(0). Transistors are either conducting(1) or nonconducting(0) ; magnetic materials are either magnetized(1) or non-magnetized(0) in one direction or in the opposite direction; a pulse or voltage is present(1) or not present(0) in wire. All information is represented within the computer by the presence or absence of these various signals. The binary number system, which has only two digits (0 and 1), is most suitable and is conveniently used to express the two possible states. The concept of binary components is illustrated in Figure 5.1.

2. The second reason is that computer circuits only have to handle two binary digits rather than ten decimal digits. The result is that the internal circuit design of computers is simplified to a great extent. This ultimately results in less expensive and more reliable circuits for computers.

3. Finally, the binary system is used because everything that can be done with a base of 10 can also be done in binary. How this is achieved has been discussed below.







BINARY STATE	ON (1)	OFF (0)
Bulb		
Switch		
Circuit Pulse		

Figure 5.1. Examples of devices that work in binary mode. These devices can only represent two states - on or off. These states can represent either a 1 or a 0 or a yes or a no.

BINARY ARITHMETIC

In this section you will see how the four basic arithmetic operations are performed inside a computer using binary numbers. Actually, binary arithmetic is much simpler to learn because binary system deals with only two digits - 0 and 1. So all binary numbers are made up of only 0's and 1's and when arithmetic operations are performed on these numbers, the results are also in 0's and 1's only.

ADDITION

Binary addition is performed in the same manner as decimal addition. However, since binary system has only two digits, the addition table for binary arithmetic is very simple consisting of only four entries. The complete table for binary addition is as follows:

$$\begin{aligned}
 0 + 0 &= 0 \\
 0 + 1 &= 1 \\
 1 + 0 &= 1 \\
 1 + 1 &= 0 \text{ plus a carry of 1 to next higher column}
 \end{aligned}$$

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, 10 plus 10 binary requires the addition of two 1's in the second position. Since $1+1 = 0$ plus a carry-over of 1, the sum of $10 + 10$ is 100 in binary.

By repeated use of the above rules, any two binary numbers can be added together by adding two bits at a time. The exact procedure is illustrated with the examples given below.

Example 5.1. Add the binary numbers 101 and 10 in both decimal and binary form.

Solution :

BINARY	DECIMAL
101	5
+10	+2
----	----
111	7

Example 5.2. Add the binary numbers 10011 and 1001 in both decimal and binary form.

Solution :

BINARY	DECIMAL
carry 11	carry 1
10011	19
+1001	+9
-----	-----
11100	28

Note that while adding the first and the second column of the binary example, a carry is generated.

Example 5.3. Add the binary numbers 100111 and 11011 in both decimal and binary form.

Solution :

BINARY	DECIMAL
CARRY 11111	CARRY 1
100111	39
+ 11011	+ 27
-----	----
1000010	66

In this example, we face a new situation (1+1+1) brought about by the carry-over of 1 in the second column. This can also be handled using the same four rules for binary addition. The addition of three 1's can be broken up into two steps. First we add only two 1's giving 10 (1+1=10). The third 1 is now added to this result to obtain 11 (a 1 sum with a 1 carry). So we conclude that 1 + 1 + 1 = 1 plus a carry of 1 to next higher column.

SUBTRACTION

The principles of decimal subtraction can as well be applied to subtraction of numbers in other bases. It consists of two steps, which are repeated for each column of the numbers. The first step is to determine if it is necessary to borrow. If the subtrahend (the lower digit) is larger than the minuend (the upper digit), it is necessary to borrow from the column to the left. It is important to note here that the value borrowed depends upon the base of the number and is always the decimal equivalent of the base. Thus, in decimal, 10 is borrowed; in binary, 2 is borrowed; in octal, 8 is borrowed; in hexadecimal, 16 is borrowed. The second step is simply to subtract the lower value from the upper value. The complete table for binary subtraction is as follows :

0 - 0 = 0
1 - 0 = 1
1 - 1 = 0
0 - 1 = 1 with a borrow from the next column

Observe that the only case in which it is necessary to borrow is when 1 is subtracted from 0. The exact procedure is illustrated with the examples given below.

Example 5.4 Subtract 01110₂ from 10101₂.

Solution :

Borrow	{	12
		0202
		10101
		-01110

		00111

In the first column, 0 is subtracted from 1. No borrow is required in this case and the result is 1. In the second column, we have to subtract 1 from 0. As seen in the table for binary subtraction, a borrow is necessary to perform this subtraction. So a 1 is borrowed from the third column which becomes 2 in the second column because the base is 2. A 1 in the 4s column is equal to 2 in the 2s column. Now, in the second column, we subtract 1 from 2 giving a result of 1. The borrow performed in the second column reduces the 1 in the third column to 0. So in the third column, once again we have to subtract 1 from 0 for which borrow is required. The fourth column contains a 0 and thus has nothing to borrow. Therefore, we have to borrow from the fifth column. Borrowing 1 from the fifth column gives 2 in the fourth column. A 1 in the 16s column equals 2 in the 8s column. Now the fourth column has something to borrow. When 1 of the 2 in the fourth column is borrowed, it becomes 2 in the third column. Now, in the third column, we subtract 1 from 2 giving a result of 1. The borrow performed in the third column reduces the 1 in the fifth column to 0 and the 2 in the fourth column to 1. Hence, subtraction of the fourth column is now 1 from 1, giving 0 and in the fifth column, subtraction is 0 from 0, giving 0. Thus the final result of subtraction is 00111₂. The result may be verified by subtracting 14₁₀ (= 01110₂) from 21₁₀ (10101₂) which gives 7₁₀ (= 00111₂).

Example 5.5. Subtract 0111000₂ from 1011100₂.

Solution :

BORROW	2	
		1011100
		- 0111000

		0100100

The result may be verified by subtracting 56₁₀ (= 0111000₂) from 92₁₀ (= 1011100₂) which gives 36₁₀ (= 0100100₂).

Additive Method of Subtraction.

The direct method of subtraction using the borrow concept seems to be easiest when people perform subtraction with paper and pencil. However, when subtraction is implemented by means of digital components, this method is found to be less efficient than the additive method of subtraction. It may sound surprising that even subtraction is performed using an additive method. This method of subtraction by an additive approach is known as *complementary subtraction*.

In order to understand complementary subtraction, it is necessary to know what is meant by the complement of a

number. For a number which has n digits in it, a *complement* is defined as the difference between the number and the base raised to the n th power minus one. The definition is illustrated with the following examples :

Example 5.6. Find the complement of 37_{10} .

Solution :

Since the number has 2 digits and the value of base is 10, so $(\text{Base})^n - 1 = 10^2 - 1 = 99$

$$\text{Now } 99 - 37 = 62$$

Thus the complement of $37_{10} = 62_{10}$.

Example 5.7. Find the complement of 6_8 .

Solution :

Since the number has 1 digit and the value of base is 8, so $(\text{Base})^n - 1 = 8^1 - 1 = 7_{10}$

$$\text{Also } 6_8 = 6_{10}$$

$$\text{No: } 7_{10} - 6_{10} = 1_{10} = 1_8$$

Thus the complement of $6_8 = 1_8$.

Example 5.8. Find the complement of 10101_2 .

Solution :

Since the number has 5 digits and the value of base is 2,

$$\text{So } (\text{Base})^n - 1 = 2^5 - 1 = 31_{10}$$

$$\text{Also } 10101_2 = 21_{10}$$

$$\text{Now } 31_{10} - 21_{10} = 10_{10} = 1010_2$$

Thus the complement of $10101_2 = 01010_2$.

Observe from Example 5.8 that in case of binary numbers, it is not necessary to go through the usual process of obtaining complement. Instead, when dealing with binary numbers, a quick way to obtain a number's complement is to transform all its 0's to 1's, and all its 1's to 0's. For example, the complement of 1011010 is 0100101 . This circuit for obtaining complement of a number in binary system can be easily designed at very less expense.

So we have seen how to obtain the complement of a

number. We will now see how subtraction is performed using the complementary method.

Subtraction by the complementary method involves the following three steps :

- Step 1 : Find the complement of the number you are subtracting (subtrahend);
- Step 2 : Add this to the number from which you are taking away (minuend);
- Step 3 : If there is a carry of 1, add it to obtain the result; if there is no carry, recomplement the sum and attach a negative sign to obtain the result.

To illustrate the procedure, let us first consider few examples for decimal numbers.

Example 5.9. Subtract 56_{10} from 92_{10} using complementary method.

Solution :

$$\begin{aligned} \text{Step 1 :} \quad & \text{Complement of } 56_{10} \\ & = 10^2 - 1 - 56 \\ & = 99 - 56 \\ & = 43_{10} \end{aligned}$$

$$\begin{array}{r} \text{Step 2 :} \quad 92 \\ + 43 \text{ (complement of } 56) \\ \hline 135 \\ | \end{array}$$

$$\text{Step 3 :} \quad \rightarrow 1 \text{ (add the carry of 1)}$$

$$\text{Result} \quad = 36$$

The result may be verified using the method of normal subtraction : $92 - 56 = 36$.

Example 5.10. Subtract 35_{10} from 18_{10} using complementary method.

Solution :

$$\begin{aligned} \text{Step 1 :} \quad & \text{Complement of } 35_{10} \\ & = 10^2 - 1 - 35 \\ & = 99 - 35 \end{aligned}$$

$$\begin{array}{r}
 = 64_{10} \\
 \text{Step 2:} \\
 18 \\
 +64 \text{ (complement of 35)} \\
 \hline
 82
 \end{array}$$

Step 3: There is no carry. So recomplement the sum and attach a negative sign to obtain the result.
 Result = $-(99 - 82)$
 $= -17$

The result may be verified using the method of normal subtraction:

$$18 - 35 = -17$$

Let us re-work these examples using binary numbers.

Example 5.11. Subtract 0111000_2 (56_{10}) from 1011100_2 (92_{10}) using complementary method.

Solution:

$$\begin{array}{r}
 1011100 \\
 +1000111 \text{ (complement of } 0111000) \\
 \hline
 10100011 \\
 | \\
 \text{-----} \rightarrow 1 \text{ (add the carry of 1)} \\
 \hline
 0100100
 \end{array}$$

$$\text{Result} = 0100100_2 = 36_{10}$$

Verify the result with the results of Example 5.5 and Example 5.9.

Example 5.12. Subtract 100011_2 (35_{10}) from 010010_2 (18_{10}) using complementary method.

Solution:

$$\begin{array}{r}
 010010 \\
 +011100 \text{ (complement of } 100011) \\
 \hline
 101110
 \end{array}$$

As there is no carry, so we have to complement the sum and attach a negative sign to it. Hence

$$\text{Result} = -010001_2 \text{ (complement of } 101110_2)$$

$$= -17_{10}$$

Verify the result with the result of Example 5.10.

Example 5.13. Subtract 01110_2 from 10101_2 using complementary method.

Solution:

$$\begin{array}{r}
 10101 \\
 +10001 \text{ (complement of } 01110) \\
 \hline
 100110 \\
 | \\
 \text{-----} \rightarrow 1 \text{ (add the carry of 1)} \\
 \hline
 00111
 \end{array}$$

$$\text{Result} = 00111_2$$

Verify the result with the result of Example 5.4.

MULTIPLICATION

Multiplication in the binary system also follows the same general rules as decimal multiplication. However, learning the binary multiplication is a trivial task because the table for binary multiplication is very short, with only four entries instead of the 100 necessary for decimal multiplication. The complete table for binary multiplication is as follows:

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

The method of binary multiplication is illustrated with the example given below. 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.

Example 5.14. Multiply the binary numbers 1010 and 1001.

1010	Multiplicand
x1001	Multiplier

1010	Partial Product
0000	Partial Product
0000	Partial Product
1010	Partial Product

1011010	Final Product

1011010 Final Product

Note that the multiplicand is simply copied when multiplier digit is 1 and when the multiplier digit is 0, the partial product is only a string of zero's. As in decimal multiplication, each partial product is shifted one place to the left from the previous partial product. Finally, all the partial products obtained in this manner are added according to the binary addition rules to obtain the final product.

In actual practice, whenever a 0 appears in the multiplier, a separate partial product consisting of a string of zeros need not be generated. Instead, only a left shift will do. As a result, Example 5.14 may be reduced to

```

1010
x1001
-----
1010
1010ss (s = left shift)
-----
1011010
    
```

A computer would also follow this procedure in performing multiplication. The result of this multiplication may be verified by multiplying 10_{10} (1010_2) by 9_{10} (1001_2) which produces a result of 90_{10} (1011010_2).

It may not be obvious how to handle the addition if the result of the multiplication gives columns with more than two 1s. They can be handled as pairs or by adjusting the column to which the carry is placed, as shown by Example 5.15.

Additive Method of Multiplication.

Most of the computers perform multiplication operation by the way of addition only. This can be easily seen by considering an example, say 4×8 . The result for this multiplication can be determined by evaluating, with necessary carry overs, $8+8+8+8$. That is, the result is obtained simply by adding the digit 8 four times. Similarly, the computer performs all multiplication operations in binary using the additive approach.

This idea of repeated addition may seem to be a longer way of doing things, but remember that the computer is well suited to carry out the operations at great speed. The internal circuit design of computer systems is

also simplified to a great extent by using this method of multiplication.

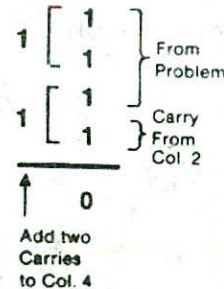
Example 5.15. Multiply the binary numbers 1111 and 111.

Solution :

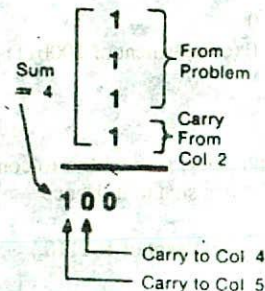
```

  1111
x 111
-----
  1111
 1111
1111
-----
1101001
    
```

Addition
Handled
As Pairs
(Column 3)



Addition
Handled
As Single
Carry (Col. 3)



DIVISION

Binary division is, again, very simple. As in the decimal system (or in any other number system), division by zero is meaningless. Hence, the complete table for binary division is as follows :

$$0 / 1 = 0$$

$$1 / 1 = 1$$

The division process is performed in a manner similar to decimal division. The rules for binary division are :

1. Start from the left of the dividend.
2. Perform a series of subtractions in which the divisor is subtracted from the dividend.
3. If subtraction is possible, put a 1 in the quotient and subtract the divisor from the corresponding digits of dividend.
4. If subtraction is not possible (divisor greater than remainder), record a 0 in the quotient.
5. Bring down the next digit to add to the remainder digits. Proceed as before in a manner similar to long division.

The method is illustrated by Example 5.16.

The result may be verified by dividing 33_{10} (100001_2) by 6_{10} (110_2) which gives a quotient of 5_{10} (101_2) and a remainder of 3_{10} (11_2).

Additive Method of Division.

Even division operation is performed inside most computers by the process of addition only. This may again sound surprising, but it is true. The computer performs the division operation essentially by repeating the complementary subtraction method. For example, $35 / 5$ may be thought of as :

$$35 - 5 = 30$$

$$30 - 5 = 25$$

$$25 - 5 = 20$$

$$20 - 5 = 15$$

$$15 - 5 = 10$$

$$10 - 5 = 5$$

$$5 - 5 = 0$$

Example 5.16. Divide 100001_2 by 110_2 .

Solution :

$$\begin{array}{r}
 0101 \quad (\text{Quotient}) \\
 110 \overline{) 100001} \quad (\text{Dividend}) \\
 \underline{110} \leftarrow 1 \\
 1000 \leftarrow 2 \\
 \underline{110} \leftarrow 3 \\
 100 \leftarrow 4 \\
 \underline{110} \leftarrow 5 \\
 1001 \leftarrow 6 \\
 \underline{110} \leftarrow 7 \\
 \hline
 11 \quad (\text{Remainder})
 \end{array}$$

- 1 Divisor greater than 100, so put 0 in quotient
- 2 Add digit from dividend to group used above
- 3 Subtraction possible so put 1 in quotient
- 4 Remainder from subtraction plus digit from dividend
- 5 Divisor greater, so put 0 in quotient
- 6 Add digit from dividend to group
- 7 Subtraction possible, so put 1 in quotient

That is, the divisor is subtracted repeatedly from the dividend until the result of subtraction becomes less than or equal to zero. The total number of times subtraction was performed gives the value of the quotient. In this case, the value of quotient is 7 because the divisor (5) has been subtracted 7 times from the dividend (35) until the result of subtraction becomes zero. If the result of last subtraction is zero then there is no remainder for the division. But, if it is less than zero, then the last subtraction is ignored and the result of the previous subtraction is taken as the value of the remainder. In this case, the last subtraction operation is not counted for evaluating the value of the quotient. The process is illustrated below with an example.

Example 5.17. Divide 33_{10} by 6_{10} using the method of addition.

Solution :

$$33 - 6 = 27$$

$$27 - 6 = 21$$

$$21 - 6 = 15$$

$$15 - 6 = 9$$

$$9 - 6 = 3$$

$$3 - 6 = -3$$

Total number of subtractions = 6. Since the result of the last subtraction is less than zero, so

Quotient = 6 - 1 (ignore last subtraction) = 5

Remainder = 3 (result of previous subtraction)

Thus, $33 / 6 = 5$ with a remainder 3.

Note that it has been assumed here that all the subtraction operations are carried out using the complementary subtraction method (additive method).

Once again, performing division inside a computer by the way of addition is desirable because the addition and complementation operations are easily performed in a computer and usually save the labour and expense of designing complicated circuits.

We have demonstrated how computer arithmetic is based on addition. Exactly how this simplifies matter can only be understood in the context of binary (not in decimal). The number of individual steps may indeed be

increased because all computer arithmetic is reduced to addition, but the computer can carry out binary additions at such great speed that this is not a disadvantage.

QUESTIONS

- Why have computers been designed to use the binary number system ?
- Add the binary numbers 1011 and 101 in both decimal and binary form.
- Add the binary numbers 1010110 and 1011010.
- Add the binary numbers 10111 and 1011.
- Find the complement of the following numbers :
 - 495_{10}
 - 29_{10}
 - 4_8
 - C_{16}
 - 2_2
 - 32_2
- Find the complement of the following binary numbers :
 - 10
 - 101
 - 101101
 - 011011
 - 001101001110
- Subtract 0110111_2 from 1101110_2 .
- Subtract 01010_2 from 10000_2 .
- Subtract 011011_2 from 110111_2 .
- Subtract 25_{10} from 50_{10} using complementary method.
- Subtract 25_{10} from 20_{10} using complementary method.
- Subtract 234_{10} from 588_{10} using complementary method.
- Subtract 216_{10} from 172_{10} using complementary method.
- Subtract 01010_2 from 10000_2 using complementary method.

15. Subtract 110111_2 from 101110_2 using complementary method.
16. Subtract 011011_2 from 110111_2 using complementary method.
17. Subtract 1111_2 from 1100_2 using complementary method.
18. Multiply the binary numbers 1100 and 1010 .
19. Multiply the binary numbers 01101 and 1001 .
20. Multiply the binary numbers 101111 and 111 .
21. Divide 11001_2 by 101_2 .
22. Divide 0110111_2 by 0111_2 .
23. Briefly explain how multiplication and division operations are performed within a computer using additive approach.
24. What is the primary advantage of performing subtraction by the complementary method in digital computers ?
25. Discuss the advantages and disadvantages of performing the various arithmetic operations by the additive method in a digital computer.

6. BOOLEAN ALGEBRA AND LOGIC CIRCUITS

In the previous chapters you have seen that computers normally use binary numbers. In this chapter, you will learn about an algebra that deals with the binary number system. This algebra, known as Boolean algebra, is very useful in designing logic circuits used by the processors of computer systems. In addition to this, you will also learn about the elementary logic gates that are used to build up circuits of different types to perform the necessary arithmetic operations. These logic gates are the building blocks of all the circuits in a computer. Finally, in this chapter, you will also learn how to use Boolean algebra to design simple logic circuits frequently used by the arithmetic logic unit of almost all computers.

BOOLEAN ALGEBRA

In the mid-1800's, an algebra which simplified the representation and manipulation of propositional logic was

developed by the English mathematician, George Boole (1815-1864). It became known as Boolean algebra after its developer's name. Later, in the year 1938, Claude E. Shannon, a research assistant in the department of electrical engineering at the Massachusetts Institute of Technology, published a thesis entitled, "A Symbolic Analysis of Relay and Switching Circuits". In his thesis, he proposed the use of Boolean algebra in the design of relay switching circuits. The basic techniques described by Shannon were adopted almost universally for the design and analysis of switching circuits. Because of the analogous relationship between the action of relays and of modern electronic circuits, the same techniques which were developed for the design of relay circuits are still being used in the design of modern high-speed computers.

Boolean algebra provides an economical and straightforward approach to the design of relay and other

types of switching circuits. Just as an ordinary algebraic expression may be simplified by means of the basic theorems, the expression describing a given switching circuit network may also be reduced or simplified using Boolean algebra. Boolean algebra is now being used extensively in designing the circuitry used in computers. In short, a knowledge of Boolean algebra is must in the computing field.

FUNDAMENTAL CONCEPTS OF BOOLEAN ALGEBRA

1. *Use of Binary digits.* In a normal algebraic expression, a variable can take any numerical value. For example, in the expression $3A + 7B = C$, we assume that A, B, and C may range through the entire field of real numbers.

Since Boolean algebra deals with the binary number system, the variables used in the Boolean equations may assume only two possible values (0 and 1). If an equation describing logical circuitry has several variables, it is still understood that each of the variables can assume only the values 0 or 1. For example, in the equation $A + B = C$, each of the variables A, B, and C may have only the values 0 or 1.

2. *Logical addition.* The symbol '+' is used for logical addition operator. It is also known as 'OR' operator. We can define the + symbol (OR operator) by listing all possible combinations of A and B and the resulting value of C in the equation $A + B = C$. It may be noted that since the variables A and B can have only two possible values (0 or 1) so only four (2^2) combinations of inputs are possible as shown in Table 6.1. The resulting output values for each of the four input combinations are given in the table. Such a table is known as a *truth table*. Thus, Table 6.1 is truth table for the logical OR operator.

Observe that the result is 0 only when the value of both the input variables is 0. The result is 1 when any of the input variables is 1. Note that a result of 1 is also obtained when both the inputs A and B are 1. This is the reason why the + symbol does not have the "normal" meaning, but is a logical addition operator. This concept of logical addition may be extended to any number of variables. For example, in the equation $A + B + C + D = E$, even if A, B, C, and D all had the

value of 1, the sum of the values (the result E) would be 1 only. The equation $A + B = C$ is normally read as "A or B equals C".

Table 6.1. Truth Table For Logical OR (+) Operator.

INPUTS			OUTPUT
A	+	B	= C
0		0	0
0		1	1
1		0	1
1		1	1

3. *Logical multiplication.* The symbol '.' is used for logical multiplication operator. It is also known as 'AND' operator. We can again define the . symbol (AND operator) by listing all possible combinations of A and B and the resulting value of C in the equation $A . B = C$. The truth table for logical AND operator is shown in Table 6.2. Observe from the truth table that the result C is equal to 1 only when both the input variables A and B are 1, otherwise it is 0. The equation $A . B = C$ is normally read as "A and B equals C".

Table 6.2. Truth Table For Logical AND (.) Operator.

INPUTS		OUTPUT
A	B	= C
0	0	0
0	1	0
1	0	0
1	1	1

4. *Complementation.* The two operations defined so far (OR and AND) are binary operations because they define an operation on two variables. The complementation operation is a unary operation which is defined on a single variable.

The symbol $\bar{}$ is normally used for complementation operator. It is also known as 'NOT' operator. Thus we write \bar{A} , meaning "take the complement of A," or $\overline{(A + B)}$, meaning "take the complement of A + B." The complementation of a variable is the reverse of its value. Thus, if $A = 0$ then $\bar{A} = 1$ and if $A = 1$ then $\bar{A} = 0$. The truth table for logical NOT ($\bar{}$) operator is shown in Table 6.3. \bar{A} is read as "complement of A" or "not of A".

Table 6.3. Truth Table For Logical NOT ($\bar{}$) Operator.

INPUT	OUTPUT
A	\bar{A}
0	1
1	0

5. *Operator precedence.* Does $A + B \cdot C$ mean $(A + B) \cdot C$ or $A + (B \cdot C)$? The two generate different values for $A = 1, B = 0,$ and $C = 0,$ for then we have $(1 + 0) \cdot 0 = 0$ and $1 + (0 \cdot 0) = 1,$ which differ. Hence it is necessary to define operator precedence in order to correctly evaluate Boolean expressions. The precedence of Boolean operators is as follows :

1. The expression is scanned from left to right.
2. Expressions enclosed within parentheses are evaluated first.
3. All complement (NOT) operations are performed next.
4. All \cdot (AND) operations are performed after that.
5. Finally all $+$ (OR) operations are performed in the end.

So according to this precedence rule, $A + \bar{B} \cdot C$ means $A + (B \cdot C)$. Similarly for the expression $\bar{A} \cdot B$, the complement of A and B are both evaluated first and the results are then ANDed. Again for the expression $\overline{(A + B)}$, the expression inside the parenthesis $(A + B)$ is evaluated first and the result so obtained is then complemented.

POSTULATES OF BOOLEAN ALGEBRA

Postulate 1 :

- (a) $A = 0$ if and only if A is not equal to 1
- (b) $A = 1$ if and only if A is not equal to 0

Postulate 2 :

- (a) $x + 0 = x$
- (b) $x \cdot 1 = x$

Postulate 3 : Commutative Law

- (a) $x + y = y + x$
- (b) $x \cdot y = y \cdot x$

Postulate 4 : Associative Law

- (a) $x + (y + z) = (x + y) + z$
- (b) $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

Postulate 5 : Distributive Law

- (a) $x \cdot (y + z) = x \cdot y + x \cdot z$
- (b) $x + y \cdot z = (x + y) \cdot (x + z)$

Postulate 6 :

- (a) $x + \bar{x} = 1$
- (b) $x \cdot \bar{x} = 0$

The postulates listed above are the basic axioms of the algebraic structure and need no proof. They are used to prove the theorems of Boolean algebra.

THE PRINCIPLE OF DUALITY

In Boolean algebra, there is a precise duality between the operators \cdot (AND) & $+$ (OR) and the digits 0 & 1. For instance, let us consider Table 6.4. We can see that the second row of the table is obtainable from the first row and vice-versa simply by interchanging \cdot with $+$ and 0 with 1 . This important property is known as the principle of duality in Boolean algebra.

The implication of this duality is that any theorem in Boolean algebra has dual obtainable by interchanging \cdot & $+$

with '1' and '0' with '1'. Thus if a particular theorem is proved, its dual theorem automatically holds and need not be proved separately.

Table 6.4. Illustrating The Principle Of Duality In Boolean Algebra.

	Column 1	Column 2	Column 3
Row 1	$1+1 = 1$	$1+0 = 0+1 = 1$	$0+0 = 0$
Row 2	$0.0 = 0$	$0.1 = 1.0 = 0$	$1.1 = 1$

THEOREMS OF BOOLEAN ALGEBRA

Some of the important theorems of Boolean algebra are stated below along with their proof.

Theorem 1 (Idempotent law)

$$(a) x + x = x$$

$$(b) x \cdot x = x$$

Proof of (a)

L.H.S.

$$= x + x$$

$$= (x + x) \cdot 1 \quad \text{by postulate 2(b)}$$

$$= (x + x) \cdot (x + \bar{x}) \quad \text{by postulate 6(a)}$$

$$= x + x \cdot \bar{x} \quad \text{by postulate 5(b)}$$

$$= x + 0 \quad \text{by postulate 6(b)}$$

$$= x \quad \text{by postulate 2(a)}$$

= R.H.S.

Proof of (b)

L.H.S.

$$= x \cdot x$$

$$= x \cdot x + 0 \quad \text{by postulate 2(a)}$$

$$= x \cdot x + x \cdot \bar{x} \quad \text{by postulate 6(b)}$$

$$= x \cdot (x + \bar{x}) \quad \text{by postulate 5(a)}$$

$$= x \cdot 1 \quad \text{by postulate 6(a)}$$

$$= x \quad \text{by postulate 2(b)}$$

= R.H.S.

Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of part (a). Any dual theorem can be similarly derived from the proof of its corresponding pair. Hence from now onwards, the proof of part (a) only will be given. Interested readers can apply the principle of duality to the various steps of the proof of part (a) to obtain the proof of part (b) for any theorem.

Theorem 2

$$(a) x + 1 = 1$$

$$(b) x \cdot 0 = 0$$

Proof of (a);

L.H.S.

$$= x + 1$$

$$= (x+1) \cdot 1 \quad \text{by postulate 2(b)}$$

$$= (x+1) \cdot (x + \bar{x}) \quad \text{by postulate 6(a)}$$

$$= x + 1 \cdot \bar{x} \quad \text{by postulate 5(b)}$$

$$= x + \bar{x} \cdot 1 \quad \text{by postulate 3(b)}$$

$$= x + \bar{x} \quad \text{by postulate 2(b)}$$

$$= 1 \quad \text{by postulate 6(a)}$$

= R.H.S.

Proof of (b) holds by duality.

Theorem 3 (Absorption law)

$$(a) x + x \cdot y = x$$

(b) $x \cdot (x+y) = x$

Proof of (a)

L.H.S.
 $= x + x \cdot y$
 $= x \cdot 1 + x \cdot y$ by postulate 2(b)
 $= x \cdot (1+y)$ by postulate 5(a)
 $= x \cdot (y+1)$ by postulate 3(a)
 $= x \cdot 1$ by theorem 2(a)
 $= x$ by postulate 2(b)
 = R.H.S.

Proof of (b) holds by duality.

Proof by the method of perfect induction. The theorems of Boolean algebra can also be proved by means of truth tables. In a truth table, both sides of the relation are checked to yield identical results for all possible combinations of variables involved. In principle, it is possible to enumerate all possible combinations of the variables involved because Boolean algebra deals with variables that can have only two values. This method of proving theorems is called exhaustive enumeration or perfect induction.

Table 6.5. Truth Table For Proving Theorem 3(a) By The Method Of Perfect Induction.

x	y	$x \cdot y$	$x + x \cdot y$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

For example, Table 6.5 is a truth table for proving Theorem 3(a) by the method of perfect induction. Similarly,

Table 6.6 proves Theorem 3(b) by the method of perfect induction.

Table 6.6. Truth Table For Proving Theorem 3(b) By The Method Of Perfect Induction.

x	y	$x + y$	$x \cdot (x + y)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Theorem 4 (Involution Law)

$\overline{\overline{x}} = x$

Proof

Table 6.7 proves this theorem by the method of perfect induction.

Table 6.7. Truth Table For Proving Theorem 4 By The Method Of Perfect Induction.

x	\overline{x}	$\overline{\overline{x}}$
0	1	0
1	0	1

Note that Theorem 4 has no dual since it deals with the NOT operator which is unary operator.

Theorem 5

(a) $x \cdot (\bar{x} + y) = x \cdot y$

(b) $x + \bar{x} \cdot y = x + y$

Proof of (a)

Table 6.8 proves this theorem by the method of perfect induction.

Table 6.8. Truth Table for Proving Theorem 5(a) By The Method Of Perfect Induction.

x	y	\bar{x}	$\bar{x} + y$	$x \cdot (\bar{x} + y)$	$x \cdot y$
0	0	1	1	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	1	0	1	1	1

Proof of (b)

Table 6.9 proves this theorem by the method of perfect induction.

Table 6.9. Truth Table For Proving Theorem 5(b) By The Method Of Perfect Induction.

x	y	\bar{x}	$\bar{x} \cdot y$	$x + \bar{x} \cdot y$	$x + y$
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	0	1	1

Theorem 6 (De Morgan's law)

(a) $\overline{x + y} = \bar{x} \cdot \bar{y}$

(b) $\overline{x \cdot y} = \bar{x} + \bar{y}$

Proof of (a)

Table 6.10 proves this theorem by the method of perfect induction.

Table 6.10. Truth Table For Proving Theorem 6(a) By The Method Of Perfect Induction.

x	y	$x + y$	$\overline{x + y}$	\bar{x}	\bar{y}	$\bar{x} \cdot \bar{y}$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Proof of (b)

Table 6.11 proves this theorem by the method of perfect induction.

Table 6.11. Truth Table For Proving Theorem 6(b) By The Method Of Perfect Induction.

x	y	$x \cdot y$	$\overline{x \cdot y}$	\bar{x}	\bar{y}	$\bar{x} + \bar{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Theorems 6(a) and 6(b) are important and very useful. They are known as De Morgan's laws. They can be extended to n variables as given below.

$$\overline{X_1 + X_2 + X_3 + \dots + X_n} = \overline{X_1} \cdot \overline{X_2} \cdot \overline{X_3} \cdot \dots \cdot \overline{X_n}$$

$$\overline{X_1 \cdot X_2 \cdot X_3 \cdot \dots \cdot X_n} = \overline{X_1} + \overline{X_2} + \overline{X_3} + \dots + \overline{X_n}$$

The basic Boolean identities are summarised in Table 6.12. It is suggested that the readers should become well conversant with the identities given in this table in order to use the algebra effectively.

Table 6.12. Summary Of Basic Boolean Identities.

SL. NO.	IDENTITIES	DUAL IDENTITIES
1.	$A + 0 = A$	$A \cdot 1 = A$
2.	$A + 1 = 1$	$A \cdot 0 = 0$
3.	$A + \overline{A} = 1$	$A \cdot A = A$
4.	$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
5.	$\overline{\overline{A}} = A$	
6.	$A + B = B + A$	$A \cdot B = B \cdot A$
7.	$(A + B) + C = A + (B + C)$	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
8.	$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + B \cdot C = (A + B) \cdot (A + C)$
9.	$A + A \cdot B = A$	$A \cdot (A + B) = A$
10.	$A + \overline{A} \cdot B = A + B$	$A \cdot (\overline{A} + B) = A \cdot B$
11.	$\overline{A + B} = \overline{A} \cdot \overline{B}$	$\overline{A \cdot B} = \overline{A} + \overline{B}$

BOOLEAN FUNCTIONS

A Boolean function is an expression formed with binary variables, the two binary operators OR and AND,

the unary operator NOT, parentheses and equal sign. For a given value of the variables, the value of the function can be either 0 or 1. For example, consider the equation

$$W = X + \overline{Y} \cdot Z$$

Here the variable W is a function of X , Y and Z . This is written as $W = f(X, Y, Z)$ and the right hand side of the equation is called an *expression*. The symbols X , Y and Z are referred to as *literals* of this function.

The above is an example of a Boolean function represented as an algebraic expression. A Boolean function may also be represented in the form of a truth table. The number of rows in the table will be equal to 2^n , where n is the number of literals (binary variables) used in the function. The combinations of 0's and 1's for each row of this table is easily obtained from the binary numbers by counting from 0 to $2^n - 1$. For each row of the table, there is a value for the function equal to either 0 or 1 which is listed in separate column of the table. Such a truth table for the function $W = X + \overline{Y} \cdot Z$ is shown in Table 6.13. Observe that there are eight (2^3) possible distinct combinations for assigning bits to three variables. The column labeled W is either a 0 or a 1 for each of these combinations. The table shows that out of eight, there are five different combinations for which $W = 1$.

Table 6.13. Truth Table For The Boolean Function $W = X + \overline{Y} \cdot Z$

X	Y	Z	W
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

The question now arises - is an algebraic expression for a given Boolean function unique? In other words, is it possible to find two algebraic expressions that specify the same Boolean function? The answer to this question is yes. As a matter of fact, the manipulation of Boolean algebra is

applied mostly to the problem of finding simpler expressions for a given expression. For example, let us consider the following two functions:

$$F_1 = \bar{x}\bar{y}z + \bar{x}y.z + x\bar{y}$$

$$F_2 = x\bar{y} + \bar{x}.z$$

Table 6.14. Truth Table For The Boolean Functions:

$$F_1 = \bar{x}\bar{y}z + \bar{x}y.z + x\bar{y} \text{ and}$$

$$F_2 = x\bar{y} + \bar{x}.z$$

x	y	z	F ₁	F ₂
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

The representation of these two functions in the form of truth table is shown in Table 6.14. From the table we find that the function F₁ is same as the function F₂ since both have identical 0's and 1's for each combination of values of the three binary variables x, y and z. In general, two functions of n binary variables are said to be equal if they have the same value for all possible 2ⁿ combinations of the n literals.

MINIMIZATION OF BOOLEAN FUNCTIONS

When a Boolean function is implemented with logic gates (discussed later in this chapter), each literal in the function designates an input to a gate and each term is implemented with a gate. Thus for a given Boolean function, the minimization of the number of literals and the number of terms will result in a circuit with less equipments. For example, since functions F₁ and F₂ of Table 6.14 are equal Boolean functions, it is more economical to implement the F₂ form than the F₁ form because the F₂ form contains fewer terms. To find simpler circuits, one must know how to manipulate Boolean functions to obtain equal

and simpler expressions. What constitutes the best form of Boolean function depends on the particular application. However, we will give consideration only to the criterion of equipment minimization which is achieved by literal minimization.

There are several methods used for minimizing the number of literals in a Boolean function. However, a discussion of all these methods is beyond the scope of this book. Hence here we will consider only the method of algebraic manipulations. Unfortunately, in this method, there are no specific rules or guidelines to be followed that will guarantee the final answer. The only method available is cut-and-try procedure employing the postulates, the basic theorems, and any other manipulation method which becomes familiar with use. The following examples illustrate this procedure.

Example 6.1. Simplify the following Boolean functions to a minimum number of literals.

(a) $x + \bar{x}.y$

(b) $x . (\bar{x} + y)$

(c) $\bar{x}\bar{y}.z + \bar{x}.y.z + x\bar{y}$

(d) $x.y + \bar{x}.z + y.z$

(e) $(x+y) . (\bar{x}+z) . (y+z)$

Solution :

(a)

$$x + \bar{x}.y$$

$$= (x + \bar{x}).(x + y) \quad \text{by postulate 5(b)}$$

$$= 1.(x + y) \quad \text{by postulate 6(a)}$$

$$= (x + y).1 \quad \text{by postulate 3(b)}$$

$$= x + y \quad \text{by postulate 2(b)}$$

(b)

$$x . (\bar{x} + y)$$

$$= x.\bar{x} + x.y \quad \text{by postulate 5(a)}$$

$$= 0 + x.y \quad \text{by postulate 6(b)}$$

$$= x \cdot y + 0 \quad \text{by postulate 3(a)}$$

$$= x \cdot y \quad \text{by postulate 2(a)}$$

(c)

$$\bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot z + x \cdot \bar{y}$$

$$= \bar{x} \cdot z \cdot (\bar{y} + y) + x \cdot \bar{y} \quad \text{by postulate 5(a)}$$

$$= \bar{x} \cdot z \cdot (y + \bar{y}) + x \cdot \bar{y} \quad \text{by postulate 3(a)}$$

$$= \bar{x} \cdot z \cdot 1 + x \cdot \bar{y} \quad \text{by postulate 6(a)}$$

$$= \bar{x} \cdot z + x \cdot \bar{y} \quad \text{by postulate 2(b)}$$

(d)

$$x \cdot y + \bar{x} \cdot z + y \cdot z$$

$$= x \cdot y + \bar{x} \cdot z + y \cdot z \cdot 1 \quad \text{by postulate 2(b)}$$

$$= x \cdot y + \bar{x} \cdot z + y \cdot z \cdot (x + \bar{x}) \quad \text{by postulate 6(a)}$$

$$= x \cdot y + \bar{x} \cdot z + y \cdot z \cdot x + y \cdot z \cdot \bar{x} \quad \text{by postulate 5(a)}$$

$$= x \cdot y + \bar{x} \cdot z + x \cdot y \cdot z + \bar{x} \cdot y \cdot z \quad \text{by postulate 3(b)}$$

$$= x \cdot y \cdot 1 + \bar{x} \cdot z + x \cdot y \cdot z + \bar{x} \cdot y \cdot z \quad \text{by postulate 2(b)}$$

$$= x \cdot y \cdot 1 + x \cdot y \cdot z + \bar{x} \cdot z + \bar{x} \cdot y \cdot z \quad \text{by postulate 3(a)}$$

$$= x \cdot y \cdot (1 + z) + \bar{x} \cdot z \cdot (1 + y) \quad \text{by postulate 5(a)}$$

$$= x \cdot y \cdot (z + 1) + \bar{x} \cdot z \cdot (y + 1) \quad \text{by postulate 3(a)}$$

$$= x \cdot y \cdot 1 + \bar{x} \cdot z \cdot 1 \quad \text{by theorem 2(a)}$$

$$= x \cdot y + \bar{x} \cdot z \quad \text{by postulate 2(b)}$$

(e)

$$(x + y) \cdot (\bar{x} + z) \cdot (y + z)$$

$$= (x + y) \cdot (\bar{x} + z) \quad \text{by duality from (d)}$$

are the dual of each other and use dual expressions in corresponding minimization steps. Function (c) shows the equality of the functions F_1 and F_2 of Table 6.14. Function (d) illustrates the fact that an increase in the number of literals sometimes leads to a final simpler expression. Observe that function (c) is the dual of function (d). Hence it is not minimized directly and can be easily derived from the dual of the steps used to derive function (d).

COMPLEMENT OF A FUNCTION

The complement of a function F is \bar{F} and is obtained by interchanging 0's for 1's and 1's for 0's in the truth table that defines the function. For example Table 6.15 defines the function $F = x \cdot \bar{y} + \bar{x} \cdot z$ and its complement \bar{F} .

Table 6.15. Truth Table For The Function $F = x \cdot \bar{y} + \bar{x} \cdot z$ And Its Complement \bar{F} .

x	y	z	F	\bar{F}
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

Algebraically, the complement of a function may be derived through De Morgan's theorems whose generalized forms are as follows:

$$\overline{A_1 + A_2 + A_3 + \dots + A_n} = \bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 \dots \bar{A}_n$$

$$\overline{A_1 \cdot A_2 \cdot A_3 \dots A_n} = \bar{A}_1 + \bar{A}_2 + \bar{A}_3 + \dots + \bar{A}_n$$

These theorems state that the complement of a function is obtained by interchanging the OR and the AND operators and complementing each literal. The method is illustrated below through an example.

Example 6.2. Find the complement of the following functions:

(a) $F_1 = \bar{x} \cdot y \cdot \bar{z} + \bar{x} \cdot y \cdot z$

(b) $F_2 = x \cdot (\bar{y} \cdot \bar{z} + y \cdot z)$

Note that in the Example 6.1, functions (a) and (b)

Solution :

Applying De Morgan's theorems as many times as necessary, the complements are obtained as follows :

(a)

$$\begin{aligned}\overline{F_1} &= \overline{\overline{x.y.z} + \overline{x.y.z}} \\ &= \overline{(\overline{x.y.z}) . (\overline{x.y.z})} \\ &= \overline{(\overline{x+y+z}) . (\overline{x+y+z})} \\ &= (x+y+z) . (x+y+z)\end{aligned}$$

(b)

$$\begin{aligned}\overline{F_2} &= \overline{x . (\overline{y.z} + y.z)} \\ &= \overline{x} + \overline{(\overline{y.z} + y.z)} \\ &= \overline{x} + (\overline{y.z}) . (y.z) \\ &= \overline{x} + (\overline{y+z}) . (y+z) \\ &= \overline{x} + (y+z) . (y+z)\end{aligned}$$

A simpler procedure for deriving the complement of a function is to take the dual of the function and then complement each literal. This method follows from the generalized De Morgan's theorems. Remember that the dual of a function is obtained by interchanging OR and AND operators and 0's and 1's. The method is illustrated below with the help of an example.

Example 6.3. Find the complement of the functions F_1 and F_2 of Example 6.2. by taking their dual and complementing each literal.

Solution :

(a)

$$F_1 = \overline{x.y.z} + \overline{x.y.z}$$

The dual of F_1 is: $(\overline{x+y+z}) . (\overline{x+y+z})$

Complementing each literal we get

$$\overline{F_1} = (x+\overline{y+z}) . (x+y+\overline{z})$$

(b)

$$F_2 = x . (\overline{y.z} + y.z)$$

The dual of F_2 is: $x + (\overline{y+z}) . (y+z)$

Complementing each literal we get

$$\overline{F_2} = \overline{x} + (y+z) . (\overline{y+z})$$

CANONICAL FORMS FOR BOOLEAN FUNCTIONS

Minterms and Maxterms. A binary variable may appear either in its normal form (x) or in its complement form (\overline{x}). Now consider two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations :

$$\overline{x.y}, \overline{x.y}, x.\overline{y}, x.y$$

Each of these four AND terms is called a *minterm* or a *standard product*.

In a similar manner, n variables can be combined to form 2^n minterms. The 2^n different minterms may be determined by a method similar to the one shown in Table 6.16 below for three variables. The binary numbers from 0 to $2^n - 1$ are listed under the n variables. Each minterm is obtained from an AND term of the n variables, with each variable being primed if the corresponding bit of the binary number is 0 and unprimed if a 1.

Table 6.16. Minterms And Maxterms For Three Variables.

VARIABLES			MINTERMS		MAXTERMS	
X	Y	Z	TERM	DESIGNATION	TERM	DESIGNATION
0	0	0	$\overline{x.y.z}$	m_0	$x+y+z$	M_0
0	0	1	$\overline{x.y.z}$	m_1	$x+y+\overline{z}$	M_1
0	1	0	$\overline{x.y.z}$	m_2	$x+\overline{y}+z$	M_2
0	1	1	$\overline{x.y.z}$	m_3	$x+\overline{y}+\overline{z}$	M_3
1	0	0	$x.\overline{y.z}$	m_4	$\overline{x}+y+z$	M_4
1	0	1	$x.\overline{y.z}$	m_5	$\overline{x}+y+\overline{z}$	M_5
1	1	0	$x.y.\overline{z}$	m_6	$\overline{x}+\overline{y}+z$	M_6
1	1	1	$x.y.z$	m_7	$\overline{x}+\overline{y}+\overline{z}$	M_7

A symbol for each minterm is also shown in the table and is of the form m_j , where j denotes the decimal equivalent of the binary number of the minterm designated.

In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide 2^n possible combinations called *maxterms* or *standard sums*.

The eight maxterms for three variables, together

with their symbolic designation, are listed in Table 6.16. Any 2^n maxterms for n variables may be determined similarly. Each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is 0 and primed if it is a 1.

Note that each maxterm is the complement of its corresponding minterm and vice-versa.

Sum-of-Products

A sum-of-products expression is a product term (minterm) or several product terms (minterms) logically added (ORed) together.

For example, the expression $x\bar{y} + \bar{x}y$ is a sum-of-products expression. The following are all sum-of-products expressions:

- x
- $x+y$
- $x + y.z$
- $x.y + z$
- $\bar{x}.y + x.\bar{y}.\bar{z}$

The following steps are followed to express a Boolean function in its sum-of-products form :

1. Construct a truth table for the given Boolean function.
2. Form a minterm for each combination of the variables which produces a 1 in the function.
3. The desired expression is the sum (OR) of all the minterms obtained in step 2.

For example, in case of function F_1 of Table 6.17, the following three combinations of the variables produce a 1:

001, 100 and 111

Their corresponding minterms are

$x.y.z, x.y.\bar{z}$ and $x.\bar{y}.z$

Hence, taking the sum (OR) of all these minterms, the function F_1 can be expressed in its sum-of-products form as:

$$F_1 = \bar{x}.\bar{y}.z + x.\bar{y}.\bar{z} + x.y.z$$

or

$$F_1 = m_1 + m_4 + m_7$$

Similarly, it may be easily verified that the function F_2 of Table 6.17 can be expressed in its sum-of-products form as:

$$F_2 = \bar{x}.y.z + x.\bar{v}.z + x.y.\bar{z} + x.y.z$$

or

$$F_2 = m_3 + m_5 + m_6 + m_7$$

Table 6.17. Truth Table For Functions F_1 And F_2 .

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

It is sometimes convenient to express a Boolean function in its sum-of-products form. If not in this form, it can be made so by first expanding the expression into a sum of AND terms. Each term is then inspected to see if it contains all the variables. If it misses one or more variables, it is ANDed with an expression of the form $(x+\bar{x})$, where x is one of the missing variables. The following example clarifies this procedure.

Example 6.4. Express the Boolean function $F = A + \bar{B}.C$ in the sum-of-minterms (products) form.

Solution :

The function has three variables A, B and C. The first term A is missing two variables, therefore;

$$A = A.(B+\bar{B}) = A.B + A.\bar{B}$$

This is still missing one variable, so

$$\begin{aligned} A &= A.B.(C+\bar{C}) + A.\bar{B}.(C+\bar{C}) \\ &= A.B.C + A.B.\bar{C} + A.\bar{B}.C + A.\bar{B}.\bar{C} \end{aligned}$$

The second term $\bar{B}.C$ is missing one variable, therefore

$$\begin{aligned} \bar{B}.C &= \bar{B}.C.(A+\bar{A}) \\ &= A.\bar{B}.C + \bar{A}.\bar{B}.C \end{aligned}$$

So by combining all the terms we get

$$F = A.B.C + A.B.\bar{C} + A.\bar{B}.C + A.\bar{B}.\bar{C} + A.\bar{B}.C + \bar{A}.\bar{B}.C$$

But in the above expression, the term $A.\bar{B}.C$ appears twice and according to theorem 1(a) we have $x+x = x$. Hence it is possible to remove one of the them. Rearranging the minterms in ascending order, we finally obtain :

$$\begin{aligned} F &= \bar{A}.\bar{B}.C + A.\bar{B}.\bar{C} + A.\bar{B}.C + A.B.\bar{C} + A.B.C \\ &= m_1 + m_4 + m_5 + m_6 + m_7 \end{aligned}$$

It is sometimes convenient to express the Boolean function, when in its sum-of-minterms, in the following short notation:

$$F(A,B,C) = \sum(1,4,5,6,7)$$

The summation symbol \sum stands for the ORing of terms. The numbers following it are the minterms of the function. And finally, the letters in parentheses with F form a list of the variables in the order taken when the minterm is converted to an AND term.

Product-of-Sums

A product-of-sums expression is a sum term (maxterm) or several sum terms (maxterms) logically multiplied (ANDed) together. For example, the expression $(\bar{x}+y).(x+\bar{y})$ is a product of sums expression. The following are all product-of-sums expressions:

$$\begin{aligned} &x \\ &(\bar{x}+y) \\ &(\bar{x}+\bar{y}).z \\ &(x+\bar{y}).(\bar{x}+v).(\bar{x}+\bar{y}) \\ &(x+y).(\bar{x}+y+z) \end{aligned}$$

The following steps are followed to express a Boolean function in its product-of-sums form:

1. Construct a truth table for the given Boolean function.
2. Form a maxterm for each combination of the variables which produce a 0 in the function.
3. The desired expression is the product (AND) of all the maxterms obtained in step 2.

For example in case of function F_1 of Table 6.17, the following five combinations of the variables produce a 0: 000, 010, 011, 101, and 110

Their corresponding maxterms are :

$$(x+y+z), (x+\bar{y}+z), (x+\bar{y}+\bar{z}), (\bar{x}+y+\bar{z}), \text{ and } (\bar{x}+\bar{y}+z)$$

Hence, taking the product (AND) of all these maxterms, the function F_1 can be expressed in its product-of-sums form as:

$$\begin{aligned} F_1 &= (x+y+z).(x+\bar{y}+z).(x+\bar{y}+\bar{z}).(\bar{x}+y+\bar{z}).(\bar{x}+\bar{y}+z) \\ \text{or} \\ F_1 &= M_0 . M_2 . M_3 . M_5 . M_6 \end{aligned}$$

Similarly, it may be easily verified that the function F_2 of Table 6.17 can be expressed in its product-of-sums form as :

$$\begin{aligned} F_2 &= (x+y+z).(x+y+\bar{z}).(x+\bar{y}+z).(\bar{x}+y+z) \\ \text{or} \\ F_2 &= M_0 . M_1 . M_3 . M_4 \end{aligned}$$

In order to express a Boolean function in its product-of-sums form, it must first be brought into a form of OR terms. This may be done by using the distributive law :

$$x + y.z = (x+y).(x+z)$$

Then any missing variable (say x) in each OR term is ORed with the form $x.\bar{x}$. This procedure is clarified by the following example :

Example 6.5. Express the Boolean function

$$\begin{aligned} F &= x.y + \bar{x}.z \\ &\text{in the product-of-maxterms (sums) form.} \end{aligned}$$

Solution:

At first we convert the function into OR terms using

the distributive law:

$$\begin{aligned} F &= x.y + \bar{x}.z \\ &= (x.y + \bar{x}).(x.y + z) \\ &= (x + \bar{x}).(y + \bar{x}).(x + z).(y + z) \\ &= (\bar{x} + y).(x + z).(y + z) \end{aligned}$$

The function has three variables x, y and z. Each OR term is missing one variable, therefore :

$$\bar{x} + y = \bar{x} + y + z.\bar{z} = (\bar{x} + y + z).(\bar{x} + y + \bar{z})$$

$$x + z = x + z + y.\bar{y} = (x + z + y).(x + z + \bar{y})$$

$$y + z = x.\bar{x} + y + z = (x + y + z).(\bar{x} + y + z)$$

Combining all the terms and removing those that appear more than once, we finally obtain :

$$\begin{aligned} F &= (x + y + z).(x + \bar{y} + z).(\bar{x} + y + z).(\bar{x} + y + \bar{z}) \\ &= M_0 . M_2 . M_4 . M_5 \end{aligned}$$

A convenient way to express this function is as follows:

$$F(x,y,z) = \prod(0,2,4,5)$$

The product symbol \prod denotes the ANDing of maxterms. The numbers following it are the maxterms of the function.

The sum-of-products and the product-of-sums form of Boolean expressions are known as *standard forms*. One prime reason for liking the sum-of-products or the product-of-sums expressions is their straightforward conversion to very nice gating networks which are more desirable from most implementation points of view. In their purest, nicest form they go into two-level networks, which are networks for which the longest path through which the signal must pass from input to output is two gates.

CONVERSION BETWEEN CANONICAL FORMS

The complement of a function expressed as the sum-of-minterms equals the sum-of-minterms missing from the original function. This is because the original function is expressed by those minterms that make the function equal to 1, while its complement is a 1 for those minterms for which the function is a 0. For example, the function

$$\begin{aligned} F(A,B,C) &= \sum(1,4,5,6,7) \\ &= m_1 + m_4 + m_5 + m_6 + m_7 \end{aligned}$$

has a complement that can be expressed as :

$$\begin{aligned} \bar{F}(A,B,C) &= \sum(0,2,3) \\ &= m_0 + m_2 + m_3 \end{aligned}$$

Now, if we take the complement of \bar{F} , by De Morgan's theorem we obtain F back in a different form :

$$\begin{aligned} F &= \overline{m_0 + m_2 + m_3} \\ &= \bar{m}_0 . \bar{m}_2 . \bar{m}_3 \\ &= M_0 . M_2 . M_3 \\ &= \prod(0,2,3) \end{aligned}$$

The last conversion follows from the definition of minterms and maxterms as shown in Table 6.16. From the table, it is clear that the following relation holds true :

$$\bar{m}_j = M_j$$

That is, the maxterm with subscript j is a complement of the minterm with the same subscript j, and vice versa.

The last example has demonstrated the conversion between a function expressed in sum-of-minterms and its equivalent in product-of-maxterms. A similar argument will show that the conversion between the product-of-maxterms and the sum-of-minterms is similar. We now state a general conversion procedure:

"To convert from one canonical form to another, interchange the symbol and list those numbers missing from the original form."

For example, the function

$$F(x,y,z) = \prod(0,2,4,5)$$

is expressed in the product-of-maxterms form. Its conversion to sum-of-minterms is :

$$F(x,y,z) = \sum(1,3,6,7)$$

Note that in order to find the missing terms, one must realize that the total number of minterms or maxterms is always 2^n , where n is the number of binary variables in the function.

LOGIC GATES

All operations within a computer are carried out by means of combinations of signals passing through standard blocks of built-in circuits that are known as logic gates. In other words, a *logic gate* is simply an electronic circuit

which operates on one or more input signals to produce standard output signals. These logic gates are the building blocks of all the circuits in a computer.

Computer circuits are built up using combinations of different types of logic gates to perform the necessary operation. There are several types of gates, but we shall consider here only some of the most important ones. These are sufficient to introduce the concept of circuit design using logic gates.

AND GATE

An AND gate is the physical realization of the logical multiplication (AND) operation. That is, it is an electronic circuit that generates an output signal of 1 only if all input signals are also 1

To have a conceptual idea, let us consider the case of Figure 6.1. Here two switches A and B are connected in series. It is obvious that the input current will reach the output point only when both the switches are in the on(1) state. There will be no output (output = 0) if either one or both the switches are in the off(0) state. So, two or more switches connected in series behave as an AND gate.



Figure 6.1. Two or more switches connected in series behave as an AND gate.

The behaviour of a logic gate, that is the state of its output signal depending on the various combinations of input signals, is conveniently described by means of a truth table. The truth table and the block diagram symbol for an AND gate for two input signals are shown in Figure 6.2. Since there are only two inputs (A & B), so only four (2^2) combinations of inputs are possible. Also observe from the truth table that an output of 1 is obtained only when both the inputs are in 1 state, otherwise it is 0.

OR GATE

An OR gate is the physical realization of the logical addition (OR) operation. That is, it is an electronic circuit that generates an output signal of 1 if any of the input signals is also 1.

Two or more switches connected in parallel behave as an OR gate. It can be seen from Figure 6.3, that the input current will reach the output point when any one of the two switches are in the on(1) state. There will be no output only when both the switches (A & B) are in the off(0) state.

The truth table and the block diagram symbol for an OR gate for two input signals are shown in Figure 6.4. Observe that an output of 1 is obtained when any of the input signals is 1. Output is 0 only when both the inputs are 0.



INPUTS		OUTPUT
A	B	$C = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Figure 6.2. Block diagram symbol and truth table for an AND gate.

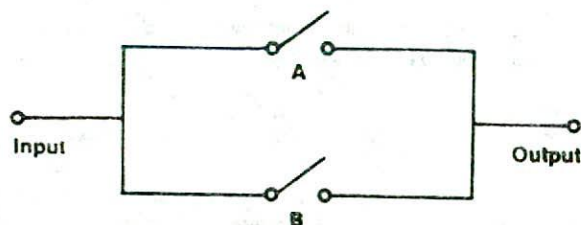
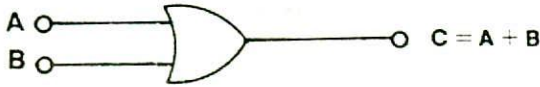


Figure 6.3. Two or more switches connected in parallel behave as an OR gate.



INPUTS		OUTPUT
A	B	$C = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Figure 6.4. Block diagram symbol and truth table for an OR gate.

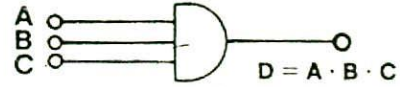
Just as the + and \cdot operations could be extended to several variables using the associative law, AND gates and OR gates can have more than two inputs. Figure 6.5 shows three input AND and OR gates and the table of all input combinations for each. As might be hoped, the output of the AND gate with inputs A, B, and C is a 1 only if A and B and C are 1, i.e., when all three of the inputs are 1, so that we write the output as $A \cdot B \cdot C$. Similarly, the OR gate with inputs A, B, and C has a 1 output if A or B or C is a 1, so that we can write $A + B + C$ for its output.

The above argument can be extended. A four-input AND gate has a 1 output only when all four inputs are 1, and a four-input OR gate has a 1 output when any of its inputs is a 1.

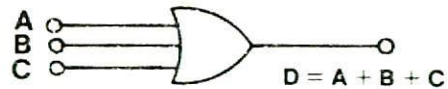
NOT GATE

A NOT gate is the physical realization of the complementation operation. That is, it is an electronic circuit that generates an output signal which is the reverse of the input signal. A NOT gate is also known as an *inverter* because it inverts the input.

The truth table and the block diagram symbol for a NOT gate are shown in Figure 6.6. Recall that the complementation operation is unary operation which is defined on a single variable. Hence a NOT gate always has a single input. Figure 6.6 shows also that connecting two NOT gates in series gives an output equal to the input, and this is the gating counterpart to the law of the double complementation, $\overline{\overline{A}} = A$.

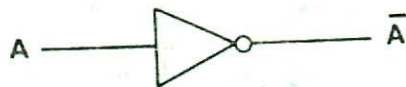


INPUTS			OUTPUT
A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



INPUTS			OUTPUT
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figure 6.5. Three input AND and OR gates.



INPUT		OUTPUT
A		\bar{A}
0		1
1		0

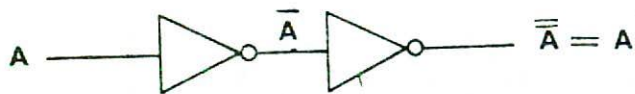
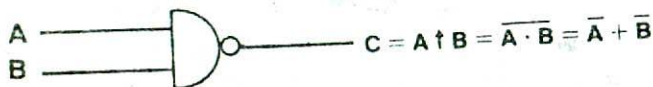


Figure 6.6. (a) Block diagram symbol and truth table for a NOT gate.

(b) Two NOT gates in series.



INPUTS		OUTPUT
A	B	$C = \bar{A} + \bar{B}$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 6.7. Block diagram symbol and truth table for a NAND gate.

NAND GATE

A NAND gate is a complemented AND gate. That is, the output of NAND gate will be a 1 if any one of the inputs is a 0 and will be a 0 only when all the inputs are 1.

The truth table and the block diagram symbol for a NAND gate are shown in Figure 6.7. The symbol '↑' is usually used to represent a NAND operation in boolean expressions. Thus, $A \uparrow B = \bar{A} \cdot \bar{B} = \bar{A+B}$.

The operation of a NAND gate can be analysed using the equivalent block diagram circuit shown in Figure 6.8, which has an AND gate followed by a NOT gate. For inputs A and B, the output of the AND gate will be $A \cdot B$ which is fed as input to the NOT gate. So the complement of $A \cdot B$ will be $\overline{A \cdot B}$ which is equal to $\bar{A} + \bar{B}$ or $A \uparrow B$. In fact, small circle on the output of the NAND gate (see Figure 6.7) represents complementation. The NAND gate can then be seen to be an AND gate followed by a NOT gate.



Figure 6.8. NAND gate realization with an AND gate and a NOT gate.

Multiple-input NAND gates can be analysed similarly. A three-input NAND gate with inputs A, B, and C will have an output equal to $\bar{A} \cdot \bar{B} \cdot \bar{C}$ or $\bar{A} + \bar{B} + \bar{C}$, which says that the output will be a 1 if any of the inputs is a 0 and will be a 0 only when all three inputs are 1.

NOR GATE

A NOR gate is a complemented OR gate. That is, the output of a NOR gate will be a 1 only when all inputs are 0 and it will be a 0 if any input represents a 1.

The truth table and the block diagram symbol for a NOR gate are shown in Figure 6.9. The symbol '↓' is usually used to represent a NOR operation in Boolean expressions. Thus, $A \downarrow B = \overline{A+B} = \bar{A} \cdot \bar{B}$.

The operation of a NOR gate can be analysed using the equivalent block diagram circuit shown in Figure 6.10, which has an OR gate followed by a NOT gate. For inputs A and B, the output of the OR gate will be $A + B$ which is fed as input to the NOT gate. So the complement of $A + B$ will be $\overline{A+B}$ which is equal to $\bar{A} \cdot \bar{B}$ or $A \downarrow B$. In fact, the

small circle on the output of the NOR gate (see Figure 6.9) represents complementation. The NOR gate can then be seen to be an OR gate followed by a NOT gate.



INPUTS		OUTPUT
A	B	$C = \overline{A \cdot B}$
0	0	1
0	1	0
1	0	0
1	1	0

Figure 6.9. Block diagram symbol and truth table for a NOR gate.

Multiple input NOR gates can be analysed similarly. A three-input NOR gate with inputs A, B, and C will have an output equal to $\overline{A + B + C}$ or $\overline{A \cdot B \cdot C}$, which says that the output will be a 1 only when all the three inputs are 0 and it will be a 0 if any of the three inputs is a 1.

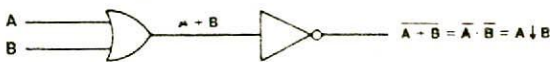
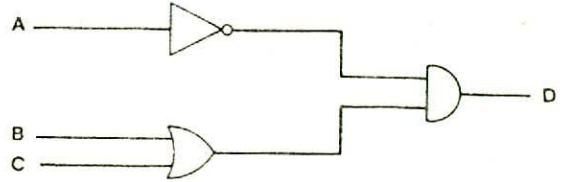


Figure 6.10. NOR gate realization with an OR gate and a NOT gate.

LOGIC CIRCUITS

The logic gates described in the previous section are seldom used by themselves but are used in combinations. They are interconnected to form gating, or logic, networks which are known as combinational logic circuits. For these logic circuits, the Boolean algebra expression can be derived by systematically progressing from input to output on the gates. Few examples are given below.

Example 6.6. Find the Boolean expression for the output of the logic circuit given below.



Solution :

Input A is fed to the NOT gate whose output will be \overline{A} .

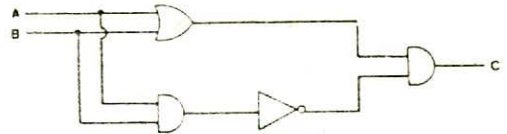
Inputs B and C are fed to the OR gate whose output will be $B + C$.

Now these two outputs (\overline{A} and $B + C$) are fed as input to the AND gate. So the output produced by the AND gate will be $\overline{A} \cdot (B + C)$

$$\text{Hence } D = \overline{A} \cdot (B + C)$$

which is the required Boolean expression for the output of the given logic circuit.

Example 6.7. Find the logic equation for the output produced by the logic circuit given below.



Solution :

The output of the OR gate is

$$A + B \text{ ----- (a)}$$

The output of the first AND gate is

$$A \cdot B \text{ ----- (b)}$$

Since the expression (b) is fed as input to the NOT gate. So the output of the NOT gate is

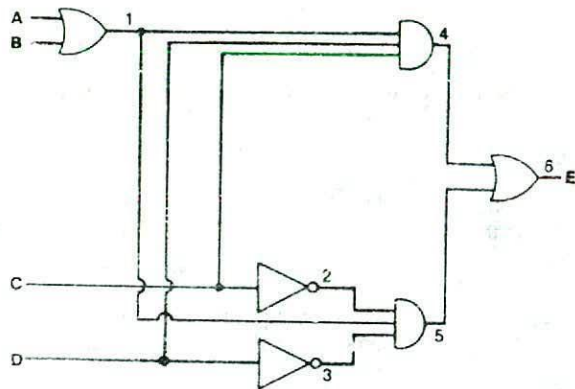
$$\overline{A \cdot B} \text{ ----- (c)}$$

Now expressions (a) and (c) are fed as input to the second AND gate. So its output will be

$$(A+B) \cdot (\overline{A \cdot B})$$

Hence $C = (A+B) \cdot (\overline{A \cdot B})$ which is the desired logic equation for the output produced by the given logic circuit.

Example 6.8. Find the Boolean expression for the output of the logic circuit given below.



Solution.

At point 1, the output of the OR gate is

$$A+B \text{ ----- (a)}$$

At point 2, the output of the NOT gate is

$$\overline{C} \text{ ----- (b)}$$

At point 3, the output of the NOT gate is

$$\overline{D} \text{ ----- (c)}$$

The inputs to the AND gate at point 4 are (A+B), C, and D. Hence at point 4, the output of the AND gate is

$$(A+B) \cdot C \cdot D \text{ ---- (d)}$$

The inputs to the AND gate at point 5 are (A+B), \overline{C} , and \overline{D} . Hence at point 5, the output of the AND gate is

$$(A+B) \cdot \overline{C} \cdot \overline{D} \text{ ---- (e)}$$

Finally, the inputs to the OR gate at point 6 are (d) and (e). Hence at point 6, the output of the OR gate is

$$(A+B) \cdot C \cdot D + (A+B) \cdot \overline{C} \cdot \overline{D}$$

So $E = (A+B) \cdot C \cdot D + (A+B) \cdot \overline{C} \cdot \overline{D}$

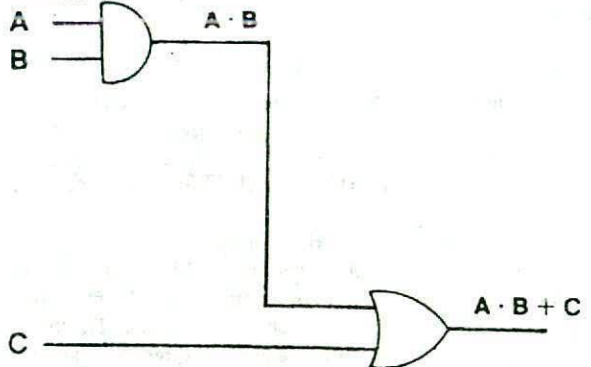
which is the required Boolean expression for the output of the given logic circuit.

CONVERTING EXPRESSIONS TO LOGIC CIRCUITS

We have just now considered few examples that illustrate the method of deriving Boolean expression for a given logic circuit. The reverse problem of constructing a logic circuit for a given Boolean expression is also not difficult. The three logic gates - AND, OR, and NOT are said to be logically complete because any Boolean expression may be realized using only these three gates. The method of constructing logic circuits for Boolean expressions using only these three gates is illustrated below with the help of some examples.

Example 6.9. Construct a logic circuit for the Boolean expression $A \cdot B + C$.

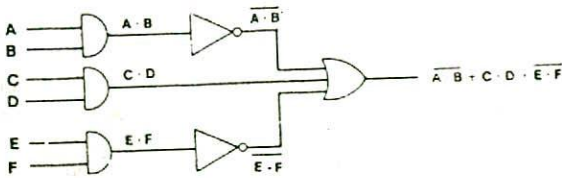
Solution :



The desired logic circuit is shown above which is self explanatory.

Example 6.10. Construct a logic circuit for the Boolean expression $\overline{A \cdot B} + C \cdot D + \overline{E \cdot F}$

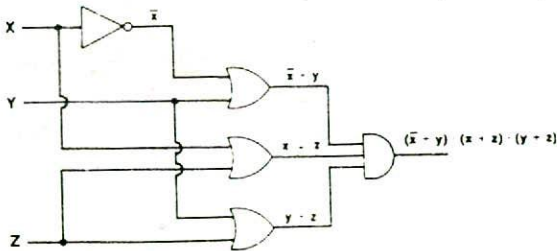
Solution :



The desired logic circuit is shown above which is self explanatory.

Example 6.11. Construct a logic circuit for the Boolean expression $(\bar{x}+y) \cdot (x+z) \cdot (y+z)$

Solution :



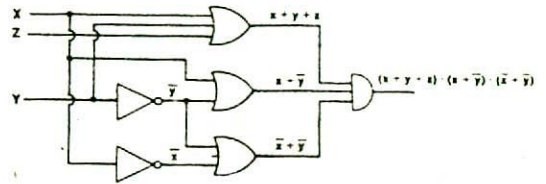
The desired logic circuit is shown above which is self explanatory.

THE UNIVERSAL NAND GATE

We have seen that AND, OR, and NOT gates are logically complete in the sense that any Boolean function may be realized using these three gates. However, the NAND gate, which was introduced in the previous section, is said to be universal gate because it is alone sufficient to implement any Boolean function.

Example 6.12. Construct a logic circuit for the Boolean expression $(x+y+z) \cdot (x+y) \cdot (\bar{x}+\bar{y})$

Solution :



The desired logic circuit is shown above which is self explanatory.

To show that any Boolean function can be implemented with the sole use of NAND gates, we need only show that the logical operations AND, OR, and NOT can be implemented with NAND gates. This is shown in Figure 6.11 below.

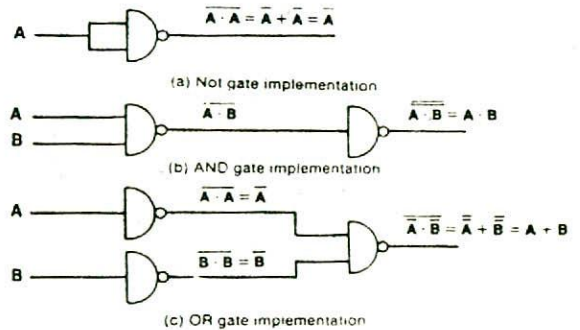


Figure 6.11. Implementation of NOT, AND, and OR gates by NAND gates.

A NOT operation is obtained from a one-input NAND gate. Thus we find that a single-input NAND gate behaves as an inverter.

The AND operation requires two NAND gates. The first one produces the inverted AND and the second one

being a single input NAND gate, acts as an inverter to obtain the normal AND output.

For the OR operation, the normal inputs A and B are first complemented using two single input NAND gates. Now the complemented variables are fed as input to another NAND gate which produces the normal ORed output.

The implementation of Boolean functions with NAND gates may be obtained by means of a simple block diagram manipulation technique. The method requires that two other logic diagrams be drawn prior to obtaining the NAND logic diagram. The following steps are to be carried out in sequence :

- Step 1: From the given algebraic expression, draw the logic diagram with AND, OR, and NOT gates. Assume that both the normal (A) and complement (\bar{A}) inputs are available.
- Step 2: Draw a second logic diagram with the equivalent NAND logic substituted for each AND, OR, and NOT gate.
- Step 3: Remove any two cascaded inverters from the diagram since double inversion does not perform any logical function. Also remove inverters connected to single external inputs and complement the corresponding input variable. The new logic diagram so obtained is the required NAND gate implementation of the Boolean function.

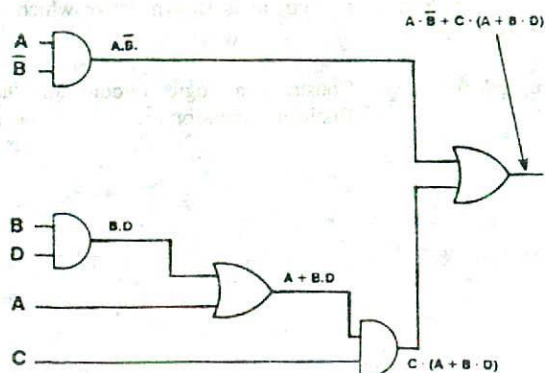
Example 6.13. Construct a logic circuit for the Boolean expression $A \cdot \bar{B} + C \cdot (A + B \cdot D)$ using only NAND gates.

Solution :

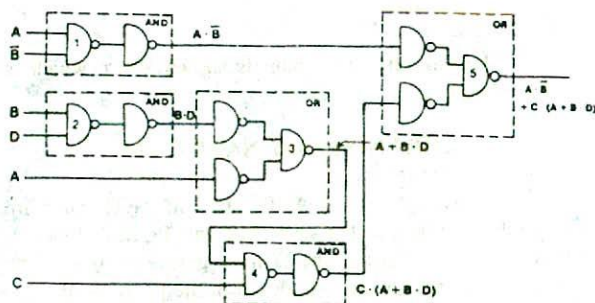
The AND/OR implementation for the given Boolean expression is drawn in Figure 6.12(a). Now each AND gate is substituted by a NAND gate followed by an inverter and each OR gate is substituted by two input inverters followed by a NAND gate. Thus each AND gate is substituted by two NAND gates and each OR gate is substituted by three NAND gates. The logic diagram so obtained is shown in Figure 6.12(b). Note that Figure 6.12(b) has seven inverters (single input NAND gates) and five two-input NAND gates. Each two-input NAND gate has a number inside the gate symbol for identification purpose. Pairs of inverters connected in cascade (from each AND box to each OR box)

are removed since they form double inversion which has no meaning. The inverter connected to input A is removed and the input variable is changed from A to \bar{A} . The result is the NAND logic diagram shown in Figure 6.12(c), with the number inside each NAND gate identifying the gate from Figure(b).

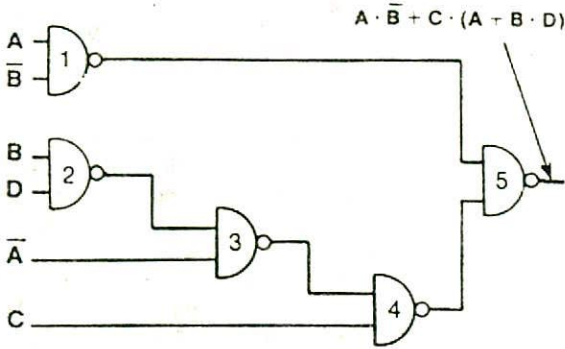
This example demonstrates that the number of NAND gates required to implement the Boolean function is equal to the number of AND/OR gates, provided both the normal and complement inputs are available. Otherwise inverters must be used to generate any required complemented inputs.



(a) AND/OR implementation.

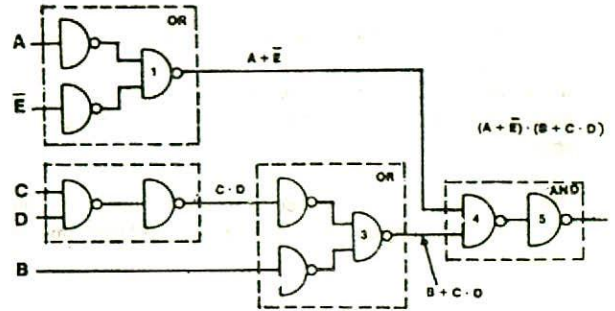


(b) Substituting equivalent NAND functions.



(c) NAND implementation.

Figure 6.12. Step-by-step NAND implementation for the Boolean expression of Example 6.13.

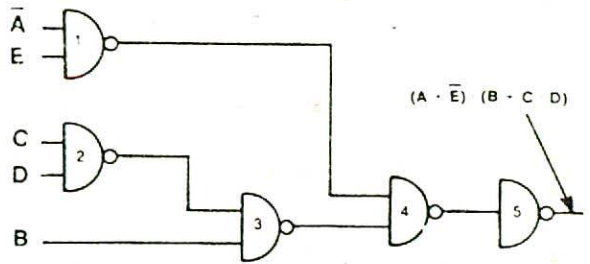


(b) Substituting equivalent NAND functions.

Example 6.14. Construct a logic circuit for the Boolean expression $(A + \bar{E}) \cdot (B + C \cdot D)$ using only NAND gates.

Solution :

The AND/OR implementation for the given Boolean expression is drawn in Figure 6.13(a). Now the NAND equivalent of each AND and each OR gate is substituted resulting in Figure 6.13(b). Note that Figure 6.13(b) has six inverters (single input NAND gates) and four two-input NAND gates. One pair of cascaded inverters may be removed. Also the three external inputs A, B, and \bar{E} , which go directly to inverters, are complemented and the corresponding inverters are removed. The final NAND gate implementation so obtained is shown in Figure 6.13(c). The number inside each NAND gate of Figure 6.13(c) corresponds to the NAND gate of Figure 6.13(b) having the same number.



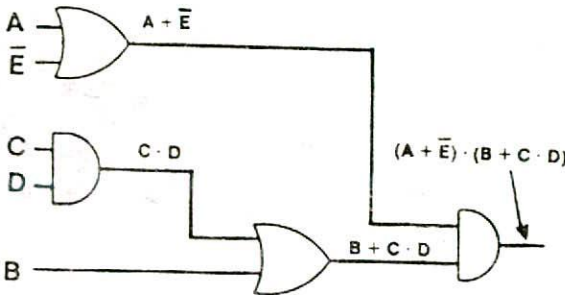
(c) NAND implementation.

Figure 6.13. Step-by-step NAND implementation for the Boolean expression of Example 6.14.

For this example, the number of NAND gates required is equal to the number of AND/OR gates plus an additional inverter at the output (NAND gate number 5). In general, the number of NAND gates required to implement a Boolean function equals the number of AND/OR gates, except for an occasional inverter. This is true only when both normal and complemented inputs are available because the conversion forces certain input variables to be complemented.

THE UNIVERSAL NOR GATE

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic form a dual of the corresponding procedures and rules



(a) AND/OR implementation.

developed from NAND logic. Like the NAND gate, the NOR gate is also universal because it is alone sufficient to implement any Boolean function.

To show that any Boolean function can be implemented with the sole use of NOR gates, we need only show that the logical operations AND, OR, and NOT can be implemented with NOR gates. This is shown in Figure 6.14 below.

The NOT operation is obtained from a one-input NOR gate. Thus, a single input NOR gate is yet another inverter circuit.

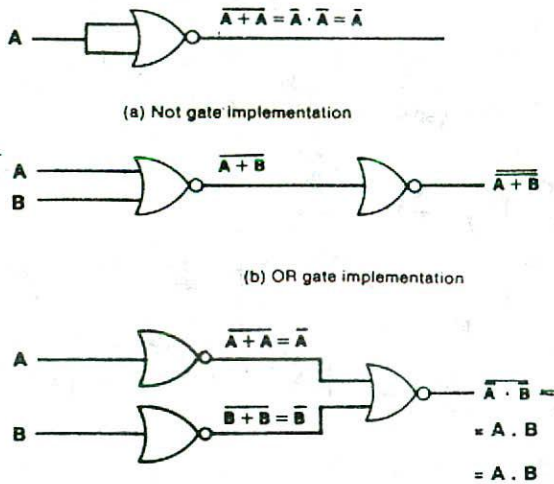


Figure 6.14. Implementation of NOT, OR and AND gates by NOR gates.

The OR operation requires two NOR gates. The first one produces the inverted OR and the second one being a single input NOT gate, acts as an inverter to obtain the normal OR output.

The AND operation is achieved through a NOR gate with additional inverters in each input.

Similar to the NAND logic diagram, the implementation of Boolean functions with NOR gates may be obtained by carrying out the following steps in sequence

Step 1: For the given algebraic expression, draw the logic diagram with AND, OR and NOT gates. Assume that both the normal (A) and complement (\bar{A}) inputs are available.

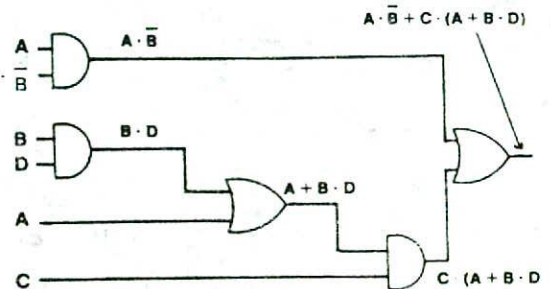
Step 2: Draw a second logic diagram with equivalent NOR logic substituted for each AND, OR, and NOT gate.

Step 3: Remove any two cascaded inverters from the diagram since double inversion does not perform any logical function. Also remove inverters connected to single external inputs and complement the corresponding input variable. The new logic diagram so obtained is the required NAND gate implementation of the given Boolean function.

Example 6.15. Construct a logic diagram for the Boolean expression $A \cdot \bar{B} + C \cdot (A + B \cdot D)$ using only NOR gates.

Solution :

The AND/OR implementation for the given Boolean expression is shown in Figure 6.15(a). Now each OR gate is substituted by a NOR gate followed by an inverter and each AND gate is substituted by two input inverters followed by a NOR gate. Thus each OR gate is substituted by two NOR gates and each AND gate is substituted by three NOR gates. The logic diagram so obtained is shown in Figure 6.15(b). Note that Figure 6.15(b) has eight inverters (single input NOR gates) and five two-input NOR gates. One pair of cascaded inverters (from the OR box to the AND box) may be removed. Also the five external inputs A, \bar{B} , B, D and C, which go directly to inverters, are complemented and the corresponding inverters are removed.



(a) AND/OR implementation.

The final NOR gate implementation so obtained is shown in Figure 6.15(c). The number inside each NOR gate of Figure 6.15(c) corresponds to the NOR gate of Figure 6.15(b) having the same number.

The number of NOR gates in this example equals the number of AND/OR gates plus an additional inverter in the output (NOR gate number 6). In general, the number of NOR gates required to implement a Boolean function equals the number of AND/OR gates, except for an occasional inverter. This is true only if both normal and complemented inputs are available because the conversion forces certain input variables to be complemented.

Combinational circuits are more frequently constructed with NAND or NOR gates than with AND, OR and NOT gates. NAND and NOR gates are more popular than the AND and OR gates because NAND and NOR gates are easily constructed with transistor circuits and Boolean functions can be easily implemented with them. Moreover, NAND and NOR gates are superior to AND and OR gates from the hardware point of view, as they supply outputs that maintain the signal value without loss of amplitude. OR and AND gates sometimes need amplitude restoration after the signal travels through a few levels of gates.

EXCLUSIVE-OR AND EQUIVALENCE FUNCTIONS

Exclusive-or and equivalence, denoted by \oplus and \odot respectively, are binary operations that perform the following Boolean functions:

$$A \oplus B = A\bar{B} + \bar{A}B$$

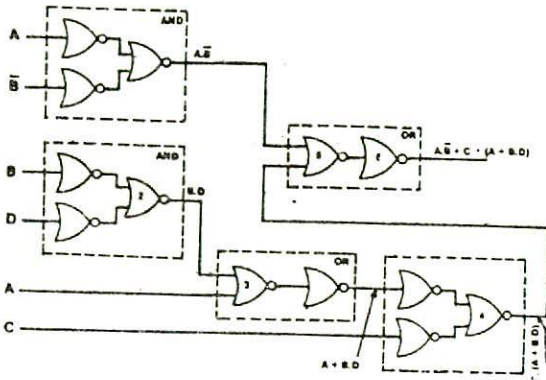
$$A \odot B = AB + \bar{A}\bar{B}$$

The truth table and the block diagram symbol for the exclusive-or and the equivalence operations are shown in Figure 6.16 and Figure 6.17 respectively. Observe that the two operations are the complement of each other. Each is commutative and associative. Because of these two properties, a function of three or more variables can be expressed without parentheses as follows:

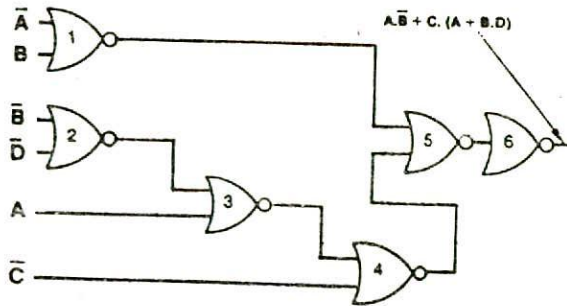
$$(A \oplus B) \oplus C = A \oplus (B \oplus C)$$

$$= A \oplus B \oplus C$$

The exclusive-or and equivalence operations have many excellent characteristics as candidates for logic gates but are expensive to construct with physical components. They are available as standard logic gates in IC packages but are usually constructed internally with other standard



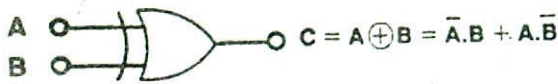
(b) Substituting equivalent NOR functions.



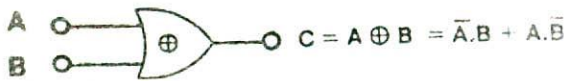
(c) NOR implementation.

Figure 6.15. Step-by-step NOR implementation for the Boolean expression of Example 6.15.

gates. For example, Figure 6.18(a) shows the implementation of a two-input exclusive-or function with AND, OR and NOT gates. Figure 6.18(b) shows its implementation with NAND gates.

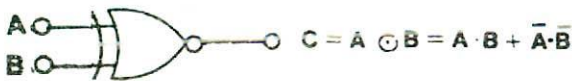


or



INPUTS		OUTPUT
A	B	C = A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

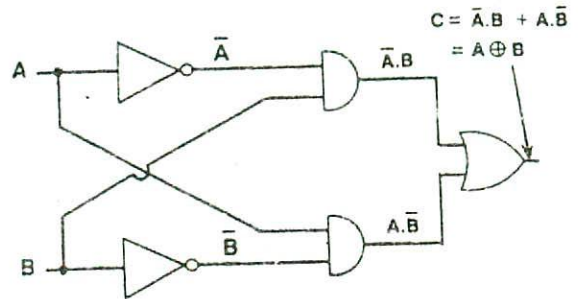
Figure 6.16. Block diagram symbol and truth table for an EXCLUSIVE-OR operation.



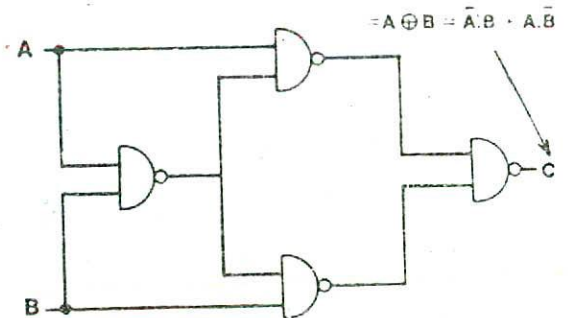
INPUTS		OUTPUT
A	B	C = A ⊙ B
0	0	1
0	1	0
1	0	0
1	1	1

Figure 6.17. Block diagram symbol and truth table for an EQUIVALENCE operation.

Only a limited number of Boolean functions can be expressed exclusively in terms of exclusive-or or equivalence operations. Nevertheless, these functions emerge quite often during the design of digital systems. The two functions are particularly useful in arithmetic operations and error detection and correction.



(a) Implementation with AND/OR/NOT gates.



(b) Implementation with NAND gates.

Figure 6.18. Logic diagrams of EXCLUSIVE-OR function.

DESIGN OF COMBINATIONAL CIRCUITS

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram.

The procedure involves the following steps:

1. State the given problem completely and exactly.

2. Interpret the problem and determine the available input variables and required output variables.
3. Assign a letter symbol to each input and output variables.
4. Design the truth table that defines the required relations between inputs and outputs.
5. Obtain the simplified Boolean function for each output.
6. Draw the logic circuit diagram to implement the Boolean function.

The simplified Boolean functions for the two outputs, directly obtained from the truth table, are :

$$S = \bar{A}.B + A.\bar{B}$$

$$C = A.B$$

The logic circuit diagram to implement this is shown in Figure 6.20.

INPUTS		OUTPUTS	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 6.19. Truth table for a half-adder.

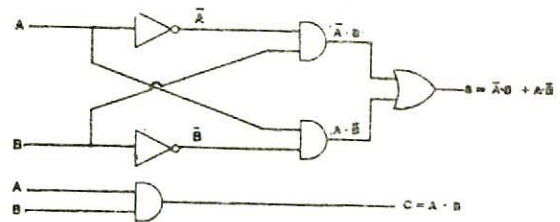


Figure 6.20. Logic circuit diagram for a half-adder.

To illustrate the design procedure, we will design adder circuits because addition is the most basic arithmetic operation for any computer system.

Addition in binary system can be summarized by the following four rules :

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

The first three operations produce a sum whose length is one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a *carry*. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half-adder*. One that performs the addition of three bits (two significant bits and previous carry) is called a *full-adder*. The name of the former stems from the fact that two half-adders can be employed to implement a full-adder.

DESIGN OF HALF-ADDER

From the definition of a half-adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits whereas the output variables produce the sum and carries bits. Let A and B be the two inputs and S (for sum) and C (for carry) be the two outputs. The truth table of Figure 6.19 exactly defines the function of the half-adder.

The half-adder is limited in the sense that it can add only two single bits. Although it generates a carry for the next higher pair of significant bits, it cannot accept a carry generated from the previous pair of lower significant bits. A full-adder solves this problem.

DESIGN OF FULL-ADDER

A full-adder forms the arithmetic sum of three input bits. Thus it consists of three inputs and two outputs. Two of the input variables (A and B) represent the augend and the addend bits and the third input variable (D) represents the carry from the previous lower significant position. Two outputs are necessary because the sum of three binary digits ranges in value from 0 to 3, and binary 2 and 3 need two digits. These two outputs are designated by the symbols S (for sum) and C (for carry). The truth-table of Figure 6.21 exactly defines the function of full-adder. The 1's and 0's for the output variables are determined from the arithmetic sum of the three input variables. When all input variables are 0, the output is 0 for both C and S. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output is 1 if two or three inputs are equal to 1.

INPUTS			OUTPUTS	
A	B	D	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 6.21. Truth table for a full-adder.

The sum-of-products expressions for the two outputs can be directly obtained from the truth table and is given below :

$$S = \bar{A}\bar{B}D + \bar{A}B\bar{D} + A\bar{B}\bar{D} + A.B.D$$

$$C = \bar{A}B.D + A\bar{B}.D + A.B.\bar{D} + A.B.D$$

Although the expression for S cannot be simplified, it is possible to simplify the expression for C as follows :

$$C = \bar{A}B.D + A\bar{B}.D + A.B.\bar{D} + A.B.D$$

$$= \bar{A}B.D + A\bar{B}.D + A.B.\bar{D} + A.B.D + A.B.D + A.B.D$$

(since $x + x = x$)

$$= (\bar{A}B.D + A.B.D) + (A\bar{B}.D + A.B.D) + (A.B.\bar{D} + A.B.D)$$

$$= (A+\bar{A}).B.D + (B+\bar{B}).A.D + (D+\bar{D}).A.B$$

$$= B.D + A.D + A.B \text{ (since } x + \bar{x} = 1)$$

$$= A.B + A.D + B.D$$

Hence, finally we obtain the following expressions for the two outputs :

$$S = \bar{A}\bar{B}D + \bar{A}B\bar{D} + A\bar{B}\bar{D} + A.B.D$$

$$C = A.B + A.D + B.D$$

The logic circuit diagram to implement this is shown in Figure 6.22.

A full-adder can also be implemented with two half-adders and one OR gate as shown in Figure 6.23.

The S output from the second half-adder is the exclusive-or of D and the output of the first half-adder giving :

$$S = \overline{(\bar{A}B + A\bar{B})}.D + (\bar{A}B + A\bar{B}).D$$

$$= (\bar{A}B + A\bar{B}).D + \bar{A}B.D + A\bar{B}.D$$

$$= (\bar{A} + B).(\bar{A} + \bar{B}).D + \bar{A}B.D + A\bar{B}.D$$

$$= (A+\bar{B}).(\bar{A}+B).D + \bar{A}B.D + A\bar{B}.D$$

$$= (A.\bar{A} + A.B + \bar{A}.\bar{B} + B.\bar{B}).D + \bar{A}B.D + A\bar{B}.D$$

$$= (A.B + \bar{A}.\bar{B}).D + \bar{A}B.D + A\bar{B}.D$$

$$= A.B.D + \bar{A}B.D + \bar{A}B.D + A\bar{B}.D$$

$$= \bar{A}B.D + \bar{A}B.D + A\bar{B}.D + A.B.D$$

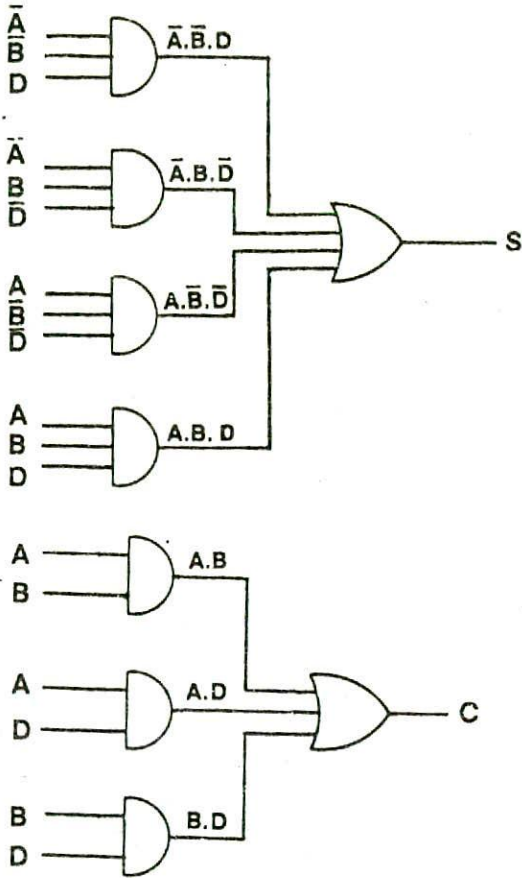


Figure 6.22. Logic Circuit diagram for a full-adder.

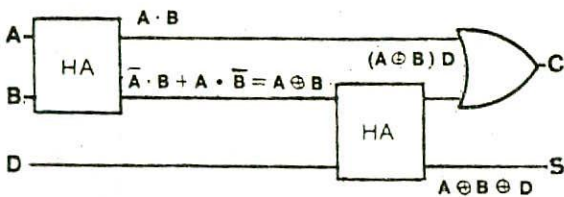


Figure 6.23. Implementation of full-adder with two half-adders and one OR gate.

And we have the carry output

$$\begin{aligned}
 C &= (\bar{A} \cdot B + A \cdot \bar{B}) \cdot D + A \cdot B \\
 &= \bar{A} \cdot B \cdot D + A \cdot \bar{B} \cdot D + A \cdot B \cdot (D + \bar{D}) \\
 &= \bar{A} \cdot B \cdot D + A \cdot \bar{B} \cdot D + A \cdot B \cdot D + A \cdot B \cdot \bar{D}
 \end{aligned}$$

This can be simplified as before to

$$C = A \cdot B + A \cdot D + B \cdot D$$

A PARALLEL BINARY ADDER

Parallel binary adders are used to add two binary numbers. For example, if we want to add two four-bit binary numbers, we need to construct a parallel four-bit binary adder as shown in Figure 6.24. Such an adder requires one half-adder (denoted by HA) and three full-adders (denoted by FA). The binary numbers being added are $A_4 A_3 A_2 A_1$ and $B_4 B_3 B_2 B_1$, and the answer is :

$$\begin{array}{r}
 A_4 A_3 A_2 A_1 \\
 + B_4 B_3 B_2 B_1 \\
 \hline
 S_4 S_3 S_2 S_1
 \end{array}$$

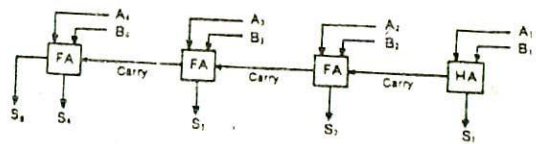


Figure 6.24. A parallel four-bit binary adder.

The first column requires only a half-adder. For any column above the first, there may be a carry from the preceding column. Therefore, we must use a full-adder for each column above the first.

To illustrate how the adder of Figure 6.24 works, let us see how it will add two numbers say 9 and 11. The binary equivalent of decimal 9 is 1001, and that of decimal 11 is 1011. Figure 6.25 shows the binary adder with these inputs.

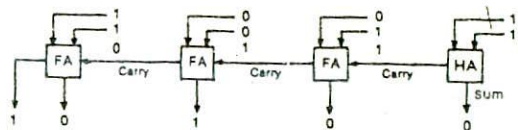


Figure 6.25. Example of adding two four-bit numbers using a parallel adder.

As shown in the figure, the half-adder adds 1+1 to give a sum of 0 and a carry 1. The carry goes into the first full-adder, which adds 0 + 1 + 1 to get a sum of 0 and a carry of 1. This carry goes into the next full-adder, which adds 0 + 0 + 1 to get a sum of 1 and a carry of 0. The last full-adder adds 1 + 1 + 0 to get a sum of 0 and a carry of 1. The final output of the system is 10100. The decimal equivalent of binary 10100 is 20 which is the correct decimal sum of 9 and 11.

The parallel binary adder of Figure 6.24 has limited capacity. The largest binary numbers that can be added using it are 1111 and 1111. So, its maximum capacity is :

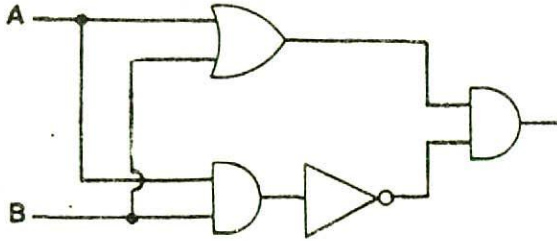
$$\begin{array}{r} 15 \quad 1111 \\ + 15 \quad + 1111 \\ \hline 30 \quad 11110 \end{array}$$

In order to increase the capacity, more full-adders can be connected to the left end of the system. For instance, to add six bit numbers, two more full-adders must be connected and for adding eight bit numbers, four more full-adders must be connected to the left end of the full-adder of Figure 6.24.

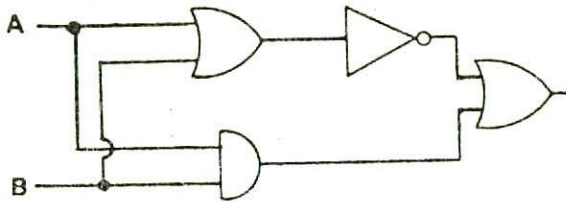
QUESTIONS

- Explain the principle of duality in Boolean algebra. How is it useful?
- Give the dual of the following Boolean expressions :
 - $\overline{A+B}$
 - $\overline{A+B+C}$
 - $\overline{A.B} + \overline{A.B}$
 - $\overline{A+B}$
 - $\overline{A.(A+B)}$
 - $\overline{A} + \overline{A.B}$
- Give the dual of the rule $A + \overline{A.B} = A + B$
- Prepare a truth table for the following Boolean expressions :
 - $\overline{A.B} + \overline{A.B}$
 - $\overline{A.B.C} + \overline{B.C}$
 - $\overline{A+B}$
 - $\overline{A+B+C}$
 - $\overline{A.B.C}$
 - $\overline{A.B.C} + \overline{A.B.C}$
 - $\overline{(A+B).(A+C).(B+C)}$
 - $\overline{A.C} + \overline{A.C}$
- State and prove the two basic De Morgan's theorems.
- Prove the following rules by the method of perfect induction.
 - $\overline{A.B} + A.B = A$
 - $A + \overline{A.B} = A + B$
 - $A . (A+C) = A$
 - $(A+B).(A.B) = \overline{A.B} + \overline{B.A}$
 - $(A+B).(A+B) = A.B + \overline{A.B}$
- Simplify the following Boolean expressions and draw logic circuit diagrams for your simplified expressions using AND, OR and NOT gates :
 - $\overline{x.y.z} + \overline{x.y.z} + \overline{x.y.z} + \overline{x.y.z}$
 - $\overline{x.y.z} + \overline{x.y.z} + \overline{x.y.z} + \overline{x.y.z}$
 - $\overline{A.C} + \overline{A.B} + \overline{A.B.C} + \overline{B.C}$
 - $\overline{A.B.C} + \overline{A.B.C} + \overline{A.B.C} + \overline{A.B.C} + \overline{A.B.C}$
 - $(A+B+C) . (A+\overline{B+C}) . (A+B+\overline{C}) . (A+\overline{B+C})$
 - $(A.B.C) . (A.B.C + \overline{A.B.C} + \overline{A.B.C})$
- Find the complement of the following expressions :
 - $\overline{A.B} + A.C$
 - $\overline{A.B} + A.B$
 - $(A+B) . (B+C) . (A+C)$
 - $A . (B.C + \overline{B.C})$
 - $A . (\overline{B+C})$
 - $A . (B+C) . (\overline{C+D})$
 - $A.B + (\overline{A.B}) . (B.C + \overline{B.C})$
- Express the following Boolean functions in their sum-of-products form. Ensure that each term has all the literals.
 - $\overline{A.(B+C)}$
 - $(A+B).(B+C)$
 - $(A.B).(A.B.C + \overline{A.C})$
 - $(\overline{A+C}).(\overline{A+B+C}).(A+B)$
 - $(A+B).\overline{C}$
 - $(\overline{A+C}).(A.B + A.C + B.C)$
- Express the following Boolean functions in their product-of-sums form. Ensure that each term has all the literals.
 - $\overline{A} + \overline{B.C}$
 - $A.B + \overline{C}$
 - $A + B + C$
 - $(\overline{A.B}).(\overline{A.C} + \overline{B.C})$
 - $(A.B).(B+C)$
 - $A + A.B + \overline{A.C}$

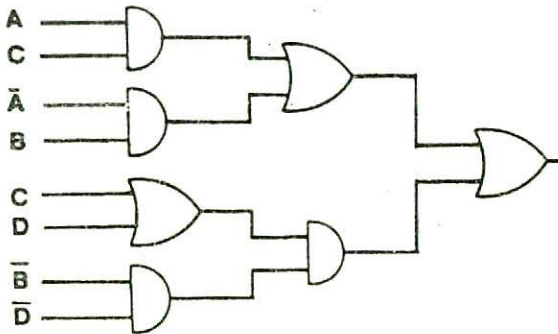
11. What will be the outputs of the following logic circuits for the specified inputs?



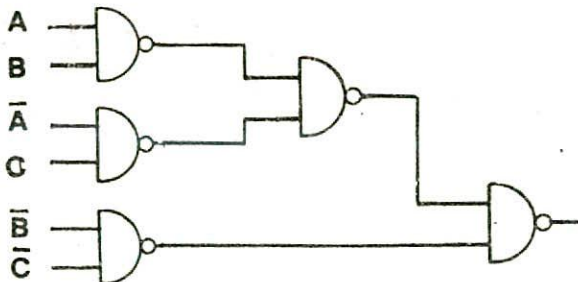
(a)



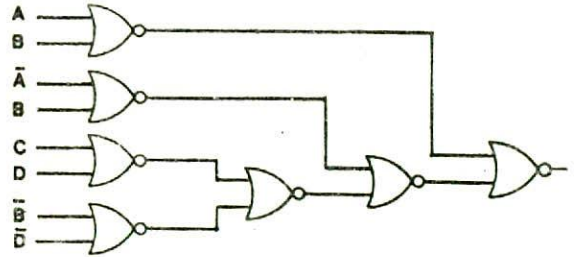
(b)



(c)



(d)



(e)

12. Construct logic circuit diagrams for the following Boolean expressions using AND/OR/NOT gates:

- (a) $A \cdot \bar{B} + A \cdot B$
- (b) $(A+B) \cdot (\bar{A} \cdot \bar{B})$
- (c) $(\bar{A} + \bar{B}) \cdot (A+C) \cdot (B + \bar{C})$
- (d) $A \cdot B + (\bar{A} \cdot \bar{B}) \cdot (B \cdot C + \bar{B} \cdot \bar{C})$
- (e) $(A+B) \cdot (A+C) \cdot (\bar{A} + \bar{B})$

13. "AND, OR and NOT gates are logically complete." Discuss.

14. Why are NAND and NOR gates called universal gates?

15. Show the implementation of the logical operations AND, OR and NOT only with NAND gates and only with NOR gates.

16. Construct logic circuit diagrams for the Boolean expressions of Question 12 using only NAND gates.

17. Construct logic circuit diagrams for the Boolean expressions of Question 12 using only NOR gates.

18. Construct logic circuit diagram for a half-adder using only NAND gates.

19. Construct logic circuit diagram for a half-adder using only NOR gates.

20. Why are combinational circuits more frequently constructed with NAND or NOR gates than with AND, OR and NOT gates?

21. Prove that

- (a) $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
- (b) $(A \oplus B) \oplus \bar{C} = A \oplus (B \oplus \bar{C})$

22. Construct a logic circuit diagram for the exclusive-or function using only NOR gates.
23. Construct a logic circuit diagram for the equivalence function using only NAND gates.
24. A logic circuit has three inputs A, B and C. It generates an output of 1 only when $A = 0, B = 1, C = 0$ or $A = 1, B = 1, C = 0$. Design a combinational circuit for this system.
25. A logic circuit has three inputs A, B and C. It generates an output of 1 only under the following conditions:
 $A = 0, B = 0, C = 0$
 $A = 0, B = 1, C = 1$
 $A = 1, B = 0, C = 1$
 $A = 1, B = 1, C = 1$
- Design a combinational circuit for this system.
26. Design a gating network which will have outputs 0 only when $A = 0, B = 0, C = 0$; $A = 1, B = 0, C = 0$; $A = 1, B = 1, C = 0$. The outputs are to be 1 for all other cases.
27. A three bit message is to be transmitted with an odd parity. An odd parity generator generates a parity bit (say P) so as to make the total number of 1's odd (including P). That is, $P = 1$ only when the number of 1's in the input string is even. Design a combinational logic circuit for such a parity generator.
28. Design a combinational logic circuit to generate an even parity for hexadecimal digits.

7. PRIMARY STORAGE

This chapter introduces the basic fundamentals related to the primary storage or the main memory of a computer system. In this chapter, you will learn about storage locations and addresses, how to determine the capacity of storage units, the difference between fixed and variable word-length storage organizations, and several other terms related to the main memory of a computer system.

Any storage unit of a computer system is ranked according to the following criteria :

1. *Access time.* This is the time required to locate and retrieve stored data from the storage unit in response to a program instruction. A fast access time is preferred.
2. *Storage capacity.* It is the amount of data that can be stored in the storage unit. A large capacity is desired.
3. *Cost per bit of storage.* An obvious goal is to minimize this cost.

Based on the above mentioned criteria, storage units are basically of two types - primary and secondary. As

compared to secondary storage units, primary storage units have faster access time, smaller storage capacity, and higher cost per bit of storage. In this chapter, we will be concentrating only on the concepts of primary storage. We shall learn about different types of secondary storage devices in the next chapter.

STORAGE LOCATIONS AND ADDRESSES

A primary or internal storage section is basic to all computers. It is made up of several small storage areas called locations or cells. Each of these locations can store a fixed number of bits called *word length* of that particular primary storage. Thus, as shown in Figure 7.1, a given memory is divided into N words, where N generally is some power of 2. Each word or location has a built-in and unique number assigned to it. This number is called the *address* of the location and is used to identify the location. Each location can hold either a data item or an instruction, and its address remains the same regardless of its contents. The addresses normally start at 0 and the highest address equals the number of words that can be stored in the memory

minus 1.